This book is affectionately dedicated
to the Type 650 computer once installed at
Case Institute of Technology,
with whom I have spent many pleasant evenings.

DONALD E. KNUTH *Stanford University*

ADDISON-WESLEY PUBLISHING COMPANY

Volume 1 / **Fundamental Algorithms**

THE ART OF COMPUTER PROGRAMMING

Reading, Massachusetts · Menlo Park, California · London · Don Mills, Ontario

This book is in the
ADDISON-WESLEY SERIES IN
COMPUTER SCIENCE AND INFORMATION PROCESSING

RICHARD S. VARGA and MICHAEL A. HARRISON, Editors

Second Printing 1969

PREFACE

Here is your book, the one your thousands of letters have asked us to publish. It has taken us years to do, checking and rechecking countless recipes to bring you only the best, only the interesting, only the perfect. Now we can say, without a shadow of a doubt, that every single one of them, if you follow the directions to the letter, will work for you exactly as well as it did for us, even if you have never cooked before.
—*McCall's Cookbook* (1963)

The process of preparing programs for a digital computer is especially attractive because it not only can be economically and scientifically rewarding, it can also be an aesthetic experience much like composing poetry or music. This book is the first volume of a seven-volume set of books that has been designed to train the reader in the various skills which go into a programmer's craft.

The following chapters are *not* meant to serve as an introduction to computer programming; the reader is supposed to have had some previous experience. The prerequisites are actually very simple, but a beginner requires time and practice before he properly understands the concept of a digital computer. The reader should possess:

a) Some idea of how a stored-program digital computer works; not necessarily the electronics, rather the manner in which instructions can be kept in the machine's memory and successively executed. Previous exposure to machine language will be helpful.

b) An ability to put the solutions to problems into such explicit terms that a computer can "understand" them. (These machines have no common sense; they have not yet learned to "think," and they do exactly as they are told, no more and no less. This fact is the hardest concept to grasp when one first tries to use a computer.)

c) Some knowledge of the most elementary computer techniques, such as looping (performing a set of instructions repeatedly), the use of subroutines, and the use of index registers.

d) A little knowledge of common computer jargon, e.g. "memory," "registers," "bits," "floating point," "overflow." Most words not defined in the text are given brief definitions in the index at the close of each volume.

These four prerequisites can perhaps be summed up into the single requirement that the reader should have already written and tested at least, say, four programs for at least one computer.

I have tried to write this set of books in such a way that it will fill several needs. In the first place, these books are reference books which summarize the knowledge which has been acquired in several important fields. They can also be used as textbooks for self-study or for college courses in the computer and information sciences. To meet both of these objectives, I have incorporated a large number of exercises into the text and have furnished answers for most of them; I have also made an effort to fill the pages with facts rather than with vague, general commentary.

This set of books is intended for people who will be more than just casually interested in computers, yet it is by no means only for the computer specialist. Indeed, one of the main goals has been to make these programming techniques more accessible to the many people working in other fields who can make fruitful use of computers, yet who cannot afford the time to locate all of the necessary information which is buried in the technical journals.

The subject of these books might be called "nonnumerical analysis." Although computers have traditionally been associated with the solution of numerical problems such as the calculation of the roots of an equation, numerical interpolation and integration, etc., topics like this are not treated here except in passing. Numerical computer programming is a very interesting and rapidly expanding field, and many books have been written about it. In recent years, however, a good deal of interesting work has been done using computers for essentially nonnumerical problems, such as sorting, translating languages, solving mathematical problems in higher algebra and combinatorial analysis, theorem proving, the development of "software" (programs to facilitate the writing of other programs), and the simulation of various processes from everyday life. Numbers occur in such problems only by coincidence, and the computer's decision-making capabilities are being used rather than its ability to do arithmetic. In nonnumerical problems, we have some use for addition and subtraction, but we rarely feel any need for multiplication and division. Note, however, that even a person who is primarily concerned with numerical computer programming will benefit from a study of the nonnumerical techniques, for these are present in the background of numerical programs as well.

The results of the recent research in nonnumerical analysis are scattered throughout numerous technical journals, and at the time of writing they are in a somewhat chaotic and disorganized state. The approach used here has been to study those techniques which are most basic, in the sense that they can be applied to many types of programming situations; I have attempted to coordinate these into more or less of a "theory," and to bring the reader up to the present frontiers of knowledge in these areas. Applications of these basic techniques to the design of software programs are also given.

Of course, "nonnumerical analysis" is a terribly negative name for this field of study, and it would be much better to have a positive, descriptive term which characterizes the subject. "Information processing" is too broad a designation for the material I am considering, and "programming techniques" is too narrow. Therefore I wish to propose *analysis of algorithms* as an appropriate name for the subject matter covered in these books; as explained more fully in the books themselves, this name is meant to imply "the theory of the properties of particular computer algorithms."

It is generally very difficult to keep up with a field that is economically profitable, and so it is only natural to expect that many of the techniques described here will eventually be superseded by better ones. It has, of course, been impossible for me to keep "two years ahead of the state of the art," and the frontiers mentioned above will certainly change. I have mixed emotions in this respect, since I certainly hope this set of books will stimulate further research, yet not so much that the books themselves become obsolete!

Actually the majority of the algorithms presented here have already been in use for five years or more by quite a number of different people, and so in a sense these methods have matured to the point where they are now reasonably well understood and are presumably in their best form. It is no longer premature, therefore, to put them into a textbook and to expect students to learn about them.

The complete seven-volume set of books, entitled *The Art of Computer Programming*, has the following general outline:

Volume 6. Theory of Languages

Chapter 11. Mathematical Linguistics

Volume 7. Compilers

Chapter 12. Programming Language Translation

I started out in 1962 to write a single book with this sequence of chapters, but I soon found that it was more important to treat the subjects in depth rather than to skim over them lightly. The resulting length of the text has meant that each chapter by itself contains enough material for a one-semester college course, so it has become sensible to publish the series in separate volumes instead of making it into one or two huge tomes. (It may seem strange to have only one or two chapters in an entire book, but I have decided to retain this chapter numbering to facilitate cross-references A shorter version of Volumes 1 through 5 will soon be published, intended specifically to serve as a more general textbook for undergraduate computer courses. Its contents will be a "subset" of the material in these books, with the more specialized information omitted; I intend to use the same chapter numbering in this abridged edition.)

The present volume may be considered as the "intersection" of the entire set of books, in the sense that it contains the basic material which is used in all the other volumes. Volumes 2 through 7, on the other hand, may be read independently of each other, except perhaps for some strong connections between Volumes 5 and 7. Volume 1 is not only a reference book to be used in connection with Volumes 2 through 7; it may also be used in college courses or for self-study as a text on the subject of *data structures* (emphasizing the material of Chapter 2), or as a text on the subject of *discrete mathematics* (emphasizing the material of Sections 1.1, 1.2, 1.3.3, and 2.3.4), or as a text on the subject of *machine-language programming* (emphasizing the material of Sections 1.3 and 1.4).

The point of view I have adopted while writing these twelve chapters differs from that taken in many contemporary books about computer programming in that I am not trying to teach the reader how to use somebody else's subroutines; I am concerned rather with teaching the reader how to write better subroutines himself!

A few words are in order about the mathematical content of this set of books. The material has been organized so that persons with no more than a knowledge of high school algebra may read it, skimming briefly over the more mathematical portions; yet a reader who is mathematically inclined will learn about many interesting mathematical techniques related to "discrete mathematics." This dual level of presentation has been achieved in part by assigning "ratings" to each of the exercises so that those which are primarily mathematical are marked specifically as such, and also by arranging most sections so that the

main mathematical results are stated *before* their proofs. The proofs are either left as exercises (with answers to be found in a separate section) or they are given at the end of a section.

A reader who is interested primarily in programming rather than in the associated mathematics may stop reading each section as soon as the mathematics becomes recognizably difficult. On the other hand, a mathematically oriented reader will find a wealth of interesting material collected here. Much of the published mathematics about computer programming has been very faulty, and one of the purposes of this book is to instruct readers in proper mathematical approaches to this subject. Since I myself profess to be a mathematician, it is my duty to maintain mathematical integrity as well as I can.

A knowledge of elementary calculus will suffice for most of the mathematics in these books, since most of the other theory that is needed is developed herein; there are some isolated places, however, in which deeper theorems of complex variable theory, probability theory, number theory, etc. are quoted when appropriate.

Even though computers are widely regarded as belonging to the domain of "applied mathematics," there are "pure mathematicians" such as myself who have found many intriguing connections between computers and abstract mathematics. From this standpoint, parts of these books may be thought of as "a pure mathematician's view of computers."

To a layman, the electronic computer has come to symbolize the importance of mathematics in today's world, yet few professional mathematicians are now closely acquainted with the machines. One reason for this surprising (and unfortunate) situation is that computers seem to have made some things "too easy," in the sense that people who no longer have to do so many things with pencil and paper never discover the mathematical simplifications which would aid the work. Some mathematicians occasionally resent the intrusion of computers, not because they are afraid they will lose their jobs to automation, but because they fear there will perhaps be less necessity to give birth to invention. On the other hand, there are obvious relations between computers and mathematics in the fields of numerical analysis, number theory, and statistics.

I wish to show that the connection between computers and mathematics is far deeper and more intimate than these traditional relationships would imply. The construction of a computer program from a set of basic instructions is very similar to the construction of a mathematical proof from a set of axioms. Furthermore, pure mathematical problems historically have always developed from the study of practical problems arising in another field, and the advent of computers has brought a number of these with it. Some of the problems investigated in these books which are essentially of this type are (a) the study of stochastic properties of particular algorithms: determination of how well they may be expected to perform; (b) the construction of optimal algorithms, e.g., for sorting or for evaluating polynomials; and (c) the theory of languages.

Besides the interesting application of mathematical tools to programming problems, there are also interesting applications of computers to the exploration of mathematical conjectures, e.g., in combinatorial analysis and algebra; and in many of these cases there is considerable interplay between programming and classical mathematics. Attempts at mechanization of mathematics are also very important, since they lead to a greater understanding of concepts we thought we knew (until we had to explain them to a computer). I believe the connections between computers and pure mathematics which have been enumerated in this paragraph will become increasingly important.

The hardest decision which I had to make while preparing these books concerned the manner in which to present the various techniques. The advantages of flowcharts and of an informal step-by-step description of an algorithm are well known; for a discussion of this, see the article "Computer-Drawn Flowcharts" in the ACM *Communications*, Vol. 6 (September, 1963), pages 555–563. Yet a formal, precise language is also necessary to specify any computer algorithm, and I needed to decide whether to use an algebraic language, such as ALGOL or FORTRAN, or to use a machine-oriented language for this purpose. Perhaps many of today's computer experts will disagree with my decision to use a machine-oriented language, but I have become convinced that it was definitely the correct choice, for the following reasons:

a) Algebraic languages are more suited to numerical problems than to the nonnumerical problems considered here; although programming languages are gradually improving, today's languages are not yet appropriate for topics such as coroutines, input-output buffering, generating random numbers, multiple-precision arithmetic, and many problems involving packed data, combinatorial searching, and recursion, which appear throughout.

b) A programmer is greatly influenced by the language in which he writes his programs; there is an overwhelming tendency to prefer constructions which are simplest in that language, rather than those which are best for the machine. By writing in a machine-oriented language, the programmer will tend to use a much more suitable method; it is much closer to reality.

c) The programs we require are, with a few exceptions, all rather short, so with a suitable computer there will be no trouble understanding the programs.

d) A person who is more than casually interested in computers should be well schooled in machine language, since it is a fundamental part of a computer.

e) Some machine language would be necessary anyway as output of the software programs described in Chapters 1, 9, 10, and 12.

From the other point of view, it is admittedly somewhat easier to write programs in higher-level programming languages, and it is considerably easier to check out the programs; thus there is a large class of problems for which the algebraic languages are much more desirable, even though the actual machine language which corresponds to an algebraic language program is usually far from its best possible form. Many of the problems of interest to us in this book, however, are those for which the programmer's art is most important; for example, with programs such as software routines, which are used so many times each day in a computer installation, it is worth while to put an additional effort into the writing of the program, since these programs need be written only once.

Given the decision to use a machine-oriented language, which language should be used? I could have chosen the language of a particular machine X, but then those people who do not possess machine X would think this book is only for X-people. Furthermore, machine X probably has a lot of idiosyncrasies which are completely irrelevant to the material in this book yet which must be explained; and in two years the manufacturer of machine X will put out machine $X + 1$ or machine $10X$, and machine X will no longer be of interest to anyone. (Of course, if I invent a hypothetical computer, it may *already* be of interest to no one!)

To avoid this dilemma, I have attempted to design an "ideal" computer called "MIX," with very simple rules of operation (requiring, say, only an hour to learn), and which is also very much like nearly every computer now in existence. Thus MIX programs can be readily adapted to most actual machines, or simulated on most machines.

There is no reason why a student should be afraid of learning the characteristics of more than one computer; indeed, he may expect to meet many different machine languages in the course of his life, and once one machine language has been mastered, others are easily assimilated. So the only remaining disadvantage of a mythical machine is that it is difficult to execute any programs written for it. (For this purpose it is recommended that college instructors have a MIX simulator available for running the students' exercises. Such a simulator has the advantage that automatic grading routines can easily be incorporated, but it has the obvious disadvantage that it will take a few days' work to prepare such a program. In order to simplify this task, Chapter 1 contains a MIX simulator written in its own language, and this program can be readily modified for a similar machine.)

Fortunately, the field of computer science is still young enough to permit a rather thorough study. I have tried to the best of my ability to scrutinize all of the literature published so far about the topics treated in this set of books, and indeed I have also read a great deal of the unpublished literature; but of course I cannot claim to have covered the subject completely. I have written

numerous letters in an attempt to establish correctly the history of the important ideas discussed in each chapter. In any work of this size, however, there are bound to be a number of errors of omission and commission, in spite of the extensive checking for accuracy that has been made. In particular, I wish to apologize to anyone who might have been unintentionally slighted in the historical sections. I will greatly appreciate receiving information about any errors noticed by the readers, so that these may be corrected as soon as possible in future editions.

I have attempted to present an annotated bibliography of the best papers currently available in each subject, and I have tried to choose terminology that is concise and consistent with current usage. In referring to the literature, the names of periodicals are given with standard abbreviations, except for the most commonly cited journals, for which the following abbreviations are used:

CACM = Communications of the Association for Computing Machinery

JACM = Journal of the Association for Computing Machinery

Comp. J. = The Computer Journal (British Computer Society)

Math. Comp. = Mathematics of Computation

AMM = American Mathematical Monthly

As an example, "*CACM* **6** (1963), 555–563" stands for the reference given in a preceding paragraph of this preface.

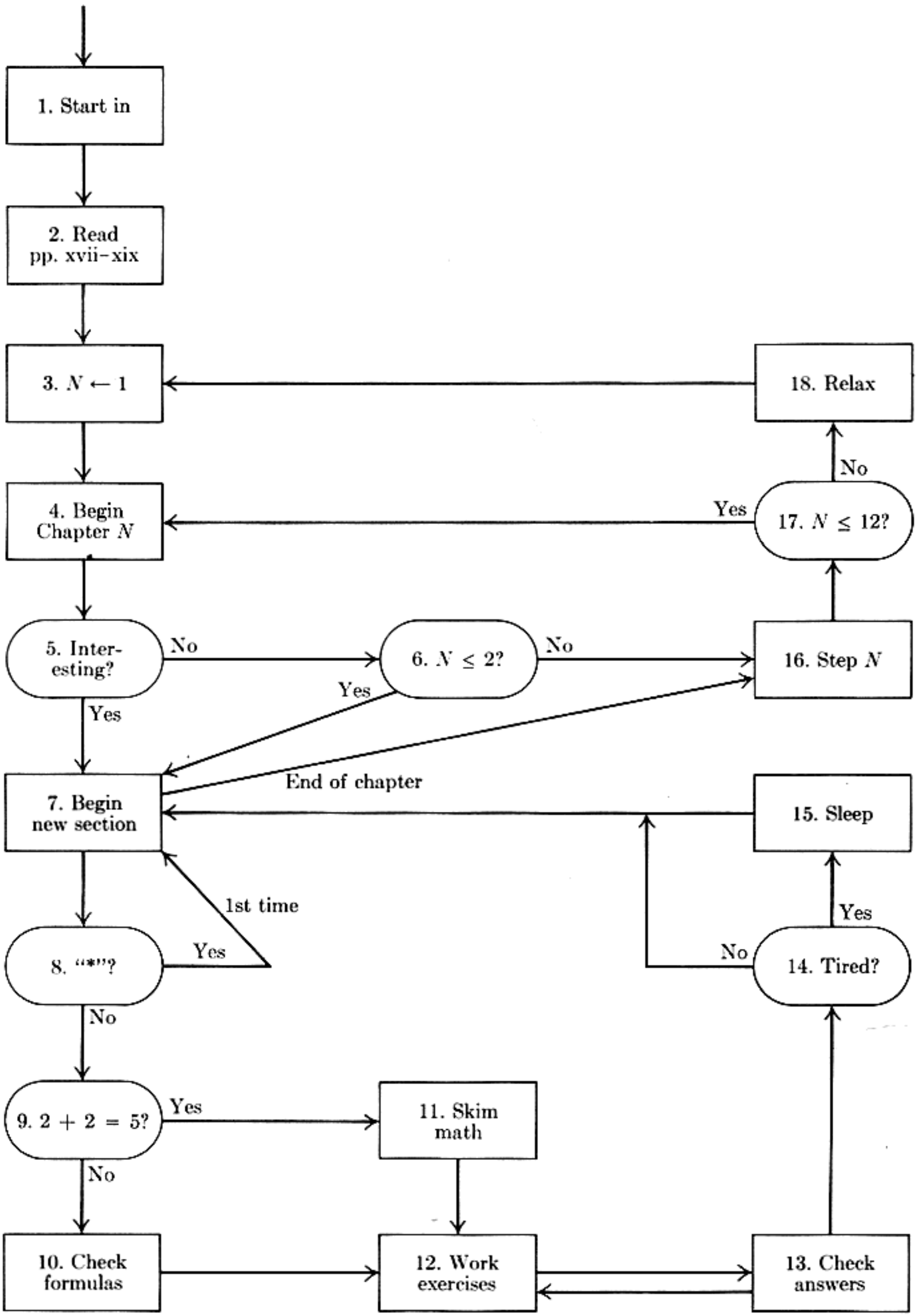
I have, of course, received assistance from a great many people during the five years I have been preparing these books, and for this I am extremely thankful. Acknowledgments are due, first, to my wife, Jill, for her infinite patience, for being the first guinea pig in reading the manuscript, and for untold further assistance of all kinds; secondly, to the ElectroData Division of the Burroughs Corporation, for the use of its B220 and B5500 computers in the testing of most of the programs in these books and the preparation of most of the tables, and also for the use of its excellent library of computer literature; also to the California Institute of Technology, for its encouragement and its excellent students; to the National Science Foundation and the Office of Naval Research, for supporting part of the work; to my father, Ervin Knuth, for assistance in the preparation of the manuscript; and to the Addison-Wesley Publishing Company for the wonderful cooperation which made these books possible.

It has been a great pleasure working together with Robert W. Floyd, of Carnegie Institute of Technology, who from the beginning has contributed a great deal of his time towards the enhancement of these books. Other people whose technical assistance I have found very valuable include J. D. Alanen, Webb T. Comfort, Melvin E. Conway, N. G. de Bruijn, R. P. Dilworth, James R. Dunlap, David E. Ferguson, Joel N. Franklin, H. W. Gould, Dennis E. Hamilton, Peter Z. Ingerman, Edgar T. Irons, William C. Lynch, Daniel D.

McCracken, John L. McNeley, Jack N. Merner, Howard H. Metcalfe, Peter Naur, William W. Parker, W. W. Peterson, Paul Purdom, James C. Robertson, Douglas T. Ross, D. V. Schorre, M. P. Schützenberger, E. J. Schweppe, Christopher J. Shaw, Donald L. Shell, Olga Taussky, John Todd, Michael Woodger, John W. Wrench, Jr., and W. W. Youden. Many of these people have kindly allowed me to make use of some of their hitherto unpublished work.

Pasadena, California D. E. K.
October 1967

(*Author's note:* In this second printing, I have incorporated several hundred minor improvements while retaining the original page numbering.)



Flow chart for reading this set of books.

Procedure for Reading This Set of Books

1. Begin reading this procedure, unless you have already begun to read it. *Continue to follow the steps faithfully.* (The general form of this procedure and its accompanying flowchart will be used throughout this book.)

2. Read the Notes on the Exercises, pp. xvii–xix.

3. Set N equal to 1.

4. Begin reading chapter N. Do *not* read the quotations which appear at the beginning of the chapter.

5. Is the subject of the chapter interesting to you? If so, go to step 7; if not, go to step 6.

6. Is $N \leq 2$? If not, go to step 16; if so, scan through the chapter anyway. (Chapters 1 and 2 contain important introductory material and also a review of basic programming techniques. You should at least skim over the sections on notation and about MIX.)

7. Begin reading the next section of the chapter; if you have reached the end of the chapter, go to step 16.

8. Is section number marked with "*"? If so, you may omit this section on first reading (it covers a rather specialized topic which is interesting but not essential); go back to step 7.

9. Are you mathematically inclined? If math is all Greek to you, go to step 11; otherwise go to step 10.

10. Check the mathematical derivations made in this section (and report errors to the author). Go to step 12.

11. If the current section is full of mathematical computations, you had better omit reading the derivations. However, you should become familiar with the basic results of the section; these are usually stated near the beginning or in italics right at the very end of the hard parts.

12. Work the recommended exercises in this section in accordance with the hints given in the Notes on the Exercises (which you read in step 2).

13. After you have worked on the exercises to your satisfaction, check your answers with the answer printed in the corresponding answer section at the rear of the book (if any answer appears for that problem). Also read the answers to the exercises you did not have time to work. *Note:* In most cases it is reasonable to read the answer to exercise n before working on exercise $n + 1$, so steps 12–13 are usually done simultaneously.

14. Are you tired? If not, go back to step 7.

15. Go to sleep. Then, wake up, and go back to step 7.

16. Increase N by one. If $N = 3, 5, 7, 9, 11$, or 12, begin the next volume of this set of books.

17. If N is less than or equal to 12, go back to step 4.

18. Congratulations. Now try to get your friends to purchase a copy of volume one and to start reading it. Also, go back to step 3.

Notes on the Exercises

The exercises in this set of books have been designed for self-study as well as classroom study. It is difficult, if not impossible, for anyone to learn a subject purely by reading about it, without applying the information to specific problems and thereby forcing himself to think about what has been read. Furthermore, we all learn best the things that we have discovered for ourselves. Therefore the exercises form a major part of this work; a definite attempt has been made to keep them as informative as possible and to select problems that are enjoyable to solve.

In many books, easy exercises are found mixed randomly among extremely difficult ones. This is sometimes unfortunate because the reader should have some idea about how much time it ought to take him to do a problem before he tackles it (otherwise he may just skip over all the problems). A classic example of this situation is the book *Dynamic Programming* by Richard Bellman; this is an important, pioneering book in which a group of problems is collected together at the end of some chapters under the heading "Exercises and Research Problems," with extremely trivial questions appearing in the midst of deep, unsolved problems. It is rumored that someone once asked Dr. Bellman how to tell the exercises apart from the research problems, and he replied, "If you can solve it, it is an exercise; otherwise it's a research problem."

Good arguments can be made for including both research problems and very easy exercises in a book of this kind; therefore, to save the reader from the possible dilemma of determining which are which, *rating numbers* have been provided to indicate the level of difficulty. These numbers have the following general significance:

Rating	*Interpretation*
00	An extremely easy exercise which can be answered immediately if the material of the text has been understood, and which can almost always be worked "in your head."
10	A simple problem, which makes a person think over the material just read, but which is by no means difficult. It should be possible to do this in one minute at most; pencil and paper may be useful in obtaining the solution.
20	An average problem which tests basic understanding of the text material but which may take about fifteen to twenty minutes to answer completely.

30 A problem of moderate difficulty and/or complexity which may involve over two hours' work to solve satisfactorily.

40 Quite a difficult or lengthy problem which is perhaps suitable for a term project in classroom situations. It is expected that a student will be able to solve the problem in a reasonable amount of time, but the solution is not trivial.

50 A research problem which (to the author's knowledge at the time of writing) has not yet been solved satisfactorily. If the reader has found an answer to this problem, he is urged to write it up for publication; furthermore, the author of this book would appreciate hearing about the solution as soon as possible (provided it is correct)!

By interpolation in this "logarithmic" scale, the significance of other rating numbers becomes clear. For example, a rating of *17* would indicate an exercise that is a bit simpler than average. Problems with a rating of *50* which are subsequently solved by some reader may appear with a *45* rating in later editions of the book.

The author has earnestly tried to assign accurate rating numbers, but it is difficult for the person who makes up a problem to know just how formidable it will be for someone else; and everyone has more aptitude for certain types of problems than for others. It is hoped that the rating numbers represent a good guess as to the level of difficulty, but they should be taken as general guidelines, not as absolute indicators.

This book has been written for readers with varying degrees of mathematical training and sophistication; and, as a result, some of the exercises are intended only for the use of more mathematically inclined readers. Therefore the rating is preceded by an *M* if the exercise involves mathematical concepts or motivation to a greater extent than necessary for someone who is primarily interested in only the programming algorithms themselves. An exercise is marked with the letters "*HM*" if its solution necessarily involves a knowledge of calculus or other higher mathematics not developed in this book. An "*HM*" designation does *not* necessarily imply difficulty.

Some exercises are preceded by an arrowhead, "▸"; this designates problems which are especially instructive and which are especially recommended. Of course, no reader/student is expected to work *all* of the exercises, and so those which are perhaps the most valuable have been singled out. This is not meant to detract from the other exercises! Each reader should at least make an attempt to solve all of the problems whose rating is *10* or less; and the arrows may help in deciding which of the problems with a higher rating should be given priority.

Solutions to most of the exercises appear in the answer section. Please use them wisely; do not turn to the answer until you have made a genuine effort to solve the problem by yourself, or unless you do not have time to work this particular problem. *After* getting your own solution or giving the problem a

decent try, you may find the answer instructive and helpful. The solution given will often be quite short, and it will sketch the details under the assumption that you have earnestly tried to solve it by your own means first. Sometimes the solution gives less information than was asked; often it gives more. It is quite possible that you may have a better answer than the one published here, or you may have found an error in the published solution; in such a case, the author will be pleased to know the details as soon as possible. Later editions of this book will give the improved solutions together with the solver's name where appropriate.

Summary of codes:

▸	Recommended	*00*	Immediate
M	Mathematically oriented	*10*	Simple (one minute)
HM	Requiring "higher math"	*20*	Medium (quarter hour)
		30	Moderately hard
		40	Term project
		50	Research problem

EXERCISES

▶ **1.** [*00*] What does the rating "*M20*" mean?

2. [*10*] Of what value can the exercises in a textbook be to the reader?

3. [*M50*] Prove that when n is an integer, $n > 2$, the equation $x^n + y^n = z^n$ has no solution in positive integers x, y, z.

CONTENTS

CHAPTER ONE

BASIC CONCEPTS

Many persons who are not conversant with mathematical studies imagine that because the business of [Babbage's Analytical Engine] is to give its results in numerical notation, the nature of its processes must consequently be arithmetical and numerical, rather than algebraical and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they were letters or any other general symbols; and in fact it might bring out its results in algebraical notation, were provisions made accordingly.

— ADA AUGUSTA, Countess of Lovelace (1844)

Wherever the term 'computer' or 'digital computer' appears throughout the text, replace it by the term 'Data Processor.'

— from a list of errata for a digital computer reference manual (1957)

1.1. ALGORITHMS

The notion of an *algorithm* is basic to all of computer programming, so we should begin with a careful analysis of this concept.

The word "algorithm" itself is quite interesting; at first glance it may look as though someone intended to write "logarithm" but jumbled up the first four letters. The word did not appear in *Webster's New World Dictionary* as late as 1957; we find only the older form "algorism" with its ancient meaning, i.e., the process of doing arithmetic using Arabic numerals. In the middle ages, abacists computed on the abacus and algorists computed by algorism. Following the middle ages, the origin of this word was in doubt, and early linguists attempted to guess at its derivation by making combinations like *algiros* [painful] + *arithmos* [number]; others said no, the word comes from "King Algor of Castile." Finally, historians of mathematics found the true origin of the word algorism: it comes from the name of a famous Arabic textbook author, Abu Ja'far Mohammed ibn Mûsâ al-Khowârizmî (*c.* 825)—literally, "Father of Ja'far, Mohammed, son of Moses, native of Khowârizm." Khowârizm is today the small Soviet city of Khiva. Al-Khowârizmî wrote the celebrated book *Kitab al jabr w'al-muqabala* ("Rules of restoration and reduction"); another word, "algebra," stems from the title of his book, although the book wasn't really very algebraic.

Gradually the form and meaning of "algorism" became corrupted; as explained by the Oxford English Dictionary, the word was "erroneously refashioned" by "learned confusion" with the word *arithmetic*. The change from "algorism" to "algorithm" is not hard to understand in view of the fact that people had forgotten the original derivation of the word. An early German mathematical dictionary, *Vollständiges Mathematisches Lexicon* (Leipzig, 1747), gives the following definition for the word *Algorithmus:* "Under this designation are combined the notions of the four types of arithmetic calculations, namely addition, multiplication, subtraction, and division." The latin phrase *algorithmus infinitesimalis* was at that time used to denote "ways of calculation with infinitely small quantities, as invented by Leibnitz."

By 1950, the word algorithm was most frequently associated with "Euclid's algorithm," a process for finding the greatest common divisor of two numbers which appears in Euclid's *Elements* (book vii, propositions i and ii). It will be instructive to exhibit Euclid's algorithm here:

Algorithm E (*Euclid's algorithm*). Given two positive integers m and n, find their greatest common divisor, i.e., the largest positive integer which evenly divides both m and n.

E1. [Find remainder.] Divide m by n and let r be the remainder. (We will have $0 \leq r < n$.)

E2. [Is it zero?] If $r = 0$, the algorithm terminates; n is the answer.

E3. [Interchange.] Set $m \leftarrow n$, $n \leftarrow r$, and go back to step E1. ∎

Of course, Euclid did not present his algorithm in just this manner. The above format illustrates the style in which all of the algorithms throughout this book will be presented.

Each algorithm we consider has been given an identifying letter (e.g., E in the above) and the steps of the algorithm are identified by this letter followed by a number (e.g., E1, E2, etc.). The chapters are divided into numbered sections; within a section the algorithms are designated by letter only, but when algorithms are referred to in other sections, the appropriate section number is also used. For example, we are now in Section 1.1; within this section Euclid's algorithm is called Algorithm E, while in later sections it is referred to as Algorithm 1.1E.

Each step of an algorithm (e.g., step E1 above) begins with a phrase in brackets which sums up as briefly as possible the principal content of that step. This phrase also usually appears in an accompanying *flow chart* (e.g., Fig. 1), so the reader will be able to picture the algorithm more readily.

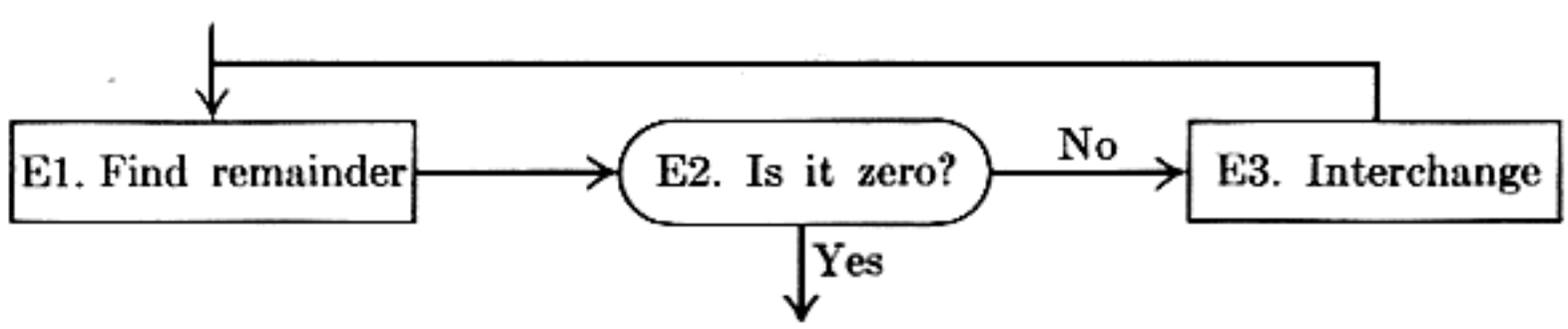


Fig. 1. Flow chart for Algorithm E.

After the summarizing phrase comes a description in words and symbols of some *action* to be performed or some decision to be made. There are also occasionally *parenthesized comments* (e.g., the second sentence in step E1) which are included as explanatory information about that step, often indicating certain characteristics of the variables or the current goals at that step, etc.; the parenthesized remarks do not specify actions which belong to the algorithm, they are only for the reader's benefit as possible aids to comprehension.

The "$\leftarrow$" arrow in step E3 is the all-important *replacement* operation (sometimes called *assignment* or *substitution*); "$m \leftarrow n$" means the value of variable m is to be replaced by the current value of variable n. When algorithm E begins, the values of m and n are the originally given numbers; but when it ends, these variables will have, in general, different values. An arrow is used to distinguish the replacement operation from the equality relation: We will not say, "Set $m = n$," but we will perhaps ask, "Does $m = n$?" The "$=$" sign denotes a condition which can be tested, the "$\leftarrow$" sign denotes an action which can be performed. The operation of *increasing n by one* is denoted by "$n \leftarrow n + 1$" (read "n is replaced by $n + 1$"); in general, "variable $\leftarrow$ formula" means the formula is to be computed using the present values of any variables appearing within it, and the result replaces the previous value of the variable at the left of the arrow. Persons untrained in computer work sometimes have a tendency to denote the operation of increasing n by one by "$n \rightarrow n + 1$," saying "n becomes $n + 1$"; this can only lead to confusion because of its conflict with the standard conventions, and it should be avoided.

Note that the order of the actions in step E3 is important; "set $m \leftarrow n$, $n \leftarrow r$" is quite different from "set $n \leftarrow r$, $m \leftarrow n$," since the latter would imply that the previous value of n is lost before it can be used to set m. Thus the latter operation is equivalent to "set $n \leftarrow r$, $m \leftarrow r$." When several variables are all to be set equal to the same quantity, we use multiple arrows; thus "$n \leftarrow r$, $m \leftarrow r$" may be written as "$n \leftarrow m \leftarrow r$." To interchange the values of two variables, we can write "Exchange $m \leftrightarrow n$"; this action may also be specified by using a new variable t and writing "set $t \leftarrow m$, $m \leftarrow n$, $n \leftarrow t$."

An algorithm starts at the lowest-numbered step, usually step 1, and steps are executed in sequential order, unless otherwise specified. In step E3, the imperative "go back to step E1" specifies the computational order in an obvious fashion. In step E2, the action is prefaced by the condition "if $r = 0$"; so if $r \neq 0$, the rest of that sentence does not apply and no action is specified. We might have added the redundant sentence, "If $r \neq 0$, go on to step E3."

The heavy vertical line, "∎", appearing at the end of step E3 is used to indicate the end of an algorithm and the resumption of text.

We have now discussed virtually all the notational conventions used in the algorithms of this book, except for a notation used to denote "subscripted" or "indexed" items which are elements of an ordered array. Suppose we have n quantities, $v_1, v_2, \ldots, v_n$; instead of writing v_j for the jth element, the notation $v[j]$ is often used. Similarly, $a[i, j]$ is sometimes used in preference to a doubly-subscripted notation like a_{ij}. Sometimes multiple-letter names are used for

variables and are usually set in capital letters, e.g., `TEMP` might be the name of a variable used for temporarily holding a computed value, `PRIME[K]` might denote the `K`th prime number, etc.

So much for the *form* of algorithms; now let us *perform* one. It should be mentioned immediately that the reader should *not* expect to read an algorithm as he reads a novel; such an attempt would make it pretty difficult to understand what is going on. An algorithm must be seen to be believed, and the best way to learn what an algorithm is all about is to try it. The reader should always take pencil and paper and work through an example of each algorithm immediately upon encountering it in the text. Usually the outline of a worked example will be given, or else the reader can easily conjure up one. This is a simple and painless method for obtaining an understanding of a given algorithm, and all other approaches are generally unsuccessful.

Let us therefore work out an example of Algorithm E. Suppose that we are given $m = 119$ and $n = 544$; we are ready to begin, at step E1. (The reader should now follow the algorithm as we give a play-by-play account.) Dividing m by n in this case is quite simple, almost too simple, since the quotient is zero and the remainder is 119. Thus, $r \leftarrow 119$. We proceed to step E2, and since $r \neq 0$ no action occurs. In step E3 we set $m \leftarrow 544$, $n \leftarrow 119$. It is clear that if $m < n$ originally, the quotient in step E1 will always be zero and the algorithm will always proceed to interchange m and n in this rather cumbersome fashion. We could add a new step:

"E0. [Ensure $m \geq n$.] If $m < n$, exchange $m \leftrightarrow n$."

if desired, without making an essential change in the algorithm except to increase its length as well as to decrease the time required to perform it in about one half of the cases.

Back at step E1, we find that $\frac{544}{119} = 4\frac{68}{119}$, so $r \leftarrow 68$. Again E2 is inapplicable, and at E3 we set $m \leftarrow 119$, $n \leftarrow 68$. The next round sets $r \leftarrow 51$, and ultimately $m \leftarrow 68$, $n \leftarrow 51$. Next $r \leftarrow 17$, and $m \leftarrow 51$, $n \leftarrow 17$. Finally, when 51 is divided by 17, $r \leftarrow 0$, so at step E2 the algorithm terminates. The greatest common divisor of 119 and 544 is 17.

So this is an algorithm. The modern meaning for algorithm is quite similar to that of *recipe*, *process*, *method*, *technique*, *procedure*, *routine*, except that the word "algorithm" connotes something just a little different. Besides merely being a finite set of rules which gives a sequence of operations for solving a specific type of problem, an algorithm has five important features:

1) Finiteness. An algorithm must always terminate after a finite number of steps. Algorithm E satisfies this condition, because after step E1 the value of r is *less* than n, so if $r \neq 0$, the value of n *decreases* the next time that step E1 is encountered. A decreasing sequence of positive integers must eventually terminate, so step E1 is executed only a finite number of times for any given original value of n. Note, however, that the number of steps can become arbi-

trarily large; certain huge choices of m and n will cause step E1 to be executed over a million times.

(A procedure which has all of the characteristics of an algorithm except that it possibly lacks finiteness may be called a "computational method." Besides his algorithm for the greatest common divisor of two integers, Euclid also gave a geometrical construction that is essentially equivalent to Algorithm E, except it is a procedure for obtaining the "greatest common measure" of the lengths of two line segments; this is a computational method that does not terminate if the given lengths are "incommensurate.")

2) Definiteness. Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case. The algorithms of this book will hopefully meet this criterion, but since they are specified in the English language, there is a possibility the reader might not understand exactly what the author intended. To get around this difficulty, formally defined *programming languages* or *computer languages* are designed for specifying algorithms, in which every statement has a very definite meaning. Many of the algorithms of this book will be given both in English and in a computer language. An expression of a computational method in a computer language is called a *program*.

In Algorithm E, the criterion of definiteness as applied to step E1 means that the reader is supposed to understand exactly what it means to divide m by n and what the remainder is. In actual fact, there is no universal agreement on what this means if m and n are not positive integers; what is the remainder of -8 divided by $-\pi$? What is the remainder of 59/13 divided by zero? Therefore the criterion of definiteness means we must make sure the values of m and n are always positive integers whenever step E1 is to be executed. This is initially true, by hypothesis, and after step E1 r is a nonnegative integer which must be nonzero if we get to step E3; so m and n are indeed positive integers as required.

3) Input. An algorithm has zero or more inputs, i.e., quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects. In Algorithm E, for example, there are two inputs, namely m and n, which are both taken from the set of *positive integers*.

4) Output. An algorithm has one or more outputs, i.e., quantities which have a specified relation to the inputs. Algorithm E has one output, namely n in step E2, which is the greatest common divisor of the two inputs.

(We can easily *prove* that this number is indeed the greatest common divisor, as follows. After step E1, we have

$$m = qn + r,$$

for some integer q. If $r = 0$, then m is a multiple of n, and clearly in such a case n is the greatest common divisor of m and n. If $r \neq 0$, note that any number which divides both m and n must divide $m - qn = r$, and any number which

divides both n and r must divide $qn + r = m$; so the set of divisors of m, n is the same as the set of divisors of n, r and, in particular, the *greatest* common divisor of m, n is the same as the greatest common divisor of n, r. Therefore step E3 does not change the answer to the original problem.)

5) Effectiveness. An algorithm is also generally expected to be *effective*. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using pencil and paper. Algorithm E uses only the operations of dividing one positive integer by another, testing if an integer is zero, and setting the value of one variable equal to the value of another. These operations are effective, because integers can be represented on paper in a finite manner and there is at least one method (the "division algorithm") for dividing one by another. But the same operations would *not* be effective if the values involved were arbitrary real numbers specified by an infinite decimal expansion, nor if the values were the lengths of physical line segments, which cannot be specified exactly. Another example of a noneffective step is, "If 2 is the largest integer n for which there is a solution to the equation $x^n + y^n = z^n$ in positive integers x, y, and z, then go to step E4." Such a statement would not be an effective operation until someone succeeds in showing that there is an algorithm to determine whether 2 is or is not the largest integer with the stated property.

Let us try to compare the concept of an algorithm with that of a cookbook recipe: A recipe presumably has the qualities of finiteness (although it is said that a watched pot never boils), input (eggs, flour, etc.) and output (TV dinner, etc.) but notoriously lacks definiteness. There are frequent cases in which the definiteness is missing, e.g., "Add a dash of salt." A "dash" is defined as "less than $\frac{1}{8}$ teaspoon"; salt is perhaps well enough defined; but where should the salt be added (on top, side, etc.)? Instructions like "toss lightly until mixture is crumbly," "warm cognac in small saucepan," etc., are quite adequate as explanations to a trained cook, perhaps, but an algorithm must be specified to such a degree that even a computer can follow the directions. Still, a computer programmer can learn much by studying a good recipe book. (In fact, the author has barely resisted the temptation to name the present volume "The Programmer's Cookbook." Perhaps someday he will attempt a book called "Algorithms for the Kitchen.")

We should remark that the "finiteness" restriction is really not strong enough for practical use; a useful algorithm should require not only a finite number of steps, but a *very* finite number, a reasonable number. For example, there is an algorithm which determines whether or not the game of chess is a forced victory for the White pieces (see exercise 2.2.3–28); here is an algorithm which can solve a problem of intense interest to thousands of people, yet it is a safe bet that we will never in our lifetimes know the answer to this problem, because the algorithm requires fantastically large amounts of time for its execution, even though it is "finite." See also Section 8.3 for a discussion of some finite numbers which are so large as to actually be beyond comprehension.

In practice we not only want algorithms, we want *good* algorithms in some loosely-defined aesthetic sense. One criterion of goodness is the length of time taken to perform the algorithm; this can be expressed in terms of the number of times each step is executed. Other criteria are the adaptability of the algorithm to computers, its simplicity and elegance, etc.

Occasionally, we will have several algorithms for the same problem, and we must decide which is best. This leads us to the extremely interesting and all-important field of *algorithmic analysis*: given an algorithm, the problem is to determine its performance characteristics.

For example, we can consider Euclid's algorithm from this point of view. Suppose we ask the question, "Assuming that the value of n is known but m is allowed to range over all positive integers, what is the *average* number of times, T_n, that step E1 of Algorithm E will be performed?" In the first place, we have to check that this question does have a meaningful answer (since we are trying to take an average over infinitely many choices for m). But it is evident that after the first execution of step E1 only the remainder of m after division by n is relevant. So all we must do to find the average, T_n, is to try the algorithm for $m = 1$, $m = 2, \ldots,$ $m = n$, count the total number of times step E1 has been executed, and divide by n.

Now the important question is to determine the *nature* of T_n; is it approximately equal to $\frac{1}{3}n$, or $\sqrt{n}$, etc.? (As a matter of fact, the answer to this question is an extremely difficult and fascinating mathematical problem, not yet resolved, which is examined in more detail in Section 4.5. For large values of n there is very good empirical evidence that T_n is approximately $(12 \ln 2/\pi^2) \ln n$, that is, proportional to the *natural logarithm* of n, with a constant of proportionality that might not have been guessed offhand!) For further details about Euclid's algorithm, and other ways to calculate the greatest common divisor, see Section 4.5.

"*Analysis of algorithms*" is the name the author likes to use to describe investigations such as this. The general idea is to take a particular algorithm and to determine its average behavior; occasionally we also study whether or not an algorithm is "optimal" in some sense. The *theory of algorithms* is another subject entirely, dealing primarily with the existence or nonexistence of effective algorithms to compute particular quantities; such theory is not investigated very deeply in this set of books, although it is considered briefly in Chapter 11.

So far our discussion of algorithms has been rather imprecise, and a mathematically oriented reader is justified in thinking that the preceding commentary makes a very shaky foundation on which to erect any theory about algorithms. We therefore close this section with a brief indication of one method by which the concept of algorithm can be firmly grounded in terms of mathematical set theory. Let us formally define a *computational method* to be a quadruple (Q, I, Ω, f), in which Q is a set containing subsets I and Ω, and f is a function from Q into itself. Furthermore f should leave Ω pointwise fixed; that is, $f(q)$ should equal q for all elements q of Ω. The four quantities Q, I, Ω, f are intended to represent respectively the states of the computation, the input, the output,

and the computational rule. Each input x in the set I defines a *computational sequence*, $x_0, x_1, x_2, \ldots$, *as* follows:

$$x_0 = x \qquad \text{and} \qquad x_{k+1} = f(x_k) \qquad \text{for} \qquad k \geq 0. \tag{1}$$

The computational sequence is said to *terminate* in k steps if k is the smallest integer for which x_k is in Ω, and in this case it is said to produce the output x_k from x. (Note that if x_k is in Ω, so is x_{k+1}, because $x_{k+1} = x_k$ in such a case.) Some computational sequences may never terminate; an *algorithm* is a computational method which terminates in finitely many steps for all x in I.

Algorithm E may, for example, be formalized in these terms as follows: Let Q be the set of all singletons (n), all ordered pairs (m, n), and all ordered quadruples $(m, n, r, 1)$, $(m, n, r, 2)$, and $(m, n, p, 3)$, where $m, n,$ and p are positive integers and r is a nonnegative integer. Let I be the subset of all pairs (m, n) and let Ω be the subset of all singletons (n). Let f be defined as follows:

$$\begin{aligned} f(m, n) &= (m, n, 0, 1); \qquad f(n) = (n); \\ f(m, n, r, 1) &= (m, n, \text{remainder of } m \div n, 2); \\ f(m, n, r, 2) &= (n) \quad \text{if} \quad r = 0, \quad (m, n, r, 3) \quad \text{otherwise}; \\ f(m, n, p, 3) &= (n, p, p, 1). \end{aligned} \tag{2}$$

The correspondence between this notation and Algorithm E is evident.

The above formulation of the concept "algorithm" does not include the restriction of "effectiveness" mentioned earlier; for example, Q might denote infinite sequences which are not computable by pencil and paper methods, or f might involve operations that mortal man cannot always perform. If we wish to restrict the notion of algorithm so that only elementary operations are involved, we can place restrictions on Q, I, Ω, and f, for example as follows: Let A be a finite set of letters, and let A^* be the set of all strings on A (i.e., the set of all ordered sequences $x_1x_2 \ldots x_n$, where $n \geq 0$ and x_j is in A for $1 \leq j \leq n$). The idea is to encode the states of the computation so that they are represented by strings of A^*. Now let N be a nonnegative integer and let Q be the set of all (σ, j) where σ is in A^* and j is an integer, $0 \leq j \leq N$; let I be the subset of Q with $j = 0$ and let Ω be the subset with $j = N$. If θ and σ are strings in A^*, we say that θ occurs in σ if σ has the form $\alpha\theta\omega$ for strings α and ω. To complete our definition, let f be a function of the following type, defined by the strings θ_j, ϕ_j and the integers a_j, b_j for $0 \leq j < N$:

$$\begin{aligned} f(\sigma, j) &= (\sigma, a_j) && \text{if } \theta_j \text{ does not occur in } \sigma; \\ f(\sigma, j) &= (\alpha\phi_j\omega, b_j) && \text{if } \alpha \text{ is the shortest possible string} \\ & && \quad \text{for which } \sigma = \alpha\theta_j\omega; \\ f(\sigma, N) &= (\sigma, N). \end{aligned} \tag{3}$$

Such a computational method is clearly "effective," and experience shows that it is also powerful enough to do anything we can do by hand. There are many

other essentially equivalent ways to formulate the concept of an effective computational method (for example, using Turing machines). The above formulation is virtually the same as that given by A. A. Markov in 1951, in his book *The Theory of Algorithms* (tr. from the Russian by J. J. Schorr-Kon, U.S. Dept. of Commerce, Office of Technical Services, number OTS 60-51085).

EXERCISES

1. [*10*] The text showed how to interchange the values of variables m and n, using the replacement notation, by setting $t \leftarrow m$, $m \leftarrow n$, $n \leftarrow t$. Show how the values (a, b, c, d) of *four* variables can be rearranged to (b, c, d, a) by a sequence of replacements. In other words, the new value of a is to be the original value of b, etc. Try to use the minimum number of replacements.

2. [*15*] Prove that m is always greater than n at the beginning of step E1, except possibly the first time this step occurs.

3. [*20*] Change Algorithm E (for the sake of efficiency) so that at step E3 we do not interchange values but immediately divide n by r and let m be the remainder. Add appropriate new steps so as to avoid all trivial replacement operations. Write this new algorithm in the style of Algorithm E, and call it Algorithm F.

4. [*16*] What is the greatest common divisor of 2166 and 6099?

▸ **5.** [*12*] Show that the "Procedure for Reading This Set of Books" which appears in the preface actually fails to be a genuine algorithm on three of our five counts! Also mention some differences in format between it and Algorithm E.

6. [*20*] What is T_5, according to the notation near the end of this section?

▸ **7.** [*M21*] Suppose that m is known and n is allowed to range over all positive integers; let U_m be the average number of times that step E1 is executed in Algorithm E. Show that U_m is well defined. Is U_m in any way related to T_m?

8. [*M25*] Give an "effective" formal algorithm for computing the greatest common divisor of positive integers m and n, by specifying θ_j, ϕ_j, a_j, b_j as in Eqs. (3). Let the input be represented by the string $a^m b^n$, that is, m a's followed by n b's. Try to make your solution as simple as possible. [*Hint:* Use Algorithm E, but instead of division in step E1, set $r \leftarrow |m - n|$, $n \leftarrow \min(m, n)$.]

▸ **9.** [*M30*] Suppose that $C_1 = (Q_1, I_1, \Omega_1, f_1)$ and $C_2 = (Q_2, I_2, \Omega_2, f_2)$ are computational methods. For example, C_1 might stand for Algorithm E as in Eqs. (2), except that m, n are restricted in magnitude, and C_2 might stand for a computer program implementation of Algorithm E. (Q_2 might be the set of all states of the machine, i.e., all possible configurations of its memory and registers; f_2 might be the definition of single machine actions; and I_2 might be the initial state including the program for determining the greatest common divisor, as well as the values of m and n.)

Formulate a set-theoretic definition for the concept "C_2 is a representation of C_1": This is to mean intuitively that any computation sequence of C_1 is mimicked by C_2, except that C_2 might take more steps in which to do the computation and it might retain more information in its states. (We thereby obtain a rigorous interpretation of the statement, "Program X is an implementation of Algorithm Y.")

1.2. MATHEMATICAL PRELIMINARIES

In this section we shall investigate the mathematical notations which are used throughout the rest of the chapters, and we shall also derive several basic formulas which are used repeatedly in this set of books. The reader who is not concerned with the more complex mathematical derivations should at least familiarize himself with the *meanings* of the various formulas, so that he can use the results of the derivations.

Mathematical notation is used for two main purposes in this set of books: (1) to describe portions of an algorithm; and (2) to analyze the performance characteristics of an algorithm. The notation used in descriptions of algorithms is quite simple, as explained in the previous section. When analyzing the performance of algorithms, we shall use other more specialized notations.

Most of the algorithms in this set of books are accompanied by mathematical calculations which determine the speed at which the algorithm may be expected to run. These calculations draw on nearly every branch of mathematics, and it would take a separate book to develop all of the mathematical concepts which are used in one place or another. However, the majority of the calculations can be carried out with a knowledge of college algebra, and the reader with a knowledge of elementary calculus will be able to understand nearly all of the mathematics which appears. In a few places we need to use deeper results of complex variable theory, group theory, number theory, probability theory, etc., and then either the topic is explained in an elementary manner, or a reference to other sources of information is given.

The mathematical techniques involved in the analysis of algorithms usually have a distinctive flavor; we will quite often find ourselves working with finite summations of rational numbers, or more frequently, with the solution to recurrence relations. Such topics are traditionally given only a light treatment in mathematics courses, and so the following subsections are designed to illustrate "in depth" the type of calculations and techniques used with such problems, as well as to give a thorough drilling in the use of the notations to be defined.

Important note. Although the following subsections provide a rather extensive training in the mathematical skills needed in connection with the study of computer algorithms, most readers will not see at first any very strong connections between this material and computer programming (except in Section 1.2.1). The reader may choose to read the following subsections carefully with implicit faith in the author's assertion that the topics treated here are indeed very relevant, or he may *skim over this section lightly at first* and then (after seeing numerous applications of these techniques in future chapters) he may wish to return to this section for more intensive study. The second alternative is probably preferable, since the reader will find himself better motivated; and if *too much* time is spent studying this material on first reading of the book, a person might find he never gets on to the computer programming topics! However, each reader should at least familiarize himself with the general contents

of these subsections, and should try his hand at a few of the exercises, even on first reading. Section 1.2.10 should receive particular attention, since it is the point of departure for most of the theoretical material developed later. Section 1.3 abruptly leaves the realm of "pure mathematics" and enters into "pure computer programming."

1.2.1. Mathematical Induction

Let $P(n)$ be some statement about the integer n; for example, $P(n)$ might be "n times $(n + 3)$ is an even number," or "if $n \geq 10$, then $2^n > n^3$." Suppose we want *to prove that $P(n)$ is true for all positive integers n.* An important way to do this is:

a) Give a proof that $P(1)$ is true;

b) Give a proof that "if all of $P(1), P(2), \ldots, P(n)$ are true, then $P(n + 1)$ is also true"; this proof should be valid for any positive integer n.

As an example, consider the following series of equations, which many people have discovered independently since ancient times:

$$1 = 1^2, \quad 1 + 3 = 2^2, \quad 1 + 3 + 5 = 3^2, \quad 1 + 3 + 5 + 7 = 4^2,$$
$$1 + 3 + 5 + 7 + 9 = 5^2. \tag{1}$$

We can formulate the general property as follows:

$$1 + 3 + \cdots + (2n - 1) = n^2. \tag{2}$$

Let us, for the moment, call this equation $P(n)$; we wish to prove that $P(n)$ is true for all positive n. Following the procedure outlined above, we have:

a) "$P(1)$ is true, since $1 = 1^2$."

b) "If all of $P(1), \ldots, P(n)$ are true, then, in particular, $P(n)$ is true, so Eq. (2) holds; adding $2n + 1$ to both sides we obtain

$$1 + 3 + \cdots + (2n - 1) + (2n + 1) = n^2 + 2n + 1 = (n + 1)^2$$

which proves that $P(n + 1)$ is also true."

We can regard this method as an *algorithmic proof procedure.* In fact, the following algorithm produces a proof of $P(n)$ for any positive integer n, assuming that steps (a) and (b) above have been worked out:

Algorithm I (*Construct a proof*). Given a positive integer n, this algorithm will output a proof that $P(n)$ is true.

I1. [Prove $P(1)$.] Set $k \leftarrow 1$, and, according to (a), output a proof of $P(1)$.

I2. [$k = n$?] If $k = n$, terminate the algorithm; the required proof has been output.

I3. [Prove $P(k+1)$.] According to (b), output a proof that "If all of $P(1), \ldots, P(k)$ are true, then $P(k+1)$ is true." Also output "We have already proved $P(1), \ldots, P(k)$; hence $P(k+1)$ is true."

I4. [Increase k.] Increase k by 1 and go to step I2. ∎

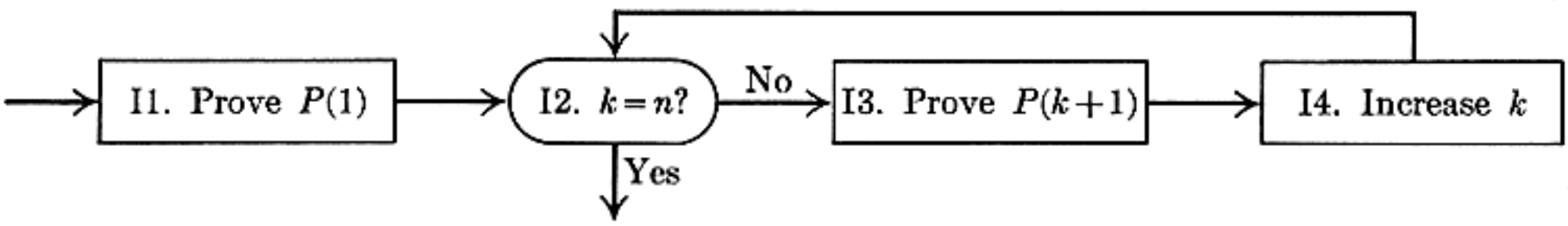


Fig. 2. Algorithm I: Mathematical induction.

Since this algorithm clearly presents a proof of $P(n)$, for any given n, we know that the above proof technique (a), (b) is logically valid. This method of proof is called a *proof by mathematical induction.*

The concept of "mathematical induction" should be distinguished from what is usually called "inductive reasoning" in science. A scientist takes specific observations and by "induction" he creates a general theory or hypothesis which accounts for these facts; for example, he might observe the five relations in (1), above, and formulate (2). In this sense, "induction" is no more than somebody's best guess about the situation; in mathematics we would call this an empirical result or a conjecture.

Another example will be helpful. Let $p(n)$ denote the numbers of "partitions of n," that is, the number of different ways to write n as a sum of positive integers, disregarding order. Since

$$\begin{aligned} 5 &= 1+1+1+1+1 = 2+1+1+1 = 2+2+1 \\ &= 3+1+1 = 3+2 = 4+1 = 5, \end{aligned}$$

we have $p(5) = 7$. In fact, it is easy to establish the first few values,

$$p(1) = 1, \quad p(2) = 2, \quad p(3) = 3, \quad p(4) = 5, \quad p(5) = 7.$$

At this point we might tentatively formulate, by "induction," the hypothesis that the sequence $p(n)$ runs through the *prime numbers.* To test this hypothesis, we proceed to calculate $p(6)$ and behold! $p(6) = 11$, confirming our conjecture.

[Unfortunately, $p(7)$ turns out to be 15, spoiling everything, and we must try again. This problem is known to be quite difficult, although S. Ramanujan succeeded in guessing and proving many remarkable things about the numbers $p(n)$; for further information, see G. H. Hardy, *Ramanujan* (London: Cambridge University Press, 1940), chapters 6 and 8.]

On the other hand, "mathematical induction" is quite different from plain "induction." It is not just guesswork, it is a conclusive proof of a statement (indeed, here it is a proof of infinitely many statements, one for each n). It has

been called "induction" only because one must first decide somehow *what* he is going to prove, *before* he can apply the technique of mathematical induction. Henceforth in this book we shall use the word induction only when we wish to imply proof by mathematical induction.

There is a geometrical way to prove Eq. (2). Figure 3 shows, for $n = 6$, n^2 cells broken into groups of $1 + 3 + \cdots + (2n - 1)$ cells. However, in the final analysis, this picture can be regarded as a "proof" only if we show that the construction can be carried out for all n, and this is essentially the same as a proof by induction.

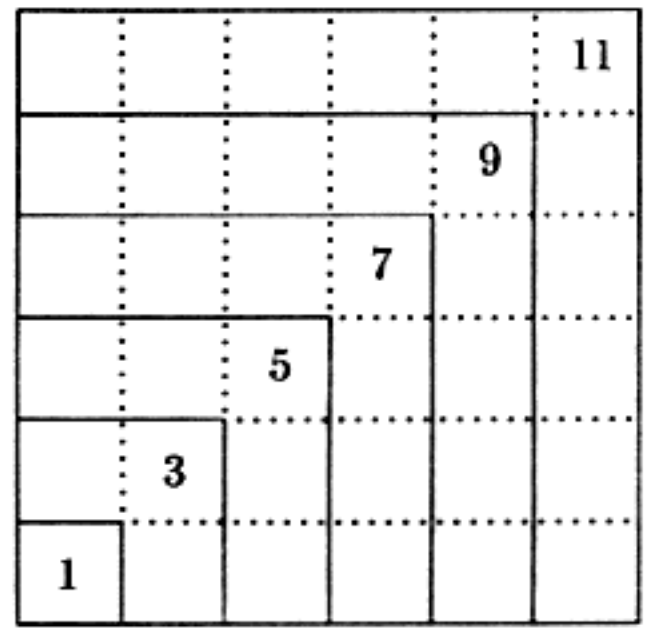


Fig. 3. The sum of odd numbers is a square.

Our proof of Eq. (2) above used only a special case of (b); we merely showed that the truth of $P(n)$ implies the truth of $P(n+1)$. This is an important simple case which arises frequently, but our next example illustrates the power of the method a little more. We define the *Fibonacci sequence* $F_0, F_1, F_2, \ldots$ by the rule that $F_0 = 0$, $F_1 = 1$, and every further term is the sum of the preceding two. Thus the sequence begins $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ This sequence is investigated in detail in Section 1.2.8. We will now prove that if ϕ is the number $(1 + \sqrt{5})/2$, we have

$$F_n \le \phi^{n-1} \tag{3}$$

for all positive integers n.

If $n = 1$, then $F_1 = 1 = \phi^0 = \phi^{n-1}$, so step (a) has been done. We must now do step (b). $P(2)$ is also true, since $F_2 = 1 < 1.6 < \phi^1 = \phi^{2-1}$. Now, if $P(1), P(2), \ldots, P(n)$ are true and $n > 1$, we have, in particular, that $P(n-1)$ and $P(n)$ are true, so $F_{n-1} \le \phi^{n-2}$ and $F_n \le \phi^{n-1}$. Adding these inequalities, we get

$$F_{n+1} = F_{n-1} + F_n \le \phi^{n-2} + \phi^{n-1} = \phi^{n-2}(1 + \phi). \tag{4}$$

The important property of the number ϕ, indeed the reason we chose this number for this problem in the first place, is that

$$\phi^2 = \phi + 1. \tag{5}$$

Putting this into (4) gives $F_{n+1} \le \phi^n$, which is $P(n+1)$. So step (b) has been done, and (3) has been proved by mathematical induction. Note that we approached step (b) in two different ways here: we proved $P(n+1)$ *directly* when $n = 1$, and we used an inductive method when $n > 1$. This was necessary, since when $n = 1$ our reference to $P(n-1) = P(0)$ would not have been legitimate.

We will now see how mathematical induction can be used to prove things about *algorithms*. Consider the following generalization of Euclid's algorithm.

Algorithm E (*Extended Euclid's algorithm*). Given two positive integers m and n, we compute their greatest common divisor d and two integers a and b, such that $am + bn = d$.

E1. [Initialize.] Set $a' \leftarrow b \leftarrow 1$, $a \leftarrow b' \leftarrow 0$, $c \leftarrow m$, $d \leftarrow n$.

E2. [Divide.] Let q, r be the quotient and remainder, respectively, of c divided by d. (We have $c = qd + r$, $0 \le r < d$.)

E3. [Remainder zero?] If $r = 0$, the algorithm terminates; we have in this case $am + bn = d$ as desired.

E4. [Recycle.] Set $c \leftarrow d$, $d \leftarrow r$, $t \leftarrow a'$, $a' \leftarrow a$, $a \leftarrow t - qa$, $t \leftarrow b'$, $b' \leftarrow b$, $b \leftarrow t - qb$, and go back to E2. ∎

If we suppress the variables a, b, a', and b' from this algorithm and use m, n for the auxiliary variables c, d, we have our old algorithm, 1.1E. The new version does a little more, by determining the coefficients a, b. Suppose that $m = 1769$ and $n = 551$; we have successively (after step E2):

a'	a	b'	b	c	d	q	r
1	0	0	1	1769	551	3	116
0	1	1	−3	551	116	4	87
1	−4	−3	13	116	87	1	29
−4	5	13	−16	87	29	3	0.

The answer is correct: $5 \times 1769 - 16 \times 551 = 8845 - 8816 = 29$, the greatest common divisor of 1769 and 551.

The problem is to *prove* that this algorithm works properly for all m and n. We can try to set this up for the method of mathematical induction by letting $P(n)$ be the statement "Algorithm E works for n and all integers m." However, this doesn't work out so easily, and we need to prove some extra facts. After a little study, we find that something must be proved about a, b, a', and b', and the appropriate fact is that

$$a'm + b'n = c, \qquad am + bn = d \tag{6}$$

always holds whenever step E2 is executed. We may prove Eqs. (6) directly by observing that it is certainly true the first time we get to E2, and step E4 does not change the validity of (6). (See exercise 1.2.1–6.)

Now we are ready to show that Algorithm E is valid, by induction on n: If m is a multiple of n, the algorithm obviously works properly, since we are done immediately at E3 the first time. This case always occurs when $n = 1$. The only case remaining is when $n > 1$ and m is not a multiple of n. In this case, the algorithm proceeds to set $c \leftarrow n$, $d \leftarrow r$ after the first execution, and since

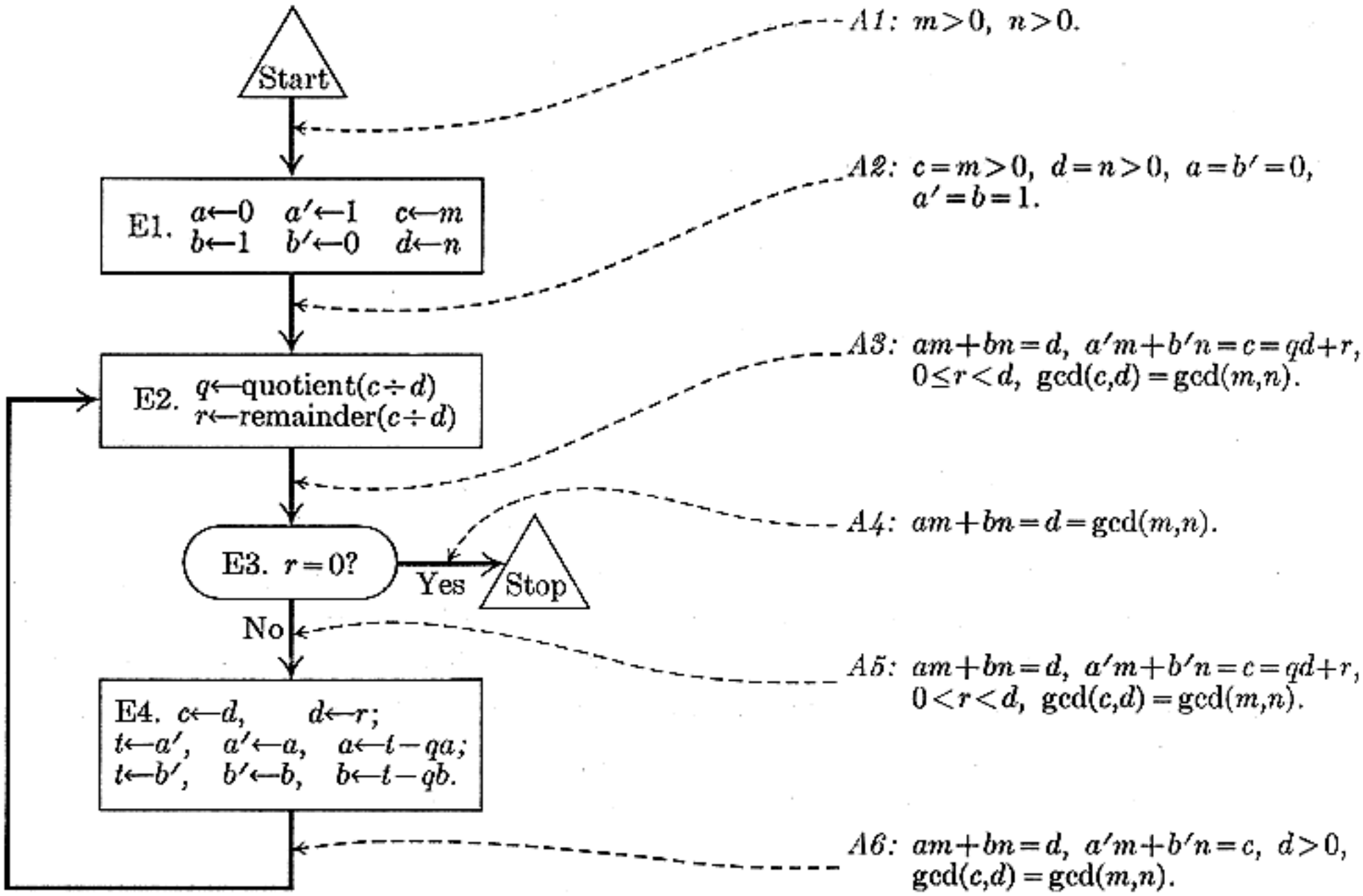


Fig. 4. Flow chart for Algorithm E, labeled with assertions which prove the validity of the algorithm.

$r < n$, *we may assume by induction* that the final value of d is the g.c.d. of n and r. By the argument given in Section 1.1, the pairs m, n and n, r have the same common divisors, and, in particular, they have the same greatest common divisor. Hence d is the g.c.d. of m and n, and by Eq. (6), $am + bn = d$.

The italicized phrase in the above proof illustrates the conventional language which is so often used in an inductive proof: when doing part (b) of the construction, rather than saying "We will now assume $P(1), P(2), \ldots, P(n)$, and with this assumption we will prove $P(n + 1)$," we often say simply "We will now prove $P(n)$; we may assume *by induction* $P(k)$ is true whenever $1 \leq k < n$."

If we examine the above argument very closely and change our viewpoint slightly, we can see *a general method applicable to proving the validity of any algorithm.* The idea is to take a flow chart for some algorithm and to label each of the arrows with an assertion about the current state of affairs at the time the computation traverses that arrow. See Fig. 4, where the assertions have been labeled *A1, A2, . . . , A6.* (All of these assertions have the additional stipulation that the variables are integers; this stipulation has been omitted to save space.) *A1* gives the initial assumptions upon entry to the algorithm, and *A4* states what we hope to prove about the output values a, b, and d.

The general method consists of proving, for each box in the flow chart, that *if any one of the assertions on the arrows leading into the box is true before the operation in that box is performed, then all of the assertions on the arrows leading*

away from the box are true after the operation. Thus, for example, we must prove that either *A2* or *A6* before E2 implies *A3* after E2. (In this case *A2* is a stronger statement than *A6*, that is, *A2* implies *A6*, so we need only prove *A6* before E2 implies *A3* after. Note that the condition $d > 0$ is necessary in *A6* just to prove that the operation E2 even makes sense.) It is also necessary to show that *A3* and $r = 0$ implies *A4*; *A3* and $r \neq 0$ implies *A5*; etc. Each of the required proofs is very straightforward.

Once the italicized statement above has been proved for each box, it follows that all assertions are true during any execution of the algorithm. For we can now use induction on the number of steps of the computation, in the sense of the number of arrows traversed in the flow chart. While traversing the first arrow, i.e., the arrow leading from "Start", the assertion *A1* is true since we always assume our input values meet the specifications; so the assertion on the first arrow traversed is correct. If the assertion that labels the nth arrow is true, then by the italicized statement the assertion that labels the $(n + 1)$st arrow is also true.

Using this general method, the problem of proving that a given algorithm is valid evidently consists mostly of inventing the right assertions to put in the flow chart. Once this "inductive leap" has been made, it is pretty much routine to carry out the proofs that each assertion leading into a box implies each assertion leading out. In fact, it is pretty much routine to invent the assertions themselves, once a few of the difficult ones have been discovered; thus it is very simple in our example to write out essentially what *A2*, *A3*, *A4*, and *A5* must be, if only *A1* and *A6* are given. In our example, the "creative" part of the proof is assertion *A6*, and all the rest could, in principle, be supplied mechanically. Hence no attempt has been made to give detailed formal proofs of most of the algorithms which follow in this book; it suffices to state the key inductive assertions, and these either appear in the discussion following the algorithm or they are given as parenthetical remarks in the text of the algorithm itself.

The above principle for proving algorithms has another aspect which is perhaps even more important: *it mirrors the way we "understand" an algorithm.* Recall that in Section 1.1 the reader was cautioned not to expect to read an algorithm like a novel; one or two trials of the algorithm on some sample data are recommended. This is done expressly because an example performance of the algorithm helps a person to formulate the various assertions in his own mind. It is the contention of the author that we really understand why an algorithm is valid only when we reach the point that our minds have implicitly filled in all the assertions, as was done in Fig. 4. This point of view has important psychological consequences for the proper communication of algorithms from one man to another (or from one man to himself, when he looks over his own algorithms several months later): it implies that the key assertions, those that cannot easily be derived by an automaton, should always be stated explicitly when an algorithm is being explained to someone else. When Algorithm E is being put forward, assertion *A6* should be mentioned too.

An alert reader will have noticed a gaping hole in our last proof of Algorithm E, however. We never showed that the algorithm terminates; all we have proved is that *if* it terminates, it gives the right answer!

(Note, for example, that Algorithm E still makes sense if we allow its variables m, n, c, and r to assume values of the form $u + v\sqrt{2}$, where u and v are integers. The variables q, a, b, a', b' are to remain integer-valued. If we start the algorithm with $m = 6 - 12\sqrt{2}$ and $n = 10 - 20\sqrt{2}$, say, it will compute a "greatest common divisor" $d = 2 - 4\sqrt{2}$, with $a = +2$, $b = -1$. Even under this extension of the assumptions, the proofs of assertions *A1* through *A6* remain valid; therefore all assertions are true throughout any execution of the algorithm. But if we start the procedure with $m = 1$ and $n = \sqrt{2}$, the computation never terminates (see exercise 12). Hence a proof of assertions *A1* through *A6* does *not* logically prove the algorithm is finite.)

Therefore proofs of termination are usually handled separately. It *is* possible to extend the above method in many important cases so that a proof of terminations is included as a by-product, as shown in exercise 13.

We have now twice proved the validity of Algorithm E. To be strictly logical, we should also try to prove that the first algorithm in this section, Algorithm I, is valid; in fact, we have used Algorithm I to establish the correctness of any proof by induction. If we attempt to *prove* that Algorithm I works properly, however, we are confronted with a dilemma—we can't really prove it without using induction again! The argument would be circular.

In the last analysis, *every* property of the integers must be proved using induction somewhere along the line, because if we get down to basic concepts, the integers are essentially *defined* by induction. Therefore we may take as axiomatic the idea that any positive integer n either equals 1 or can be reached by starting with **1** and repetitively "adding" 1; this suffices to prove that Algorithm I is valid. [For a rigorous study of fundamental concepts about the integers, see the article "On Mathematical Induction," by Leon Henkin, *AMM* **67** (1960), 323–338.]

The idea behind mathematical induction is thus intimately related to the concept of number. The first person to apply mathematical induction to rigorous proofs was the Italian scientist Francesco Maurolico, in 1575, and further improvements were made by Pierre de Fermat, in the early 17th century. Fermat called it the "method of infinite descent." The notion also appears clearly in the later writings of Blaise Pascal (1653). The phrase "mathematical induction" apparently was coined by A. de Morgan in the early nineteenth century. [See *AMM* **24** (1917), 199–207; **25** (1918), 197–201.] For futher discussion of mathematical induction, see G. Polya, *Induction and Analogy in Mathematics* (Princeton, N.J.: Princeton University Press, 1954), Chapter 7.

The formulation of algorithm-proving in terms of assertions and induction, as given above, is essentially due to R. W. Floyd. He points out that a semantic definition of each operation in a programming language is most properly given

as a logical rule which tells exactly what assertions can be proved after the operation, from what assertions that are true beforehand [see "Assigning Meanings to Programs," *Proc. Symp. Appl. Math.*, AMS **19** (1967), 19–32]. Similar ideas have been voiced independently by Peter Naur, *BIT* **6** (1966), 310–316, who calls the assertions "general snapshots." The idea of inductive assertions actually appeared in embryonic form in 1946, at the same time the concept of flow charts was introduced by H. H. Goldstine and J. von Neumann. These original flow charts included "assertion boxes" which are in close analogy with the assertions in Fig. 4. [See John von Neumann, *Collected Works*, **5** (New York: Macmillan, 1963), 91–99.]

Another method of proving the correctness of algorithms by induction is discussed in Section 2.3.1.

EXERCISES

1. [*05*] Explain how to modify the idea of proof by mathematical induction, in case we want to prove some statement $P(n)$ for all *nonnegative* integers, i.e., for $n = 0, 1, 2, \ldots$ instead of for $n = 1, 2, 3, \ldots$.

▶ **2.** [*15*] There must be something wrong with the following proof; what is it? "*Theorem:* Let a be any positive number. For all positive integers n we have $a^{n-1} = 1$. *Proof:* If $n = 1$, $a^{n-1} = a^{1-1} = a^0 = 1$. And by induction, assuming that the theorem is true for $1, 2, \ldots, n$, we have

$$a^{(n+1)-1} = a^n = \frac{a^{n-1} \times a^{n-1}}{a^{n-2}} = \frac{1 \times 1}{1} = 1;$$

so the theorem is true for $n + 1$ as well."

3. [*18*] The following proof by induction seems correct, but for some reason the equation for $n = 6$ gives $\frac{1}{2} + \frac{1}{6} + \frac{1}{12} + \frac{1}{20} + \frac{1}{30} = \frac{5}{6}$ on the left-hand side, and $\frac{3}{2} - \frac{1}{6} = \frac{4}{3}$ on the right-hand side. Can you find a mistake? "*Theorem:*

$$\frac{1}{1 \times 2} + \frac{1}{2 \times 3} + \cdots + \frac{1}{(n-1) \times n} = \frac{3}{2} - \frac{1}{n}.$$

Proof: We use induction on n. For $n = 1$, $\frac{3}{2} - 1/n = \frac{1}{2}$; and, assuming the theorem is true for n,

$$\frac{1}{1 \times 2} + \cdots + \frac{1}{(n-1) \times n} + \frac{1}{n \times (n+1)}$$
$$= \frac{3}{2} - \frac{1}{n} + \frac{1}{n(n+1)} = \frac{3}{2} - \frac{1}{n} + \left(\frac{1}{n} - \frac{1}{n+1}\right) = \frac{3}{2} - \frac{1}{n+1}.$$ "

4. [*20*] Prove that, in addition to Eq. (3), $F_n \geq \phi^{n-2}$.

5. [*25*] A *prime number* is an integer greater than one which has no exact divisors other than 1 and itself. Using this definition and mathematical induction, prove that every positive integer greater than one may be written as a product of prime numbers.

6. [*20*] Prove that if Eq. (6) holds before step E4 is performed, it holds afterwards also.

7. [*23*] Formulate and prove by induction a rule for the sums 1^2, $2^2 - 1^2$, $3^2 - 2^2 + 1^2$, $4^2 - 3^2 + 2^2 - 1^2$, $5^2 - 4^2 + 3^2 - 2^2 + 1^2$, etc.

▸ **8.** [*25*] (a) Prove the following theorem of Nicomachus (c. 100 A.D.) by induction: $1^3 = 1$, $2^3 = 3 + 5$, $3^3 = 7 + 9 + 11$, $4^3 = 13 + 15 + 17 + 19$, etc. (b) Use this result to prove the remarkable formula

$$1^3 + 2^3 + \cdots + n^3 = (1 + 2 + \cdots + n)^2.$$

[*Note:* An attractive, geometric interpretation of this formula, suggested to the author by R. W. Floyd, is shown in Fig. 5. The idea is related to Nicomachus's theorem and Fig. 3.]

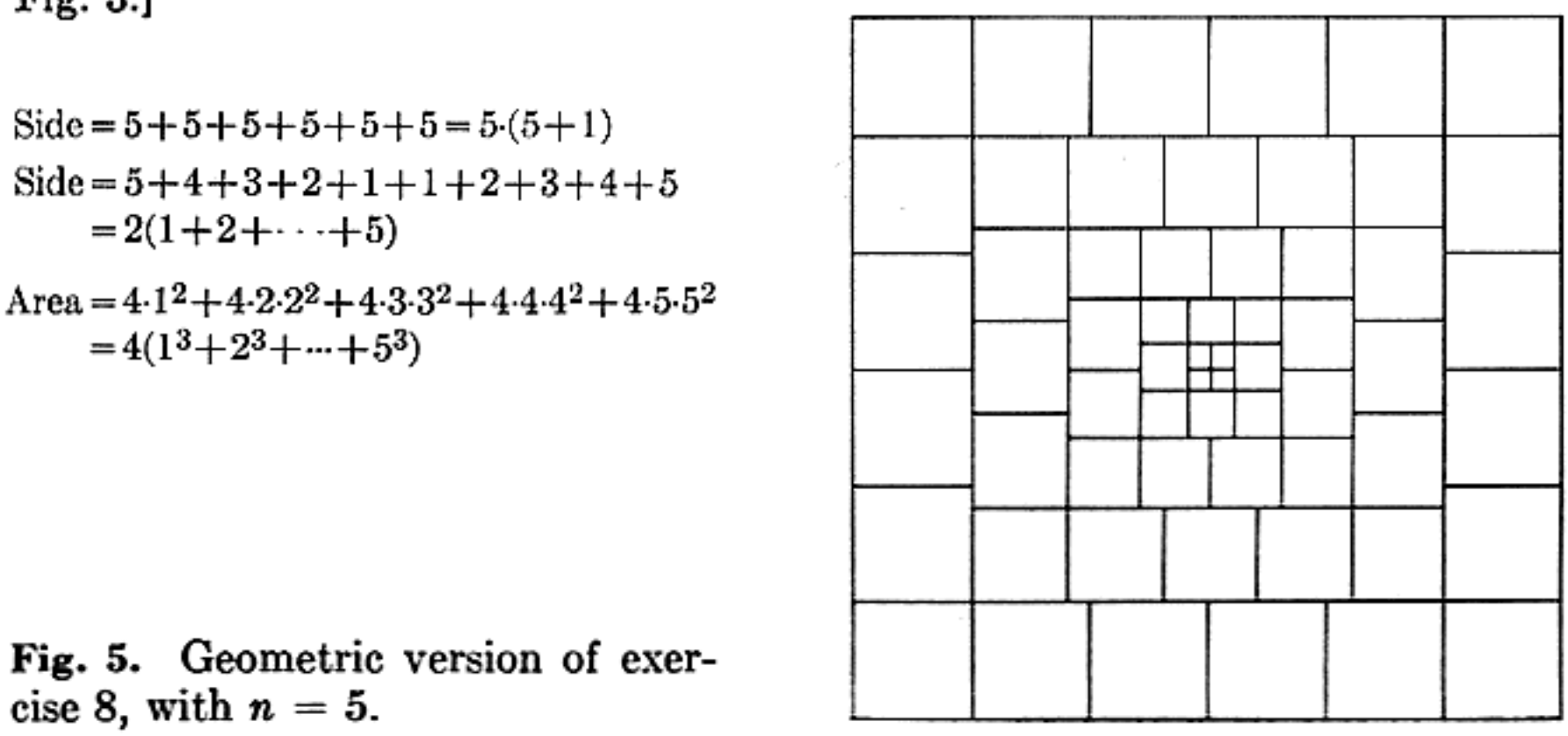


Fig. 5. Geometric version of exercise 8, with $n = 5$.

9. [*20*] Prove by induction that if $0 < a < 1$, then $(1 - a)^n \geq 1 - na$.

10. [*M25*] Prove by induction that if $n \geq 10$, then $2^n > n^3$.

11. [*M30*] Find and prove a simple formula for the sum

$$\frac{1^3}{1^4 + 4} - \frac{3^3}{3^4 + 4} + \frac{5^3}{5^4 + 4} - \cdots + \frac{(-1)^n(2n + 1)^3}{(2n + 1)^4 + 4}.$$

12. [*M25*] Show how Algorithm E can be generalized as stated in the text so that it will accept input values of the form $u + v\sqrt{2}$, where u and v are integers, and the computations can still be done in an elementary way (i.e., without using the infinite decimal expansion of $\sqrt{2}$). Prove that the computation will not terminate, however, if $m = 1$ and $n = \sqrt{2}$.

▸ **13.** [*M23*] Extend Algorithm E by adding a new variable T and adding the operation "$T \leftarrow T + 1$" at the beginning of each step. (T is like a time clock, counting the number of steps executed.) Assume that T is initially zero, so that assertion *A1* in Fig. 4 becomes "$m > 0$, $n > 0$, $T = 0$." The additional condition "$T = 1$" should similarly be appended to *A2*. Show how to append additional conditions to the assertions in such a way that any one of *A1*, *A2*, . . . , *A6* implies $T \leq 3n_0$, where n_0 is the original value of n, and such that the inductive proof can still be carried out. (Hence the computation must terminate in at most $3n_0$ steps.)

14. [*50*] (R. W. Floyd.) Prepare a computer program which accepts, as input, programs in some programming language together with optional assertions, and which attempts to fill in the remaining assertions necessary to make a proof that the computer program is valid. (For example, strive to get a program that is able to prove the validity of Algorithm E, given only assertions *A1* and *A6*. Note that the existence of such a program makes debugging unnecessary, except for proofs of termination!)

▸ **15.** [*HM28*] (*Generalized induction.*) The text shows how to prove statements $P(n)$ which depend on a single integer n, but it does not describe how to prove statements $P(m, n)$ depending on two integers. In these circumstances a proof is often given by some sort of "double induction," which frequently seems confusing. Actually, there is an important principle more general than simple induction which applies not only to this case but also to situations in which statements are to be proved about uncountable sets, for example, $P(x)$ for all real x. This general principle is called *well-ordering*.

Let "$\prec$" be a relation on a set S, satisfying the following properties:

i) Given x, y, z in S, if $x \prec y$ and $y \prec z$, then $x \prec z$.

ii) Given x, y in S, exactly one of the following three possibilities is true: $x \prec y$, $x = y$, or $y \prec x$.

iii) If A is any nonempty subset of S, there is an element x in A with $x \preceq y$ for all y in A.

This relation is said to be a well-ordering of S. For example, it is clear that the positive integers are well-ordered by the ordinary "less than" relation, $<$.

a) Show that the set of *all* integers is not well-ordered by $<$.

b) Define a well-ordering relation on the set of all integers.

c) Is the set of all nonnegative real numbers well-ordered by $<$?

d) (*Lexicographic order.*) Let S be well-ordered by $\prec$, and for $n > 0$ let T_n be the set of all n-tuples $(x_1, x_2, \ldots, x_n)$ of elements x_j in S. Define $(x_1, x_2, \ldots, x_n) \prec (y_1, y_2, \ldots, y_n)$, if there is some k, $1 \leq k \leq n$, such that $x_j = y_j$ for $1 \leq j < k$, but $x_k \prec y_k$ in S. Is $\prec$ a well-ordering of T_n?

e) As in part (d), let $T = \bigcup_{n \geq 1} T_n$; define $(x_1, x_2, \ldots, x_n) \prec (y_1, y_2, \ldots, y_m)$ if $x_j = y_j$ for $1 \leq j < k$ and $x_k \prec y_k$, for some $k \leq m, n$; or if $x_j = y_j$ for $1 \leq j \leq n$ and $n < m$. Is $\prec$ a well-ordering of T?

f) Show that $\prec$ is a well-ordering of S if and only if it satisfies (i) and (ii) above and there is no infinite sequence $x_1, x_2, x_3, \ldots$ with $x_{j+1} \prec x_j$ for all $j \geq 1$.

g) Let S be well-ordered by $\prec$, and let $P(x)$ be a statement about the element x of S. Show that if $P(x)$ can be proved under the assumption that $P(y)$ is true for all $y \prec x$, then $P(x)$ is true for *all* x in S.

[*Notes:* Part (g) is the generalization of simple induction that was promised; in the case S = positive integers, it is just the simple case of mathematical induction treated in the text. Note that we are asked to prove that $P(1)$ is true if $P(y)$ is true for all positive integers $y < 1$; this is the same as saying we should prove $P(1)$, since $P(y)$ certainly is (vacuously) true for all such y. Consequently, one finds that in many situations $P(1)$ need not be proved using a special argument.

Part (d), in connection with part (g), gives us in particular the rather powerful method of n-tuple induction for proving statements $P(m_1, m_2, \ldots, m_n)$ about n positive integers $m_1, m_2, \ldots, m_n$.

Part (f) has further application to computer algorithms: if we can map the states of a computation into a well-ordered set S in such a way that every step of the computa-

tion takes a state x into a state y with $f(y) \prec f(x)$, then the algorithm must terminate. This principle generalizes the argument about the strictly decreasing values of n that was used to prove that Algorithm 1.1E terminates.]

1.2.2. Numbers, Powers, and Logarithms

Let us now begin our study of numerical mathematics by taking a good look at the numbers we are dealing with. The *integers* are the whole numbers

$$\ldots,\ -3,\ -2,\ -1,\ 0,\ 1,\ 2,\ 3,\ \ldots$$

(positive, negative, or zero). A *rational number* is the ratio (quotient) of two integers, p/q, where q is positive. A *real number* is a quantity x which has a "decimal expansion":

$$x = n + 0.d_1d_2d_3\ldots, \tag{1}$$

where n is an integer, each d_i is a digit between 0 and 9, and no infinite sequence of 9's appears. The representation (1) means that

$$n + \frac{d_1}{10} + \frac{d_2}{100} + \cdots + \frac{d_k}{10^k} \le x < n + \frac{d_1}{10} + \frac{d_2}{100} + \cdots + \frac{d_k}{10^k} + \frac{1}{10^k}, \tag{2}$$

for all positive integers k. Two examples of real numbers that are not rational are

$\pi = 3.14159265358979\ldots$, the ratio of the circumference of a circle to its diameter;

$\phi = 1.61803398874989\ldots$, the "golden ratio" $(1 + \sqrt{5})/2$ (see Section 1.2.8).

A table of important constants, to forty decimal places of accuracy, appears in Appendix B. We will not discuss the familiar properties of addition, subtraction, multiplication, division, and comparison of real numbers.

Throughout this section, let the letter b stand for a positive real number. If n is an integer, then b^n is defined by the familiar rules:

$$b^0 = 1, \qquad b^n = b^{n-1}b \quad \text{if} \quad n > 0, \qquad b^n = b^{n+1}/b \quad \text{if} \quad n < 0. \tag{3}$$

It is easy to prove by induction that the *laws of exponents* are valid:

$$b^{x+y} = b^x b^y, \qquad (b^x)^y = b^{xy}, \tag{4}$$

whenever x and y are integers.

If u is a positive real number and if m is a positive integer, there is always a unique positive real number v which is its "mth root," that is, $v^m = u$. We write $v = \sqrt[m]{u}$.

We now define b^r for rational numbers r as follows:

$$b^{p/q} = \sqrt[q]{b^p}. \tag{5}$$

This definition, due to Oresme (c. 1360), is a good one, since $b^{ap/aq} = b^{p/q}$, and since the laws of exponents are still correct even when x and y are arbitrary rational numbers (see exercise 9).

Finally, we define b^x for all real values of x. Suppose first that $b > 1$; if x is given by Eq. (1), we want

$$b^{n+d_1/10+\cdots+d_k/10^k} \le b^x < b^{n+d_1/10+\cdots+d_k/10^k+1/10^k}. \tag{6}$$

This *defines* b^x as a unique positive real number, since the difference between the left and right extremes in Eq. (6) is $b^{n+d_1/10+\cdots+d_k/10^k}(b^{1/10^k} - 1)$; by exercise 13 below, this difference is less than $b^{n+1}(b-1)/10^k$, and if we take k large enough, we can therefore get any desired accuracy for b^x.

For example, we find that

$$10^{0.30102999} = 1.9999999737\ldots, \qquad 10^{0.30103000} = 2.0000000198\ldots, \tag{7}$$

and therefore if $b = 10$, $x = 0.30102999\ldots$, we know the value of 10^x with an accuracy of better than one part in 10 million (although we still don't even know whether the decimal expansion of 10^x is $1.999\ldots$ or $2.000\ldots$!).

When $b < 1$, we define $b^x = (1/b)^{-x}$; and when $b = 1$, $1^x = 1$. With these definitions, it can be proved that the laws of exponents (Eqs. 4) hold for any *real* values of x and y. These ideas for defining b^x were first formulated by John Wallis (1655) and Isaac Newton (1669).

Now we come to an important question. Suppose that a positive real number y is given; can we find a real number x such that $y = b^x$? The answer is "yes" (provided that $b \neq 1$), for we simply use Eq. (6) in *reverse* to determine n and $d_1, d_2, \ldots$ when $b^x = y$ is given. The resulting number x is called the *logarithm of y to the base b*, and we write this as $x = \log_b y$. By this definition we have

$$x = b^{\log_b x} = \log_b (b^x). \tag{8}$$

As an example, Eqs. (7) show that

$$\log_{10} 2 = 0.30102999\ldots \tag{9}$$

From the laws of exponents it follows that

$$\log_b (xy) = \log_b x + \log_b y, \qquad \text{if} \qquad x > 0, \quad y > 0 \tag{10}$$

and

$$\log_b (c^y) = y \log_b c, \qquad \text{if} \qquad c > 0. \tag{11}$$

Equation (9) illustrates the so-called "common logarithms," i.e., logarithms to the base 10. One might expect that in computer work *binary logarithms* (to the base 2) might be more useful, since binary arithmetic is often used in computers. Actually, we will see that binary logarithms *are* very useful, but not really for that reason; the reason is primarily that a computer algorithm often makes two-way branches.

The question now arises as to whether or not there is any relationship between $\log_2 x$ and $\log_{10} x$; fortunately there is one, because according to Eqs. (8) and (11),

$$\log_{10} x = \log_{10} (2^{\log_2 x}) = (\log_2 x)(\log_{10} 2).$$

Hence

$$\log_2 x = \log_{10} x / \log_{10} 2,$$

and in general we find that

$$\log_c x = \log_b x / \log_b c. \tag{12}$$

Equations (10), (11), and (12) are the fundamental rules for manipulating logarithms.

It turns out that neither base 10 nor base 2 is really the most convenient base to work with in most cases. There is a real number, denoted by $e = 2.718281828459045\ldots$, for which the logarithms have simpler properties. By convention, we call logarithms to the base e "natural logarithms," and we write

$$\ln x \equiv \log_e x. \tag{13}$$

This rather arbitrary definition (in fact, we haven't really defined e) probably doesn't strike the reader as being a very "natural" logarithm; yet we will find that $\ln x$ will seem more and more natural, the more we work with it. John Napier actually discovered natural logarithms (with slight modifications, and without connecting them with powers) before the year 1590, many years before any other kind of logarithm was known. We can give two brief examples, without proof, of why these logarithms might seem most "natural": (a) In Fig. 6 the area of the shaded portion is $\ln x$. (b) If a bank pays compound interest at rate r, compounded semiannually, the return on each dollar is $(1 + r/2)^2$ dollars; if it is compounded quarterly, you get $(1 + r/4)^4$ dollars; and if it is compounded daily you probably get $(1 + r/365)^{365}$ dollars. Now if the interest were compounded *continuously*, you would get exactly e^r dollars for every dollar (ignoring roundoff error)! In this age of computers, some bankers may soon approach or reach this limiting formula.

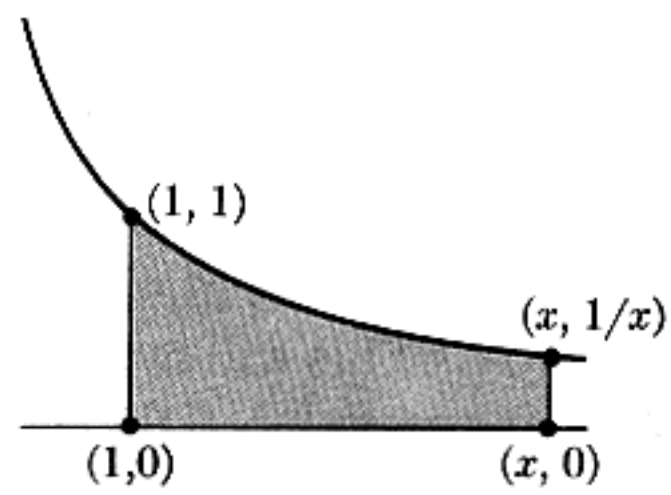


Fig. 6. Natural logarithm.

For the interesting history of the concepts of logarithm and exponential, see the series of articles by F. Cajori, *AMM* **20** (1913), 5–14, 35–47, 75–84, 107–117, 148–151, 173–182, 205–210.

We conclude this section by considering how to *compute* logarithms. One method is suggested immediately by Eq. (6): if we let $b^x = y$ and raise all parts of that equation to the 10^k-th power, we find that

$$b^m \leq y^{10^k} < b^{m+1}, \tag{14}$$

for some integer m. All we have to do to get the logarithm of y is to raise y to this huge power and find which powers $(m, m+1)$ of b this result lies between, and $m/10^k$ is the answer to k decimal places.

A slight modification of this apparently impractical method leads to a simple and reasonable procedure. We will show how to calculate $\log_{10} x$ and to express the answer in the *binary* system, as

$$\log_{10} x = n + b_1/2 + b_2/4 + b_3/8 + \cdots .$$

First we shift the decimal point of x to the left or to the right so that we have $1 \le x/10^n < 10$; this determines n for us. To obtain $b_1, b_2, b_3, \ldots$ we now set $x_0 = x/10^n$ and, for $k \ge 1$,

$$\begin{aligned} &b_k = 0, \quad x_k = x_{k-1}^2, \quad \text{if} \quad x_{k-1}^2 < 10; \\ &b_k = 1, \quad x_k = x_{k-1}^2/10, \quad \text{if} \quad x_{k-1}^2 \ge 10. \end{aligned} \tag{15}$$

The validity of this procedure follows from the fact that

$$1 \le x_k = x^{2^k}/10^{2^k(n+b_1/2+\cdots+b_k/2^k)} < 10, \tag{16}$$

for $k = 0, 1, 2, \ldots$, as is easily proved by induction.

In practice, of course, we must work with only finite accuracy, so we cannot set $x_k = x_{k-1}^2$ exactly. Instead, we set $x_k = x_{k-1}^2$ *rounded* or *truncated* to a certain number of decimal places. For example, here is the evaluation of $\log_{10} 2$ rounded to four significant figures:

$$\begin{aligned} x_0 &= 2.000, \\ x_1 &= 4.000, \quad b_1 = 0, \quad & x_6 &= 1.845, \quad b_6 = 1, \\ x_2 &= 1.600, \quad b_2 = 1, \quad & x_7 &= 3.404, \quad b_7 = 0, \\ x_3 &= 2.560, \quad b_3 = 0, \quad & x_8 &= 1.159, \quad b_8 = 1, \\ x_4 &= 6.554, \quad b_4 = 0, \quad & x_9 &= 1.343, \quad b_9 = 0, \\ x_5 &= 4.295, \quad b_5 = 1, \quad & x_{10} &= 1.804, \quad b_{10} = 0, \end{aligned}$$

etc. Computational error has caused errors to propagate; the true value of x_{10} is 1.7977. This will eventually cause b_{19} to be computed incorrectly, and we get the binary value 0.01001101000100000011 which corresponds to the decimal equivalent 0.301031 rather than the true value given in Eq. (9).

With any method such as this it is necessary to examine the amount of computational error due to the limitations imposed. Exercise 27 of this section derives an upper bound for the error; working to four figures as above, we find that the error in the value of the logarithm will be less than 0.00044. Our answer above was more accurate than this primarily because x_0, x_1, x_2, and x_3 were obtained *exactly*.

This method is simple and quite interesting, but it is probably not the best way to calculate logarithms on a computer. Another method is given in exercise 25.

EXERCISES

1. [*00*] What is the smallest positive rational number?

2. [*00*] Is $1 + 0.239999999\ldots$ a decimal expansion?

3. [*02*] What is 7^{-3}?

▸ **4.** [*05*] What is $(0.125)^{-2/3}$?

5. [*05*] We defined real numbers in terms of a decimal expansion. Discuss how we could have defined them in terms of a binary expansion instead, and give a definition to replace Eq. (2).

6. [*10*] Let $x = m + 0.d_1d_2\ldots$ and $y = n + 0.e_1e_2\ldots$ be real numbers. Give a rule for determining whether $x = y$, $x < y$, or $x > y$, based on the decimal representation.

7. [*M23*] Given that x and y are integers, prove the laws of exponents, starting from the definition given by Eq. (3).

8. [*25*] Let m be a positive integer. *Prove* that every positive real number u has a unique positive mth root, by giving a method to construct successively n, d_1, d_2, etc. of the decimal expansion of the root.

9. [*M23*] Given that x and y are rational, prove the laws of exponents under the assumption that the laws hold when x and y are integers.

10. [*18*] Prove that $\log_{10} 2$ is not a rational number.

▸ **11.** [*10*] If $b = 10$ and $x = \log_{10} 2$, to how many decimal places of accuracy will we need to know the value of x in order to determine the first three decimal places of the decimal expansion of b^x? (*Note:* You may use the result of exercise 10 in your discussion.)

12. [*02*] Explain why Eq. (9) follows from Eqs. (7).

▸ **13.** [*M23*] (a) Given that x is a positive real number and n is a positive integer, prove that $\sqrt[n]{1+x} - 1 \leq x/n$. (b) Use this fact to justify the remarks following Eq. (6).

14. [*15*] Prove Eq. (11).

15. [*10*] Prove or disprove:

$$\log_b x/y = \log_b x - \log_b y, \qquad \text{if} \qquad x, y > 0.$$

16. [*00*] How can $\log_{10} x$ be expressed in terms of $\ln x$ and $\ln 10$?

▸ **17.** [*05*] What is $\log_2 32$? $\log_\pi \pi$? $\ln e$? $\log_b 1$? $\log_b (-1)$?

18. [*10*] Prove or disprove: $\log_8 x = \frac{1}{2} \log_2 x$.

▸ **19.** [*20*] If n is a 14-digit integer, will the value of n fit in a computer word with a capacity of 47 bits plus sign?

20. [*10*] Is there any simple relation between $\log_{10} 2$ and $\log_2 10$?

21. [*15*] Express $\log_b (\log_b x)$ in terms of $\ln(\ln x)$, $\ln(\ln b)$, and $\ln b$.

▸ **22.** [*20*] Prove that

$$\log_2 x \approx \ln x + \log_{10} x,$$

with less than 1% error! (Thus a table of natural logarithms and of common logarithms can be used to get approximate values of binary logarithms as well.)

23. [*M25*] Give a *geometric* proof that $\ln xy = \ln x + \ln y$, based on Fig. 6.

24. [*15*] Explain how the method used for calculating logarithms to the base 10 at the end of this section can be modified to produce logarithms to base 2.

25. [*20*] Suppose that we have a binary computer and a number x, $1 \le x < 2$. Show that the following algorithm, which uses only shifting, addition, and subtraction operations proportional to the number of places of accuracy desired, may be used to calculate an approximation to $y = \log_b x$:

L1. [Initialize.] Set $y \leftarrow 0$, $z \leftarrow x/2$, $k \leftarrow 1$.

L2. [Test for end.] If $x = 1$, stop.

L3. [Compare.] If $x - z < 1$, go to L5.

L4. [Reduce values.] Set $x \leftarrow x - z$, $z \leftarrow x/2^k$, $y \leftarrow y + \log_b (2^k/(2^k - 1))$, and go to L2.

L5. [Shift.] Set $z \leftarrow z/2$, $k \leftarrow k + 1$, and go to L2. ∎

[*Notes:* This method is very similar to the method used for division in computer hardware. We need an auxiliary *table* of $\log_b 2$, $\log_b (4/3)$, $\log_b (8/7)$, etc., to as many values as the precision of the computer. The algorithm involves an intentional computational error in step L5, as z is shifted to the right one place, so that eventually x will be reduced to 1 and the algorithm will terminate. This exercise is to explain why the above algorithm will terminate and why it computes an approximation to $\log_b x$.]

26. [*M27*] Determine upper bounds on the accuracy of the algorithm in the previous exercise, based on the precision used in the arithmetic operations.

▸ **27.** [*M25*] Consider the method for calculating $\log_{10} x$ discussed in the text. Let x_k' denote the computed approximation to x_k, determined as follows: $x(1 - \eta) \le 10^n x_0' \le x(1 + \epsilon)$; and in the determination of x_k' by Eqs. (15), the quantity y_k is used in place of $(x_{k-1}')^2$, where $(x_{k-1}')^2(1 - \eta) \le y_k \le (x_{k-1}')^2(1 + \epsilon)$. Here η and ϵ are small constants which reflect the upper and lower errors due to rounding or truncation. If $\log' x$ denotes the result of the calculations, show that after k steps we have

$$\log_{10} x + 2 \log_{10} (1 - \eta) - 1/2^k < \log' x < \log_{10} x + 2 \log_{10} (1 + \epsilon).$$

28. [*M30*] (R. Feynman.) Develop a method for computing b^x when $0 \le x < 1$, using only shifting, addition, and subtraction (similar to the algorithm in exercise 25), and analyze its accuracy.

29. [*HM20*] Let x be a real number greater than 1. (a) For what real number $b > 1$ is $b \log_b x$ a minimum? (b) For what *integer* $b > 1$ is it a minimum? (c) For what integer $b > 1$ is $(b + 1) \log_b x$ a minimum?

1.2.3. Sums and Products

Let $a_1, a_2, \ldots,$ be any sequence of numbers. We are often interested in sums such as $a_1 + a_2 + \cdots + a_n$, and this sum is more compactly written using the following notation:

$$\sum_{1 \le j \le n} a_j. \tag{1}$$

If n is zero or negative, the value of this summation is defined to be zero. In general if $R(j)$ is any relation involving j, the symbol

$$\sum_{R(j)} a_j \tag{2}$$

means the sum of all a_j where j is *an integer* satisfying the condition $R(j)$. If no such integers exist, notation (2) denotes zero. The letter j in (1) and (2) is a "dummy index" or "index variable" which has been introduced just for the purposes of this notation. Symbols used as index variables are usually the letters i, j, k, m, n, r, s, t (occasionally with subscripts or accent marks). The use of a $\sum$ and index variables to indicate summation was introduced by J. Lagrange in 1772.

The notation $\sum_{R(j)} a_j$ is used in this book as a condensed form of (2).

Strictly speaking, notation (1) is ambiguous, since it is not completely clear whether the summation is taken with respect to j or to n. In this particular case it would be rather silly to interpret (1) as a sum on values of $n \geq j$, but it is quite possible to construct meaningful examples in which the index variable is not clearly specified, for example, $\sum_{j \leq k} k^j$. In such cases the context must make clear which variable is a dummy variable and which variable has a significance which extends beyond its appearance in this notation; the example in the preceding sentence would presumably be used only if either j or k (not both) has exterior significance.

In most cases notation (2) will be used only if the sum is *finite*; i.e., only a finite number of values j satisfy $R(j)$, as in (1). When an infinite sum is used, for example,

$$\sum_{j \geq 1} a_j \equiv a_1 + a_2 + a_3 + \cdots,$$

the techniques of calculus must be employed; the precise meaning of (2) is then

$$\sum_{R(j)} a_j = \left(\lim_{n \to \infty} \sum_{R(j), 0 \leq j \leq n} a_j \right) + \left(\lim_{n \to \infty} \sum_{R(j), -n \leq j < 0} a_j \right), \tag{3}$$

provided both limits exist. If one or both limits fail to exist, the infinite sum is "divergent"; it does not exist.

If two or more conditions are placed under the $\sum$ sign, as in (3), we mean *all* conditions must hold.

Four simple algebraic operations on sums are very important, and a familiarity with these transformations makes the solution of many problems possible. We will now discuss these four operations.

a) *The distributive law*, for products of sums:

$$\left(\sum_{R(i)} a_i \right) \left(\sum_{S(j)} b_j \right) = \sum_{R(i)} \left(\sum_{S(j)} a_i b_j \right). \tag{4}$$

For example, consider the special case

$$\left(\sum_{1\le i\le 2} a_i\right)\left(\sum_{1\le j\le 3} b_j\right) = (a_1+a_2)(b_1+b_2+b_3)$$
$$= (a_1b_1+a_1b_2+a_1b_3)+(a_2b_1+a_2b_2+a_2b_3)$$
$$= \sum_{1\le i\le 2}\left(\sum_{1\le j\le 3} a_ib_j\right).$$

It is customary to drop the parentheses on the right-hand side of (4); "multiple summation" $\sum_{R(i)}(\sum_{S(j)} a_{ij})$ is written simply $\sum_{R(i)}\sum_{S(j)} a_{ij}$.

b) *Change of variable:*

$$\sum_{R(i)} a_i = \sum_{R(j)} a_j = \sum_{R(p(j))} a_{p(j)}. \tag{5}$$

This equation represents two kinds of transformations. In the first case we are simply changing the name of an index variable. The second case is a little more interesting: here $p(j)$ is a function of j which represents a permutation of the range; i.e., for each integer j satisfying the relation $R(j)$, there must be exactly one integer j satisfying the relation $R(p(j))$, and conversely. This condition is always satisfied in the important cases when $p(j) = c+j$ or $p(j) = c-j$, where c is an integer not depending on j, and these are the cases used most frequently in applications. For example,

$$\sum_{1\le j\le n} a_j = \sum_{1\le j-1\le n} a_{j-1} = \sum_{2\le j\le n+1} a_{j-1}. \tag{6}$$

The reader should study this example carefully.

The replacement of j by $p(j)$ cannot be done for all *infinite* sums. The operation is always valid if $p(j) = c \pm j$, as above, but in other cases some care must be used. [For example, see T. M. Apostol, *Mathematical Analysis* (Reading, Mass.: Addison-Wesley, 1957), Chapter 12. A sufficient condition to guarantee the validity of (5) for any permutation of the integers, $p(j)$, is that $\sum_{R(j)} |a_j|$ exists.]

c) *Interchanging order of summation:*

$$\sum_{R(i)}\sum_{S(j)} a_{ij} = \sum_{S(j)}\sum_{R(i)} a_{ij}. \tag{7}$$

Let us consider a very simple special case of this equation:

$$\sum_{R(i)}\sum_{1\le j\le 2} a_{ij} = \sum_{R(i)} (a_{i1}+a_{i2}),$$
$$\sum_{1\le j\le 2}\sum_{R(i)} a_{ij} = \sum_{R(i)} a_{i1} + \sum_{R(i)} a_{i2}.$$

By Eq. (7), these two are equal; this says no more than

$$\sum_{R(i)} (b_i + c_i) = \sum_{R(i)} b_i + \sum_{R(i)} c_i, \tag{8}$$

where we let

$$b_i = a_{i1} \qquad \text{and} \qquad c_i = a_{i2}.$$

The operation of interchanging the order of summation is extremely useful, since it often happens that we know a simple form for $\sum_{R(i)} a_{ij}$, but not for $\sum_{S(j)} a_{ij}$. We often need to interchange summation order in a more general case, where the relation $S(j)$ depends on i as well as j. In such a case, we can denote the relation by "$S(i,j)$." The interchange of summation can always be carried out, in theory at least, as follows:

$$\sum_{R(i)} \sum_{S(i,j)} a_{ij} = \sum_{S'(j)} \sum_{R'(i,j)} a_{ij}, \tag{9}$$

where $S'(j)$ is the relation "there is an integer i such that both $R(i)$ and $S(i,j)$ are true"; and $R'(i,j)$ is the relation "both $R(i)$ and $S(i,j)$ are true." For example, if the summation is $\sum_{1\le i\le n} \sum_{1\le j\le i} a_{ij}$, then $S'(j)$ is the relation "either $1 \le j \le 1$ or $1 \le j \le 2$ or . . . or $1 \le j \le n$," that is, $1 \le j \le n$, and $R'(i,j)$ is the relation "$1 \le i \le n$ and $1 \le j \le i$," that is, $j \le i \le n$. Thus,

$$\sum_{1\le i\le n} \sum_{1\le j\le i} a_{ij} = \sum_{1\le j\le n} \sum_{j\le i\le n} a_{ij}. \tag{10}$$

[*Note:* As in case (b), the operation of interchanging order of summation is *not always valid for infinite series.* If the series is "absolutely convergent," i.e., if $\sum_{R(i)} \sum_{S(j)} |a_{ij}|$ exists, it can be shown that Eqs. (7) and (9) are valid. Also if *either one* of $R(i)$ or $S(j)$ specifies a *finite* sum in Eq. (7), and if each infinite sum which appears is convergent, then the interchange is justified, and in particular Eq. (8) is always true for convergent infinite sums.]

d) *Manipulating the domain.* If $R(j)$ and $S(j)$ are *two* relations, we have

$$\sum_{R(j)} a_j + \sum_{S(j)} a_j = \sum_{R(j) \text{ or } S(j)} a_j + \sum_{R(j) \text{ and } S(j)} a_j. \tag{11}$$

For example,

$$\sum_{1\le j\le m} a_j + \sum_{m\le j\le n} a_j = \left(\sum_{1\le j\le n} a_j\right) + a_m, \tag{12}$$

assuming that $m \le n$. In this case "$R(j)$ and $S(j)$" becomes simply "$j = m$" so we reduced the second sum to simply "a_m." In most applications of Eq. (11), either $R(j)$ and $S(j)$ are simultaneously satisfied for only one or two values of j, or else it is impossible to have both $R(j)$ and $S(j)$ true for the same j. In the latter case, the second sum on the right-hand side of Eq. (11) simply disappears.

Now that we have given the four basic rules for manipulating sums, let us give some further illustrations of how to apply these techniques.

Example 1.

$$\sum_{0\le j\le n} a_j = \sum_{\substack{0\le j\le n,\\ j \text{ even}}} a_j + \sum_{\substack{0\le j\le n,\\ j \text{ odd}}} a_j \qquad \text{by rule (d)}$$

$$= \sum_{\substack{0\le 2j\le n,\\ 2j \text{ even}}} a_{2j} + \sum_{\substack{0\le 2j+1\le n,\\ 2j+1 \text{ odd}}} a_{2j+1} \qquad \text{by rule (b)}$$

$$= \sum_{0\le j\le n/2} a_{2j} + \sum_{0\le j<n/2} a_{2j+1}.$$

The last step merely consists of simplifying the relations below the $\sum$'s.

Example 2. Let

$$S_1 = \sum_{0\le i\le n} \sum_{0\le j\le i} a_i a_j = \sum_{0\le j\le n} \sum_{j\le i\le n} a_i a_j \qquad \text{by rule (c)} \quad \text{[cf. Eq. (10)]}$$

$$= \sum_{0\le i\le n} \sum_{i\le j\le n} a_i a_j \qquad \text{by rule (b),}$$

interchanging the names i and j and recognizing that $a_j a_i = a_i a_j$. If we denote the latter sum by S_2, we have

$$2S_1 = S_1 + S_2 = \sum_{0\le i\le n} \left(\sum_{0\le j\le i} a_i a_j + \sum_{i\le j\le n} a_i a_j \right) \qquad \text{by Eq. (8)}$$

$$= \sum_{0\le i\le n} \left(\left(\sum_{0\le j\le n} a_i a_j \right) + a_i a_i \right) \qquad \text{by rule (d) [cf. Eq. (12)]}$$

$$= \sum_{0\le i\le n} \sum_{0\le j\le n} a_i a_j + \sum_{0\le i\le n} a_i a_i \qquad \text{by Eq. (8)}$$

$$= \left(\sum_{0\le i\le n} a_i \right) \left(\sum_{0\le j\le n} a_j \right) + \left(\sum_{0\le i\le n} a_i^2 \right) \qquad \text{by rule (a)}$$

$$= \left(\sum_{0\le i\le n} a_i \right)^2 + \left(\sum_{0\le i\le n} a_i^2 \right) \qquad \text{by rule (b).}$$

Thus we have derived the important identity

$$\sum_{0\le i\le n} \sum_{0\le j\le i} a_i a_j = \frac{1}{2} \left(\left(\sum_{0\le i\le n} a_i \right)^2 + \left(\sum_{0\le i\le n} a_i^2 \right) \right). \qquad (13)$$

Example 3. The sum of a geometric progression. Assume that $x \neq 1$, $n \geq 0$. Then

$$
\begin{aligned}
a + ax + \cdots + ax^n &= \sum_{0 \leq j \leq n} ax^j && \text{by definition (2)} \\
&= a + \sum_{1 \leq j \leq n} ax^j && \text{by rule (d)} \\
&= a + x \sum_{1 \leq j \leq n} ax^{j-1} && \text{by a very special case of (a)} \\
&= a + x \sum_{0 \leq j \leq n-1} ax^j && \text{by rule (b) [cf. Eq. (6)]} \\
&= a + x \sum_{0 \leq j \leq n} ax^j - ax^{n+1} && \text{by rule (d).}
\end{aligned}
$$

Comparing the first relation with the fifth, we have

$$(1 - x) \sum_{0 \leq j \leq n} ax^j = a - ax^{n+1},$$

and so we obtain the basic formula

$$\sum_{0 \leq j \leq n} ax^j = a\left(\frac{1 - x^{n+1}}{1 - x}\right). \tag{14}$$

Example 4. The sum of an arithmetic progression. Assume that $n \geq 0$. Then

$$
\begin{aligned}
&a + (a + b) + \cdots + (a + nb) \\
&\quad = \sum_{0 \leq j \leq n} (a + bj) && \text{by definition (2)} \\
&\quad = \sum_{0 \leq n-j \leq n} (a + b(n - j)) && \text{by rule (b)} \\
&\quad = \sum_{0 \leq j \leq n} (a + bn - bj) && \text{by simplification} \\
&\quad = \sum_{0 \leq j \leq n} (2a + bn) - \sum_{0 \leq j \leq n} (a + bj) && \text{by Eq. (8)} \\
&\quad = (n + 1)(2a + bn) - \sum_{0 \leq j \leq n} (a + bj),
\end{aligned}
$$

since the first sum was simply a sum of $(n + 1)$ terms which did not depend on j. Now by equating the first and fifth expressions and dividing by 2, we obtain

$$\sum_{0 \leq j \leq n} (a + bj) = a(n + 1) + \tfrac{1}{2}bn(n + 1). \tag{15}$$

Note that we have obtained the important equations, (13), (14), and (15), purely by using simple manipulations of sums. Most textbooks would simply *state* those formulas, and prove them by *induction*. That is, of course, a perfectly valid procedure, except it does not give any insight into how on earth a person would ever have dreamed up the formula in the first place, except by some lucky guess. In the analysis of algorithms we are confronted with hundreds of sums which do not conform to any apparent pattern; by manipulating these sums, as above, we can often get the answer without the need for ingenious guesses.

There is a notation for products, analogous to our notation for sums:

$$\prod_{R(j)} a_j \tag{16}$$

stands for the product of all a_j for which the integer j satisfies $R(j)$. If no such integer j exists, the product is defined to have the value of unity (*not* zero). The question of infinite products is considered in exercise 21.

Operations (b), (c), and (d) are valid for the $\prod$-notation as well as for the $\sum$-notation, with suitable simple modifications. The exercises at the end of this section give a number of examples of the use of the product notation.

We conclude this section by mentioning another notation for multiple summation which is often convenient: a single $\sum$-sign may be used with one or more relations in *several* index variables, meaning that the sum is taken over all combinations of variables which meet the conditions. For example,

$$\sum_{0\le i\le n}\ \sum_{0\le j\le n} a_{ij} = \sum_{0\le i,j\le n} a_{ij}, \qquad \sum_{0\le i\le n}\ \sum_{0\le j\le i} a_{ij} = \sum_{0\le j\le i\le n} a_{ij}.$$

A further example which demonstrates the usefulness of this notation is

$$\sum_{\substack{j_1+\cdots+j_n=n,\\ j_1\ge\cdots\ge j_n\ge 0}} a_{j_1\ldots j_n},$$

where a is an n-tuply subscripted variable; for example, if $n = 5$ this notation stands for

$$a_{11111} + a_{21110} + a_{22100} + a_{31100} + a_{32000} + a_{41000} + a_{50000}.$$

(See the remarks on partitions of a number in Section 1.2.1.)

EXERCISES—First Set

1. [*01*] What is the meaning of notation (1), if $n = 3.14$?
2. [*10*] Without using the $\sum$-notation, write out the equivalent of

$$\sum_{0 \le n \le 5} \frac{1}{2n+1},$$

and also the equivalent of

$$\sum_{0\le n^2\le 5} \frac{1}{2n^2+1}.$$

▸ **3.** [*13*] Explain why the two results of the previous exercise are different, in spite of rule (b).

4. [*10*] Without using the $\sum$-notation, write out the equivalent of each side of Eq. (10) as a sum of sums for the case $n = 3$.

5. [*HM20*] Prove that rule (a) is valid for an arbitrary infinite series.

6. [*HM20*] Prove that rule (d) is valid for an arbitrary infinite series.

7. [*HM23*] Given that c is an integer, show that $\sum_{R(j)} a_j = \sum_{R(c-j)} a_{c-j}$, even if both series are infinite.

8. [*HM25*] Find an example of infinite series in which Eq. (7) is false.

▸ **9.** [*05*] Is the derivation of Eq. (14) valid even if $n = -1$?

10. [*05*] Is the derivation of Eq. (14) valid even if $n = -2$?

11. [*03*] What should the right-hand side of Eq. (14) be if $x = 1$?

12. [*10*] What is $1 + \frac{1}{7} + \frac{1}{49} + \frac{1}{343} + \cdots + (\frac{1}{7})^n$?

13. [*10*] Using Eq. (15) and assuming that $m \le n$, evaluate $\sum_{m\le j\le n} j$.

14. [*15*] Using the result of the previous exercise, evaluate $\sum_{m\le j\le n} \sum_{r\le k\le s} jk$.

▸ **15.** [*M22*] Compute the sum $1\times 2 + 2\times 2^2 + 3\times 2^3 + \cdots + n2^n$ for small values of n. Do you see the pattern developing in these numbers? If not, discover it by manipulations similar to those leading up to Eq. (14).

16. [*M22*] Prove that

$$\sum_{0\le j\le n} jx^j = \frac{nx^{n+2} - (n+1)x^{n+1} + x}{(x-1)^2},$$

if $x \ne 1$, without using mathematical induction.

▸ **17.** [*M00*] Let S be a set of integers. What is $\sum_{j \text{ in } S} 1$?

18. [*M20*] Show how to interchange the order of summation as in Eq. (9) given that $R(i)$ is the relation "n is a multiple of i" and $S(i, j)$ is the relation "$1 \le j < i$."

19. [*20*] What is $\sum_{m\le j\le n} (a_j - a_{j-1})$?

▸ **20.** [*25*] Dr. I. J. Matrix has observed a remarkable sequence of formulas:

$$9\times 1+2 = 11,\ 9\times 12+3 = 111,\ 9\times 123+4 = 1111,\ 9\times 1234+5 = 11111.$$

a) Write the good doctor's great discovery in terms of the $\sum$-notation.
b) Your answer to part (a) undoubtedly involves the number 10 as base of the decimal system; generalize this formula so that you get a formula which will perhaps work in any base b.
c) Prove the formula in part (b) by using formulas derived in the text or in exercise 16 above.

21. [*M25*] Give a definition for infinite products which is compatible both with Eq. (3) and with standard mathematical conventions in advanced calculus.

▸ **22.** [*20*] State the appropriate analogs of Eqs. (5), (7), (8), and (9) for *products* instead of sums.

23. [*10*] Explain why it is a good idea to define $\sum_{R(j)} a_j$ and $\prod_{R(j)} a_j$ as zero and one, respectively, when no integers satisfy $R(j)$.

24. [*20*] Suppose that $R(j)$ is true for only finitely many j. By induction on the number of integers satisfying $R(j)$, prove that $\log_b \prod_{R(j)} a_j = \sum_{R(j)} (\log_b a_j)$, assuming that all $a_j > 0$.

▸ **25.** [*15*] Consider the following derivation; is anything amiss?

$$\left(\sum_{1\le i\le n} a_i\right)\left(\sum_{1\le j\le n} \frac{1}{a_j}\right) = \sum_{1\le i\le n}\sum_{1\le j\le n} \frac{a_i}{a_j} = \sum_{1\le i\le n}\sum_{1\le i\le n} \frac{a_i}{a_i} = \sum_{1\le i\le n} 1 = n.$$

26. [*25*] Show that $\prod_{0\le i\le n}\prod_{0\le j\le i} a_i a_j$ may be expressed in terms of $\prod_{0\le i\le n} a_i$ by manipulating the $\prod$-notation as stated in exercise 22.

27. [*M20*] Generalize the result of exercise 1.2.1–9 by proving that

$$\prod_{1\le j\le n} (1 - a_j) \ge 1 - \sum_{1\le j\le n} a_j,$$

assuming that $0 < a_j < 1$.

28. [*M22*] Find a simple formula for $\prod_{2\le j\le n} (1 - 1/j^2)$.

▸ **29.** [*M30*] (a) Express $\sum_{0\le i\le n}\sum_{0\le j\le i}\sum_{0\le k\le j} a_i a_j a_k$ in terms of the multiple-summation notation explained at the end of the section. (b) Express the same sum in terms of $\sum_{0\le i\le n} a_i$, $\sum_{0\le i\le n} a_i^2$, and $\sum_{0\le i\le n} a_i^3$ [cf. Eq. (13)].

30. [*M23*] Prove "Lagrange's identity" without using induction:

$$\left(\sum_{1\le j\le n} a_j b_j\right)^2 = \left(\sum_{1\le j\le n} a_j^2\right)\left(\sum_{1\le j\le n} b_j^2\right) - \sum_{1\le k<j\le n} (a_k b_j - a_j b_k)^2.$$

▸ **31.** [*M23*] Show that $\sum_{1\le j<k\le n} (a_j - a_k)(b_j - b_k)$ can be expressed in terms of $\sum_{1\le j\le n} a_j b_j$, $\sum_{1\le j\le n} a_j$, and $\sum_{1\le j\le n} b_j$. Don't use induction.

32. [*M20*] Prove that

$$\prod_{1\le j\le n}\sum_{1\le i\le m} a_{ij} = \sum_{1\le i_1,\ldots,i_n\le m} a_{i_1 1}\ldots a_{i_n n}.$$

▸ **33.** [*M30*] One evening Dr. Matrix discovered some formulas that might even be classed as more remarkable than those of exercise 20:

$$\frac{1}{(a-b)(a-c)} + \frac{1}{(b-a)(b-c)} + \frac{1}{(c-a)(c-b)} = 0,$$

$$\frac{a}{(a-b)(a-c)} + \frac{b}{(b-a)(b-c)} + \frac{c}{(c-a)(c-b)} = 0,$$

$$\frac{a^2}{(a-b)(a-c)} + \frac{b^2}{(b-a)(b-c)} + \frac{c^2}{(c-a)(c-b)} = 1,$$

$$\frac{a^3}{(a-b)(a-c)} + \frac{b^3}{(b-a)(b-c)} + \frac{c^3}{(c-a)(c-b)} = a + b + c.$$

Prove that these formulas are a special case of a general law; let $x_1, x_2, \ldots, x_n$ be distinct numbers, and show that

$$\sum_{1\le j\le n}\left(x_j^r \Big/ \prod_{\substack{1\le k\le n,\\ k\ne j}} (x_j - x_k)\right) = \begin{cases} 0, & 0 \le r < n-1 \\ 1, & r = n-1 \\ \sum_{1\le j\le n} x_j, & r = n \end{cases}.$$

34. [*M25*] Prove that

$$\sum_{1\le k\le n} \frac{\prod_{1\le r\le n,\ r\ne m} (x+k-r)}{\prod_{1\le r\le n,\ r\ne k} (k-r)} = 1,$$

provided that $1 \le m \le n$ and x is arbitrary. For example, if $n = 4$ and $m = 2$, then

$$\frac{x(x-2)(x-3)}{(-1)(-2)(-3)} + \frac{(x+1)(x-1)(x-2)}{(1)(-1)(-2)} + \frac{(x+2)x(x-1)}{(2)(1)(-1)} + \frac{(x+3)(x+1)x}{(3)(2)(1)} = 1.$$

35. [*HM20*] The notation $\sup_{R(j)} a_j$ is used to denote the least upper bound of the elements a_j, in a manner exactly analogous to the $\sum$- and $\prod$-notations. (When $R(j)$ is satisfied for only finitely many j, the notation $\max_{R(j)} a_j$ is often used to denote the same quantity.) Show how rules (a), (b), (c), and (d) can be adapted for manipulation of *this* notation. In particular, discuss the following analog of rule (a):

$$(\sup_{R(i)} a_i) + (\sup_{S(j)} b_j) = \sup_{R(i)} (\sup_{S(j)} (a_i + b_j)),$$

and give a suitable definition for the notation when $R(j)$ is satisfied for *no* j.

EXERCISES—Second Set

Determinants and matrices. The following interesting problems are for the reader who has experienced at least an introduction to determinants and elementary matrix theory. A determinant may be evaluated by astutely combining the operations of: (a) factoring a quantity out of a row or column; (b) adding a multiple of one row (or column) to another row (or column); (c) expanding by "cofactors." The simplest and most often used version of operation (c) is to simply delete the entire first row and column, provided that the element in the upper left corner is $+1$ and the remaining elements in either the entire first row or the entire first column are zero; then evaluate the resulting smaller determinant. In general, the cofactor of an element a_{ij} in an $n \times n$ determinant is $(-1)^{i+j}$ times the $(n-1) \times (n-1)$ determinant obtained by deleting the row and column in which a_{ij} appeared. The value of a determinant is equal to $\sum a_{ij} \cdot \text{cofactor}(a_{ij})$ summed with either i or j held constant and with the other subscript varying from 1 to n.

If (b_{ij}) is the *inverse* of matrix (a_{ij}), then b_{ij} equals the cofactor of a_{ji} (note, *not* a_{ij}), divided by the determinant of the whole matrix. The notation δ_{ij} stands for the value *one* if $i = j$, *zero* otherwise.

The following types of matrices are of special importance:

Vandermonde's matrix,

$$a_{ij} = x_j^i$$

$$\begin{pmatrix} x_1 & x_2 & \dots & x_n \\ x_1^2 & x_2^2 & \dots & x_n^2 \\ \vdots & & & \vdots \\ x_1^n & x_2^n & \dots & x_n^n \end{pmatrix}$$

Combinatorial matrix,

$$a_{ij} = y + \delta_{ij}x$$

$$\begin{pmatrix} x+y & y & \dots & y \\ y & x+y & \dots & y \\ \vdots & & & \vdots \\ y & y & \dots & x+y \end{pmatrix}$$

Cauchy's matrix,

$$a_{ij} = 1/(x_i + y_j)$$

$$\begin{pmatrix} 1/(x_1+y_1) & 1/(x_1+y_2) & \dots & 1/(x_1+y_n) \\ 1/(x_2+y_1) & 1/(x_2+y_2) & \dots & 1/(x_2+y_n) \\ \vdots & & & \vdots \\ 1/(x_n+y_1) & 1/(x_n+y_2) & \dots & 1/(x_n+y_n) \end{pmatrix}$$

36. [*M23*] Show that the determinant of the combinatorial matrix is $x^{n-1}(x+ny)$.

▶ **37.** [*M24*] Show that the determinant of Vandermonde's matrix is

$$\prod_{1\le j\le n} x_j \prod_{1\le i<j\le n} (x_j - x_i).$$

▶ **38.** [*M25*] Show that the determinant of Cauchy's matrix is

$$\prod_{1\le i<j\le n} (x_j - x_i)(y_j - y_i) \Big/ \prod_{1\le i,\, j\le n} (x_i + y_j).$$

39. [*M23*] Show that the inverse of the combinatorial matrix is given by $b_{ij} = (-y + \delta_{ij}(x+ny))/x(x+ny)$.

40. [*M24*] Show that the inverse of Vandermonde's matrix is given by

$$b_{ij} = (-1)^{j+1} \sum_{\substack{1\le k_1<\cdots<k_{n-j}\le n,\\ k_1,\dots,k_{n-j}\ne i}} (x_{k_1}x_{k_2}\dots x_{k_{n-j}}) \Big/ x_i \prod_{\substack{1\le k\le n,\\ k\ne i}} (x_k - x_i)$$

Do not be dismayed by the complicated sum in the numerator—it is just the coefficient of x^{j-1} in the polynomial $(x_1 - x)\cdots(x_n - x)/(x_i - x)$.

41. [*M26*] Show that the inverse of Cauchy's matrix is given by

$$b_{ij} = \left(\prod_{1\le k\le n} (x_j + y_k)(x_k + y_i)\right) \Big/ (x_j + y_i)\left(\prod_{\substack{1\le k\le n,\\ k\ne j}} (x_j - x_k)\right)\left(\prod_{\substack{1\le k\le n,\\ k\ne i}} (y_i - y_k)\right).$$

42. [*M18*] What is the sum of all n^2 elements in the inverse of the combinatorial matrix?

43. [*M24*] What is the sum of all n^2 elements in the inverse of Vandermonde's matrix? [*Hint:* Use exercise 33.]

▶ **44.** [*M26*] What is the sum of all n^2 elements in the inverse of Cauchy's matrix?

▶ **45.** [*M25*] A *Hilbert matrix*, sometimes called "an $n \times n$ segment of *the* (infinite) Hilbert matrix" is a matrix for which $a_{ij} = 1/(i+j-1)$. Show that this is a special case of Cauchy's matrix, find its inverse, show that each element of the inverse is an integer, and show that the sum of all elements of the inverse is n^2. (*Note:* Hilbert matrices have often been used to test various matrix manipulation algorithms, because they are numerically unstable, and they have known inverses. However, it is a mistake to compare the *known* inverse, given in this exercise, to the *computed* inverse of a Hilbert matrix, since the matrix to be inverted must be expressed in rounded numbers beforehand; the inverse of an approximate Hilbert matrix will be somewhat different from the inverse of an exact one, due to the instability present. Since the elements of the inverse are integers, and since the inverse matrix is just as unstable as the original, the inverse can be specified exactly, and one could try to invert the inverse; however, the integers which appear in the inverse are quite large.) The solution to this problem requires an elementary knowledge of factorials and binomial coefficients, which are discussed in Sections 1.2.5 and 1.2.6.

▶ **46.** [*M30*] Let A be an $m \times n$ matrix, and let B be an $n \times m$ matrix. Given that $1 \le j_1, j_2, \ldots, j_m \le n$, let $A_{j_1 j_2 \ldots j_m}$ denote the $m \times m$ matrix consisting of columns $j_1, \ldots, j_m$ of A, and let $B_{j_1 j_2 \ldots j_m}$ denote the $m \times m$ matrix consisting of rows $j_1, \ldots, j_m$ of B. Prove that

$$\det(AB) = \sum_{1 \le j_1 < j_2 < \cdots < j_m \le n} \det(A_{j_1 j_2 \ldots j_m}) \det(B_{j_1 j_2 \ldots j_m}).$$

(Note the special cases: (i) $m = n$, (ii) $m = 1$, (iii) $B = A^T$.)

1.2.4. Integer Functions and Elementary Number Theory

If x is any real number, we write

$\lfloor x \rfloor =$ the greatest integer less than or equal to x (the "floor" of x);
$\lceil x \rceil =$ the least integer greater than or equal to x (the "ceiling" of x).

The notation $[x]$ is often used elsewhere for one or the other of these functions, usually the former; the notations above, which are due to K. E. Iverson, are more useful, because both functions occur about equally often in practice. The function $\lfloor x \rfloor$ is sometimes called the *entier* function, from the French word for "integer."

The following formulas and examples are easily verified:

$$\lfloor \sqrt{2} \rfloor = 1, \qquad \lceil \sqrt{2} \rceil = 2;$$

$$\lfloor +\tfrac{1}{2} \rfloor = 0, \qquad \lceil -\tfrac{1}{2} \rceil = 0, \qquad \lfloor -\tfrac{1}{2} \rfloor = -1 \quad (\textit{not}\ \text{zero!});$$

$$\lceil x \rceil = \lfloor x \rfloor \quad \text{if and only if } x \text{ is an integer},$$

$$\lceil x \rceil = \lfloor x \rfloor + 1 \quad \text{if and only if } x \text{ is not an integer};$$

$$\lfloor -x \rfloor = -\lceil x \rceil; \qquad x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1.$$

Exercises at the end of this section list other important formulas involving the floor and ceiling operations.

If x and y are any real numbers, we define the following binary operation:

$$x \bmod y = x - y\lfloor x/y \rfloor, \quad \text{if} \quad y \neq 0; \qquad x \bmod 0 = x. \tag{1}$$

From this definition we can see that when $y \neq 0$,

$$0 \leq \frac{x}{y} - \left\lfloor \frac{x}{y} \right\rfloor = \frac{x \bmod y}{y} < 1; \tag{2}$$

therefore

a) if $y > 0$, then $0 \leq x \bmod y < y$;

b) if $y < 0$, then $0 \geq x \bmod y > y$;

c) the quantity $x - (x \bmod y)$ is an integral multiple of y; and so we may think of $x \bmod y$ as *the remainder when x is divided by y.*

Thus, "mod" is a familiar operation when x and y are integers:

$$\begin{aligned} 5 \bmod 3 &= 2, \\ 18 \bmod 3 &= 0, \\ -2 \bmod 3 &= 1. \end{aligned} \tag{3}$$

We have $x \bmod y = 0$ if and only if x is a multiple of y, that is, if and only if x is divisible by y.

The "mod" operation is also useful when x and y take arbitrary real values; for example, with trigonometric functions we can write

$$\tan x = \tan (x \bmod \pi).$$

The quantity $x \bmod 1$ is the "fractional part" of x; we have, by Eq. (1),

$$x = \lfloor x \rfloor + (x \bmod 1). \tag{4}$$

In number theory, the abbreviation "mod" is used in a different but related sense; we will use the following form to express the number-theoretical concept of *congruence:*

$$x \equiv y \pmod{z} \tag{5}$$

means that $x \bmod z = y \bmod z$, that is, the difference $x - y$ is an integral multiple of z. Expression (5) is read, "x is congruent to y modulo z."

Let us now state the basic elementary properties of congruences which will be used in the number-theoretical arguments of this book. All variables in the following formulas are assumed to be integers. Two integers are said to be *relatively prime* if they have no common factor, i.e., if their greatest common

divisor is 1. The concept of relatively prime integers is a familiar one, since it is customary to say a fraction is in "lowest terms" when the numerator is relatively prime to the denominator.

LAW A. If $a \equiv b$ and $x \equiv y$, then $a \pm x \equiv b \pm y$ and $ax \equiv by$ (modulo m).

LAW B. If $ax \equiv by$ and $a \equiv b$, and if a is relatively prime to m, then $x \equiv y$ (modulo m).

LAW C. $a \equiv b$ (modulo m) if and only if $an \equiv bn$ (modulo mn).

LAW D. If r is relatively prime to s, then $a \equiv b$ (modulo rs) if and only if $a \equiv b$ (modulo r) and $a \equiv b$ (modulo s).

Law A states that we can do addition, subtraction, and multiplication (and hence we can take powers x^n, for $n \geq 0$) modulo m just as we do ordinary addition, subtraction, multiplication, and taking powers. Law B considers the operation of division, and shows that in certain cases (namely, that the divisor is relatively prime to the modulus) we can also divide out common factors. Laws C and D consider relations when the modulus is changed.

As an example of these relationships, we will prove an important theorem.

Theorem F (*Fermat's theorem*, 1640). *If p is a prime number, then $a^p \equiv a$* (modulo p).

Proof. If a is a multiple of p, obviously $a^p \equiv 0 \equiv a$ (modulo p). So we need only consider the case $a \bmod p \neq 0$. Since p is a prime number, this means that a is relatively prime to p. Consider the numbers

$$0 \bmod p, \quad a \bmod p, \quad 2a \bmod p, \quad \ldots, \quad (p-1)a \bmod p. \tag{6}$$

These p numbers are all *distinct*, for if $ax \bmod p = ay \bmod p$, then by definition (5) $ax \equiv ay$ (modulo p); hence by Law B, $x \equiv y$ (modulo p).

Since (6) gives p distinct numbers, all nonnegative and less than p, we see that the first number is zero and the rest are the integers $1, 2, \ldots, p-1$ in some order. Therefore by Law A,

$$(a)(2a) \cdots ((p-1)a) \equiv 1 \cdot 2 \cdots (p-1) \quad \text{(modulo } p\text{)}. \tag{7}$$

Multiplying each side of this congruence by a, we obtain

$$a^p(1 \cdot 2 \cdots (p-1)) \equiv a(1 \cdot 2 \cdots (p-1)) \quad \text{(modulo } p\text{)}, \tag{8}$$

and this proves the theorem, since each of the factors $1, 2, \ldots, (p-1)$ is relatively prime to p and can be canceled by Law B. ∎

Exercises 17 through 21 below develop the basic laws underlying the elementary theory of numbers.

EXERCISES

1. [*00*] What are $\lfloor 1.1 \rfloor$, $\lfloor -1.1 \rfloor$, $\lceil -1.1 \rceil$, $\lfloor 0.99999 \rfloor$, and $\lfloor \log_2 35 \rfloor$?

▶ **2.** [*01*] What is $\lceil \lfloor x \rfloor \rceil$?

3. [*M10*] Let n be an integer, and let x be a real number. Prove that

a) $\lfloor x \rfloor < n$ if and only if $x < n$; b) $n \leq \lfloor x \rfloor$ if and only if $n \leq x$;
c) $\lceil x \rceil \leq n$ if and only if $x \leq n$; d) $n < \lceil x \rceil$ if and only if $n < x$;
e) $\lfloor x \rfloor = n$ if and only if $x - 1 < n \leq x$, and if and only if $n \leq x < n + 1$;
f) $\lceil x \rceil = n$ if and only if $x \leq n < x + 1$, and if and only if $n - 1 < x \leq n$.

These formulas are the most important tools for proving statements about $\lfloor x \rfloor$ *and* $\lceil x \rceil$.

▶ **4.** [*M10*] Using the previous exercise, prove that $\lfloor -x \rfloor = -\lceil x \rceil$.

5. [*16*] Given that x is a positive real number, state a simple formula which expresses "*x rounded to the nearest integer*." The desired rounding rule is to produce $\lfloor x \rfloor$ when $x \bmod 1 < \frac{1}{2}$, and to produce $\lceil x \rceil$ when $x \bmod 1 \geq \frac{1}{2}$. Your answer should be a single formula which covers both cases. Discuss the rounding which would be obtained by your formula when x is negative.

▶ **6.** [*20*] Which of the following equations are true for all positive real numbers x? (a) $\lfloor \sqrt{\lfloor x \rfloor} \rfloor = \lfloor \sqrt{x} \rfloor$; (b) $\lceil \sqrt{\lceil x \rceil} \rceil = \lceil \sqrt{x} \rceil$; (c) $\lceil \sqrt{\lfloor x \rfloor} \rceil = \lceil \sqrt{x} \rceil$.

7. [*M15*] Show that $\lfloor x \rfloor + \lfloor y \rfloor \leq \lfloor x + y \rfloor$ and that equality holds if and only if $x \bmod 1 + y \bmod 1 < 1$. Does a similar formula hold for ceilings?

8. [*00*] What are 100 mod 3, 100 mod 7, -100 mod 7, -100 mod 0?

9. [*05*] What are 5 mod -3, 18 mod -3, -2 mod -3?

▶ **10.** [*10*] What are 1.1 mod 1, 0.11 mod .1, 0.11 mod $-.1$?

11. [*00*] What does "$x \equiv y$ (modulo 0)" mean by our conventions?

12. [*00*] What integers are relatively prime to 1?

13. [*M00*] By convention, we say the greatest common divisor of 0 and n is $|n|$. What integers are relatively prime to 0?

▶ **14.** [*12*] If $x \bmod 3 = 2$ and $x \bmod 5 = 3$, what is $x \bmod 15$?

15. [*10*] Prove that $z(x \bmod y) = (zx) \bmod (zy)$. (Note that Law C is an immediate consequence of this distributive law.)

16. [*M10*] Assume that $y > 0$. Show that if $(x - z)/y$ is an integer and if $0 \leq z < y$, then $z = x \bmod y$.

17. [*M15*] Prove Law A directly from the definition of congruence, and also prove half of Law D: If $a \equiv b$ (modulo rs), then $a \equiv b$ (modulo r) and $a \equiv b$ (modulo s). (Here r, s are arbitrary integers.)

18. [*M15*] Using Law B, prove the other half of Law D: If $a \equiv b$ (modulo r) and $a \equiv b$ (modulo s), then $a \equiv b$ (modulo rs), provided that r and s are relatively prime.

▶ **19.** [*M10*] (Law of inverses.) If n is relatively prime to m, there is an integer n' such that $nn' \bmod m = 1$. Prove this, using the extension of Euclid's algorithm (Algorithm 1.2.1E).

20. [*M15*] Use the law of inverses and Law A to prove Law B.

21. [*M16*] Use Law B and exercise 1.2.1–5 to prove that every integer $n > 1$ has a *unique* representation as a product of primes (except for order of factors); i.e., that there is exactly one way to write $n = p_1 p_2 \cdots p_k$, where each p_j is prime and $p_1 \le p_2 \le \cdots \le p_k$.

▸ **22.** [*M10*] Give an example to show that Law B is not always true if a is not relatively prime to m.

23. [*M10*] Give an example to show that Law D is not always true if r is not relatively prime to s.

▸ **24.** [*M20*] To what extent can Laws A, B, C, and D be generalized to apply to arbitrary real numbers instead of integers?

25. [*M00*] Show that, according to Theorem F, $a^{p-1} \bmod p = 1$ if a is not a multiple of p, and that $a^{p-1} \bmod p = 0$ if a is a multiple of p, whenever p is a prime number.

26. [*M15*] Let p be an *odd* prime number, let a be any integer, and let $b = a^{(p-1)/2}$. Show that $b \bmod p$ is either 0 or 1 or $p - 1$. [*Hint:* Consider $(b + 1)(b - 1)$.]

27. [*M15*] Given that n is a positive integer, let $\varphi(n)$ be the number of values among $0, 1, \ldots, n - 1$ that are relatively prime to n. Thus $\varphi(1) = 1$, $\varphi(2) = 1$, $\varphi(3) = 2$, $\varphi(4) = 2$, etc. Show that $\varphi(p) = p - 1$ if p is a prime number; and evaluate $\varphi(p^e)$, where e is a positive integer.

▸ **28.** [*M25*] Show that the method used to prove Theorem F can be used to prove the following extension, which is called *Euler's theorem:* $a^{\varphi(m)} \bmod m = 1$, for *any* positive integer m, when a is relatively prime to m. (In particular, the number n' in exercise 19 may be taken to be $n^{\varphi(m)-1} \bmod m$.)

29. [*M20*] A function $f(n)$ of positive integers n is called *multiplicative* if $f(rs) = f(r)f(s)$ whenever r and s are relatively prime. Show that the following functions are multiplicative: (a) $f(n) = n^k$; (b) $f(n) = 0$ if n is divisible by k^2 for some integer $k > 1$, $f(n) = 1$ otherwise; (c) $f(n) = c^k$, where k is the number of distinct primes which divide n; (d) the product of any two multiplicative functions.

30. [*M30*] Prove that the function $\varphi(n)$ of exercise 27 is multiplicative. Using this fact, evaluate $\varphi(1000000)$ and give a method for evaluating $\varphi(n)$ in a simple way once n has been factored into primes.

31. [*M22*] Prove that if $f(n)$ is multiplicative, so is $g(n) = \sum_{d \backslash n} f(d)$. The notation $d \backslash n$ means "d divides n," that is, d is a positive integer and $n \bmod d = 0$.

32. [*M18*] In connection with the notation in the previous exercise, show that

$$\sum_{d \backslash n} \sum_{c \backslash d} f(c, d) = \sum_{c \backslash n} \sum_{d \backslash (n/c)} f(c, cd),$$

for any function $f(x, y)$.

33. [*M18*] If n, m are integers, evaluate

$$\text{(a)} \left\lfloor \frac{n+m}{2} \right\rfloor + \left\lfloor \frac{n-m+1}{2} \right\rfloor; \qquad \text{(b)} \left\lceil \frac{n+m}{2} \right\rceil + \left\lceil \frac{n-m+1}{2} \right\rceil.$$

(The special case $m = 0$ is worth noting.)

▸ **34.** [*M21*] What conditions on the real number $b > 1$ are necessary and sufficient to guarantee that $\lfloor \log_b x \rfloor = \lfloor \log_b \lfloor x \rfloor \rfloor$ for all real $x \ge 1$?

▶ **35.** [*M20*] Given that m, n are integers and $n > 0$, prove that $\lfloor (x + m)/n \rfloor = \lfloor (\lfloor x \rfloor + m)/n \rfloor$ for all real x. (When $m = 0$, we have an important special case.) Does an analogous result hold for the ceiling function?

36. [*M23*] Prove that $\sum_{1 \le k \le n} \lfloor k/2 \rfloor = \lfloor n^2/4 \rfloor$; also evaluate $\sum_{1 \le k < n} \lceil k/2 \rceil$.

▶ **37.** [*M30*] Let m, n be integers, $n > 0$. Show that

$$\sum_{0 \le k < n} \left\lfloor \frac{mk + x}{n} \right\rfloor = \frac{(m-1)(n-1)}{2} + \frac{d-1}{2} + x - x \bmod d,$$

where d is the greatest common divisor of m and n, and x is any real number.

38. [*M22*] Prove that, for all positive integers n and for any real x,

$$\lfloor x \rfloor + \left\lfloor x + \frac{1}{n} \right\rfloor + \cdots + \left\lfloor x + \frac{n-1}{n} \right\rfloor = \lfloor nx \rfloor.$$

39. [*HM35*] A function f for which

$$f(x) + f\left(x + \frac{1}{n}\right) + \cdots + f\left(x + \frac{n-1}{n}\right) = f(nx),$$

whenever n is a positive integer, is called a *replicative function*. The previous exercise establishes the fact that $\lfloor x \rfloor$ is replicative. Show that the following functions are replicative:

a) $f(x) = x - \frac{1}{2}$;
b) $f(x) = 1$, if x is an integer, 0 otherwise;
c) $f(x) = 1$, if x is a *positive* integer, 0 otherwise;
d) $f(x) = 1$, if there exists a rational number r and an integer m such that $x = r\pi + m$, 0 otherwise;
e) three other functions like the one in (d) with r and/or m restricted to positive values;
f) $f(x) = \log |2 \sin \pi x|$, if the value $f(x) = -\infty$ is allowed;
g) the sum of any two replicative functions;
h) a constant multiple of a replicative function;
i) the function $g(x) = f(x - \lfloor x \rfloor)$, where $f(x)$ is replicative.

40. [*HM50*] Study the class of replicative functions; determine all replicative functions of a special type (e.g., is the function in (a) of exercise 39 the only continuous replicative function?). It may be interesting to study also the more general class of functions for which

$$f(x) + \cdots + f\left(x + \frac{n-1}{n}\right) = a_n f(nx) + b_n.$$

Here a_n, b_n are numbers which depend on n but not on x. Derivatives and (if $b_n = 0$) integrals of these functions are of the same type. If we require that $b_n = 0$, we have, for example, the Bernoulli polynomials, the trigonometric functions $\cot \pi x$ and $\csc^2 \pi x$, as well as Hurwitz's generalized zeta function $\zeta(s, x) = \sum_{k \ge 0} 1/(k + x)^s$ for fixed s. With $b_n \neq 0$ we have still other well-known functions, e.g., the psi-function. For further properties of these functions, see L. J. Mordell, "Integral Formulae of Arithmetical Character," *J. London Math. Soc.* **33** (1958), 371–375.

41. [*M23*] Let $a_1, a_2, a_3, \ldots$ be the sequence $1, 2, 2, 3, 3, 3, 4, 4, 4, 4, \ldots$; find an expression for a_n in terms of n (using the floor and/or ceiling operation).

42. [*M24*] (a) Prove that

$$\sum_{1\le k\le n} a_k = na_n - \sum_{1\le k<n} k(a_{k+1} - a_k), \qquad \text{if} \qquad n > 0.$$

(b) The preceding formula is useful for evaluating certain sums involving the floor function. Prove that, if b is an integer ≥ 2,

$$\sum_{1\le k\le n} \lfloor \log_b k \rfloor = (n+1)\lfloor \log_b n \rfloor - (b^{\lfloor \log_b n \rfloor + 1} - b)/(b-1).$$

43. [*M23*] Evaluate $\sum_{1\le k\le n} \lfloor \sqrt{k} \rfloor$.

44. [*M24*] Show that $\sum_{k\ge 0} \sum_{1\le j<b} \lfloor (n + jb^k)/b^{k+1} \rfloor = n$, if b and n are integers, and $b \ge 2$.

▸ **45.** [*M28*] The result of exercise 37 is somewhat surprising, since it implies that

$$\sum_{0\le k<n} \left\lfloor \frac{mk+x}{n} \right\rfloor = \sum_{0\le k<m} \left\lfloor \frac{nk+x}{m} \right\rfloor.$$

This "reciprocity relationship" is one of many similar formulas (cf. Section 3.3.3). Show that for any function f

$$\sum_{0\le j<n} f\left(\left\lfloor \frac{mj}{n} \right\rfloor\right) = \sum_{0\le r<m} \left\lceil \frac{rn}{m} \right\rceil (f(r-1) - f(r)) + nf(m-1).$$

In particular, prove that

$$\sum_{0\le j<n} \binom{\lfloor mj/n \rfloor + 1}{k} + \sum_{0\le j<m} \left\lceil \frac{jn}{m} \right\rceil \binom{j}{k-1} = n\binom{m}{k}.$$

[*Hint:* Consider the change of variable, $r = \lfloor mj/n \rfloor$. Binomial coefficients $\binom{m}{k}$ are discussed in Section 1.2.6.]

46. [*M29*] (General reciprocity law.) Extend the formula of exercise 45 to obtain an expression for $\sum_{0\le j<\alpha n} f(\lfloor mj/n \rfloor)$, where α is *any* positive real number.

47. [*M31*] When p is an odd prime number, the *Legendre symbol*, $\left(\frac{q}{p}\right)$, is defined to be $+1$, 0, or -1, depending on whether $q^{(p-1)/2} \bmod p = 1$, 0, or $p-1$. (Cf. exercise 26.)

a) Given that q is not a multiple of p, show that the numbers

$$(-1)^{\lfloor 2kq/p \rfloor}(2kq \bmod p), \qquad 0 < k < p/2,$$

are congruent in some order to the numbers $2, 4, \ldots, p-1$ (modulo p). Hence $\left(\frac{q}{p}\right) = (-1)^\sigma$ where $\sigma = \sum_{0\le k<p/2} \lfloor 2kq/p \rfloor$.

b) Use the result of (a) to calculate $\left(\frac{2}{p}\right)$.

c) Given that q is odd, show that $\sum_{0\le k<p/2} \lfloor 2kq/p \rfloor \equiv \sum_{0\le k<p/2} \lfloor kq/p \rfloor$ (modulo 2).

[*Hint:* Consider $\lfloor (p-1-2k)q/p \rfloor$.]

d) Use the general reciprocity formula of exercise 46 to obtain the *law of quadratic reciprocity*, $(\frac{q}{p})(\frac{p}{q}) = (-1)^{(p-1)(q-1)/4}$, given that p and q are distinct odd primes.

48. [*M26*] Prove or disprove the following identities given that m and n are integers:

$$\text{(a)}\ \left\lfloor \frac{m+n-1}{n} \right\rfloor = \left\lceil \frac{m}{n} \right\rceil; \qquad \text{(b)}\ \left\lfloor \frac{n+2-\lfloor n/25 \rfloor}{3} \right\rfloor = \left\lfloor \frac{8n+24}{25} \right\rfloor.$$

1.2.5. Permutations and Factorials

A *permutation of n objects* is an arrangement of n distinct objects in a row. There are six permutations of three objects a, b, c:

$$a\ b\ c, \qquad a\ c\ b, \qquad b\ a\ c, \qquad b\ c\ a, \qquad c\ a\ b, \qquad c\ b\ a. \tag{1}$$

The properties of permutations are of great importance in the analysis of algorithms, and we will deduce many interesting facts about them later in this book. At this point we will simply *count* them, i.e., we will determine how many permutations of n objects are possible: There are n ways to choose the leftmost object, and once this choice has been made, there are $(n - 1)$ ways to select a different object to place *next* to it; this gives us $n(n - 1)$ choices for the first two positions. Similarly, we find there are $(n - 2)$ choices for the third object distinct from the first two, and a total of $n(n - 1)(n - 2)$ possible ways to choose the first three objects. In general, if p_{nk} denotes the number of ways to choose k objects out of n and to arrange them in a row, we see that

$$p_{nk} = n(n - 1) \cdots (n - k + 1). \tag{2}$$

The total number of permutations is $p_{nn} = n(n - 1) \cdots (1)$.

The process of *constructing* all permutations of n objects in an inductive manner, assuming that all permutations of $n - 1$ objects have been constructed, is very important in our applications. Let us rewrite (1) using the numbers 1, 2, 3 instead of the letters a, b, c; the permutations of order 3 are

$$1\ 2\ 3, \qquad 1\ 3\ 2, \qquad 2\ 1\ 3, \qquad 2\ 3\ 1, \qquad 3\ 1\ 2, \qquad 3\ 2\ 1. \tag{3}$$

Consider how to get from this array to the permutations of 4 objects. There are two principal methods for going from $n - 1$ objects to n objects.

METHOD 1. For each permutation $a_1 a_2 \cdots a_{n-1}$ on $(n - 1)$ elements, form n others by inserting the number n in all possible places, obtaining

$$n\ a_1\ a_2 \cdots a_{n-1}, \quad a_1\ n\ a_2 \cdots a_{n-1}, \quad \ldots,$$
$$a_1\ a_2 \cdots n\ a_{n-1}, \quad a_1\ a_2 \cdots a_{n-1}\ n.$$

For example, from the permutation 2 3 1 in (3), we get 4 2 3 1, 2 4 3 1, 2 3 4 1, 2 3 1 4. It is clear that all permutations of n objects are obtained in this manner and that no permutation is obtained more than once.

METHOD 2. For each permutation $a_1\ a_2\ \ldots\ a_{n-1}$ of the elements $\{1,\ 2,\ \ldots,\ n-1\}$, form n others as follows: First construct the array

$$a_1\ a_2\ \ldots\ a_{n-1}\ \tfrac{1}{2},\quad a_1\ a_2\ \ldots\ a_{n-1}\ \tfrac{3}{2},\quad \ldots,\quad a_1\ a_2\ \cdots\ a_{n-1}\ (n-\tfrac{1}{2}).$$

Then rename the elements of each permutation using the numbers $1, 2, \ldots, n$, *preserving order*. For example, from the permutation 2 3 1 in (3) we get

$$2\ 3\ 1\ \tfrac{1}{2},\qquad 2\ 3\ 1\ \tfrac{3}{2},\qquad 2\ 3\ 1\ \tfrac{5}{2},\qquad 2\ 3\ 1\ \tfrac{7}{2}$$

and, renaming, we get

$$3\ 4\ 2\ 1,\qquad 3\ 4\ 1\ 2,\qquad 2\ 4\ 1\ 3,\qquad 2\ 3\ 1\ 4.$$

Another way to describe the same process is to take the permutation $a_1\ a_2\ \cdots\ a_{n-1}$ and a number k, $1 \le k \le n$; add one to each a_j whose value is $\ge k$, thus obtaining a permutation $b_1\ b_2\ \cdots\ b_{n-1}$ on the elements $\{1, \ldots, k-1, k+1, \ldots, n\}$; now $b_1\ b_2\ \cdots\ b_{n-1}\ k$ is a permutation on $\{1, \ldots, n\}$.

Again it is clear that we obtain each permutation on n elements exactly once by this construction. A similar method (which puts k at the left instead of the right, or which puts k in any other fixed position) could of course also be used.

If p_n is the number of permutations of n objects, both of these methods show that $p_n = np_{n-1}$, and this offers us two further proofs that $p_n = n(n-1)\cdots(1)$, as we already established in Eq. (2).

The important quantity p_n is called n *factorial* and it is written

$$n! = 1\cdot 2\cdot \cdots \cdot n = \prod_{1\le k\le n} k. \tag{4}$$

Our convention on vacuous products (cf. Section 1.2.3) gives us the value

$$0! = 1, \tag{5}$$

and with this convention the basic identity

$$n! = (n-1)!\ n \tag{6}$$

is valid for all positive integers n.

Factorials come up sufficiently often in computer work that the reader is advised to memorize the values of the first few factorials:

$$0! = 1,\quad 1! = 1,\quad 2! = 2,\quad 3! = 6,\quad 4! = 24,\quad 5! = 120.$$

The factorials increase very rapidly; the number 1000! is an integer with over 2500 decimal digits.

It is helpful to keep the value

$$10! = 3{,}628{,}800$$

in mind; one should remember that 10! is about $3\frac{1}{2}$ million. In a sense, the number 10! represents an approximate dividing line between things which are practical to compute and things which are not. If an algorithm requires the testing of more than 10! cases, chances are it may take too long to run on a computer to be practical. On the other hand, if we are to test 10! cases and each case requires, say, one millisecond of computer time, then the entire run will take about an hour. These comments are very vague, of course, but they can be useful to give an intuitive idea of what is computationally feasible.

It is only natural to wonder what relation $n!$ bears to other quantities in mathematics; is there any way to tell how large 1000! is, without laboriously carrying out the multiplications implied in Eq. (4)? The answer was found by James Stirling in his famous work *Methodus Differentialis* (1730), p. 137; we have

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n. \tag{7}$$

The "$\approx$" sign which appears here denotes "approximately equal," and "e" is the base of natural logarithms introduced in Section 1.2.2. We will prove Stirling's approximation (7) in Section 1.2.11.2.

As an example of the use of this formula, we may compute

$$40320 = 8! \approx 4\sqrt{\pi}\left(\frac{8}{e}\right)^8 = 2^{26}\sqrt{\pi}e^{-8} \approx (67108864)(1.77245)(0.00033546)$$
$$\approx 39902.$$

In this case the error is about 1%; we will see later that the relative error is approximately $1/12n$.

In addition to the approximate value given by Eq. (7), we can also rather easily obtain the exact value of $n!$ factored into primes. In fact, the prime p is a divisor of $n!$ with the multiplicity

$$\mu = \left\lfloor\frac{n}{p}\right\rfloor + \left\lfloor\frac{n}{p^2}\right\rfloor + \left\lfloor\frac{n}{p^3}\right\rfloor + \cdots = \sum_{k>0}\left\lfloor\frac{n}{p^k}\right\rfloor. \tag{8}$$

For example, if $n = 1000$ and $p = 3$, we have

$$\mu = \left\lfloor\frac{1000}{3}\right\rfloor + \left\lfloor\frac{1000}{9}\right\rfloor + \left\lfloor\frac{1000}{27}\right\rfloor + \left\lfloor\frac{1000}{81}\right\rfloor + \left\lfloor\frac{1000}{243}\right\rfloor + \left\lfloor\frac{1000}{729}\right\rfloor$$
$$= 333 + 111 + 37 + 12 + 4 + 1 = 498,$$

so 1000! is divisible by 3^{498} but not by 3^{499}. Although formula (8) is written as an infinite sum, it is really finite for any particular values of n and p, because all of the terms are eventually zero. It follows from exercise 1.2.4–35 that $\lfloor n/p^{k+1}\rfloor = \lfloor\lfloor n/p^k\rfloor/p\rfloor$, and this fact facilitates the calculation in Eq. (8), since we just divide the value of the previous term by p and discard the remainder.

We can prove the correctness of Eq. (8) by observing that $\lfloor n/p^k \rfloor$ is the number of integers among $\{1, 2, \ldots, n\}$ which are multiples of p^k. Thus, if we study the integers in the product (4), any integer which is divisible by p^j but not by p^{j+1} is counted exactly j times: once in $\lfloor n/p \rfloor$, once in $\lfloor n/p^2 \rfloor$, ..., once in $\lfloor n/p^j \rfloor$. This accounts for all occurrences of p as a factor of $n!$.

Another natural question arises: Now that we have defined $n!$ for nonnegative integers n, perhaps the factorial function is meaningful also for rational values of n, and even for real values. What is $(\frac{1}{2})!$, for example? Let us illustrate this point by introducing the "termial" function

$$n? = 1 + 2 + \cdots + n = \sum_{1 \le k \le n} k, \tag{9}$$

which is analogous to the factorial function, except we are adding instead of multiplying. We already know the sum of this arithmetic progression (cf. Eq. 1.2.3–15):

$$n? = \tfrac{1}{2}n(n+1). \tag{10}$$

This suggests a good way to generalize the "termial" function to arbitrary n, by using Eq. (10) instead of Eq. (9). We have $(\frac{1}{2})? = \frac{3}{8}$.

Stirling himself made several attempts to generalize $n!$ to noninteger n. He extended the approximation (Eq. 7) into an infinite sum, but unfortunately the sum did not converge for any value of n; the approximation method gives extremely good approximations, but it cannot be extended to give an *exact* value. [For a discussion of this rather unusual situation, see K. Knopp, *Theory and Application of Infinite Series*, 2nd ed. (Glasgow: Blackie, 1951), pp. 518–520, 527, 534.]

Stirling tried again, by noticing that

$$\begin{aligned} n! = 1 &+ \left(1 - \frac{1}{1!}\right) n + \left(1 - \frac{1}{1!} + \frac{1}{2!}\right) n(n-1) \\ &+ \left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!}\right) n(n-1)(n-2) + \cdots . \end{aligned} \tag{11}$$

(We will prove this formula in the next section.) The apparently infinite sum in Eq. (11) is in reality finite for any nonnegative integer n; however, it does not provide the desired generalization of $n!$, since the infinite sum does not exist *except* when n is a nonnegative integer. (Cf. exercise 16.)

Still undaunted, he found a sequence $a_1, a_2, \ldots$ such that

$$\ln n! = a_1 n + a_2 n(n-1) + \cdots = \sum_{k \ge 0} a_{k+1} \prod_{0 \le j \le k} (n - j). \tag{12}$$

He was unable to *prove* that this sum defined $n!$ for all fractional values of n, although he was able to find the value of $(\frac{1}{2})! = \sqrt{\pi}/2$.

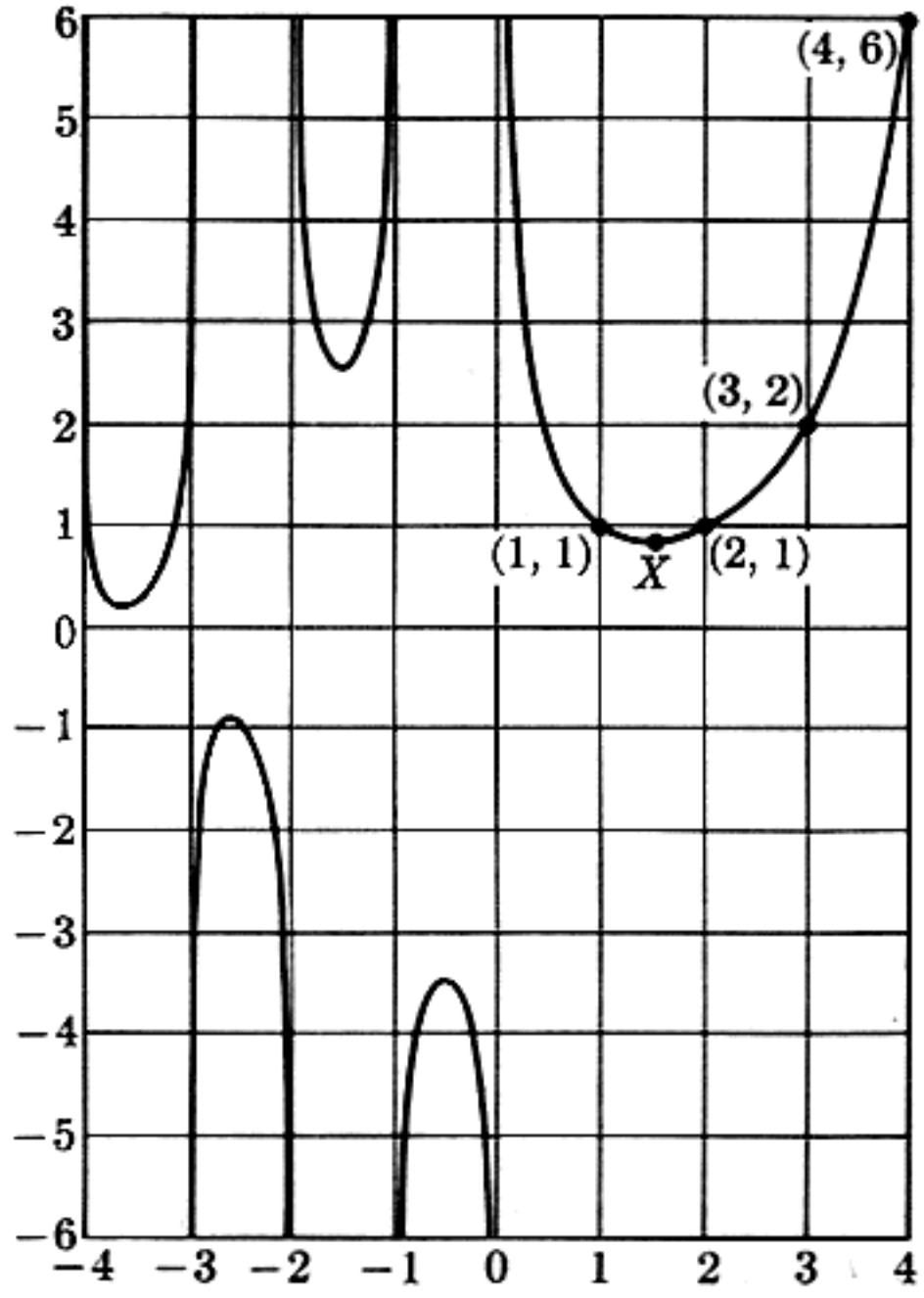


Fig. 7. The function $\Gamma(x) = (x - 1)!$. Point X has the coordinates (1.4616321450, 0.8856031944).

At about the same time, Leonhard Euler considered the same problem, and he was the first to find the appropriate generalization:

$$n! = \lim_{m\to\infty} \frac{m^n m!}{(n+1)(n+2)\cdots(n+m)}. \tag{13}$$

Euler communicated this idea in a letter to Christian Goldbach, on Oct. 13, 1729. His formula defines $n!$ for any value of n except negative integers (when the denominator in Eq. (13) becomes zero), and in this case $n!$ is taken to be infinite.

Nearly two centuries later, in 1900, C. Hermite proved that Stirling's idea (Eq. 12) actually did define $n!$ for nonintegers n and that in fact Euler's and Stirling's generalizations were identical. Equation (13) is not extremely mysterious; with a little coaching (see exercise 22), the reader may discover it for himself.

Historically, many notations have been used for factorials. Euler actually wrote $[n]$, Gauss wrote $\pi(n)$, and the symbols $\lfloor\underline{n}$ and $\underline{n}\rfloor$ were used in England. The notation $n!$ which is universally used today (when n is an integer) was introduced by a comparatively little known mathematician, Christian Kramp, in an algebra text in 1808.

When n is *not* an integer, however, the notation $n!$ is very seldom used, and instead we customarily employ a notation due to A. M. Legendre:

$$n! = \Gamma(n+1) = n\Gamma(n). \tag{14}$$

The function $\Gamma(x)$ is called the *Gamma function*, and by Eq. (13) we have the definition

$$\Gamma(x) = \lim_{m\to\infty} \frac{m^x m!}{x(x+1)(x+2)\cdots(x+m)}. \tag{15}$$

A graph of this function is shown in Fig. 7.

The interesting history of factorials from the time of Stirling to the present day is traced in the article by P. J. Davis, "Leonhard Euler's Integral: A Historical Profile of the Gamma Function," *AMM* **66** (1959), 849–869.

EXERCISES

1. [*00*] How many ways are there to shuffle a 52-card deck?

2. [*10*] In the notation of Eq. (1), show that $p_{n(n-1)} = p_{nn}$, and explain why this happens.

3. [*10*] What permutations on 1, 2, 3, 4, 5 would be constructed from the permutation 3 1 2 4 using methods 1 and 2, respectively?

▶ **4.** [*13*] Given the fact that $\log_{10} 1000! = 2567.60464\ldots$, determine exactly how many decimal digits there are in the number 1000!. What is the *most significant* digit? What is the *least significant* digit?

5. [*15*] Approximate 8! using the following more exact version of Stirling's approximation:

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\frac{1}{12n}\right).$$

▶ **6.** [*17*] Using Eq. (8), write 20! as a product of prime factors.

7. [*M10*] Show that the "generalized termial" function in Eq. (10) satisfies the identity $x? = x + (x-1)?$ for all real numbers x.

8. [*HM15*] Show that the limit in Eq. (13) does equal $n!$ when n is a nonnegative integer.

9. [*M10*] Determine the values of $\Gamma(\frac{1}{2})$ and $\Gamma(-\frac{1}{2})$, given that $(\frac{1}{2})! = \sqrt{\pi}/2$.

▶ **10.** [*HM20*] Does the identity $\Gamma(x+1) = x\Gamma(x)$ hold for all real numbers x? (Cf. exercise 7.)

11. [*M15*] Let the representation of n in the binary system be $n = 2^{e_1} + 2^{e_2} + \cdots + 2^{e_r}$, where $e_1 > e_2 > \cdots > e_r \geq 0$. Show that $n!$ is divisible by 2^{n-r} but not by 2^{n-r+1}.

▶ **12.** [*M22*] (A. Legendre, 1808.) Generalizing the result of the previous exercise, let p be a prime number, and let the representation of n in the p-ary number system be $n = a_k p^k + a_{k-1}p^{k-1} + \cdots + a_1 p + a_0$. Express the number μ of Eq. (8) in a simple formula involving n, p, and the a's.

13. [*M23*] ("Wilson's theorem," actually due to Leibnitz, 1682.) If p is prime, $(p-1)! \bmod p = p - 1$. Prove this, by pairing off numbers among $1, 2, \ldots, p-1$ whose product mod p is 1.

▸ **14.** [*M28*] (L. Stickelberger, 1890.) In the notation of exercise 12, we can determine $n!$ mod p in terms of the p-ary representation, for *any* integer n, thus generalizing Wilson's theorem. In fact, prove that $n!/p^{\mu} \equiv (-1)^{\mu} a_0!\, a_1! \cdots a_k!$ (modulo p).

15. [*HM15*] The "permanent" of a square matrix is defined to be the same as the determinant except that each term in the expansion is given a plus sign instead of a minus sign. Thus, the permanent of

$$\begin{pmatrix} abc \\ def \\ ghi \end{pmatrix}$$

is $aei + bfg + cdh + gec + hfa + idb$. What is the permanent of

$$\begin{pmatrix} 1\times 1 & 1\times 2 & \dots & 1\times n \\ 2\times 1 & 2\times 2 & \dots & 2\times n \\ \vdots & & & \vdots \\ n\times 1 & n\times 2 & \dots & n\times n \end{pmatrix}?$$

16. [*HM15*] Show that the infinite sum in Eq. (11) does not converge unless n is a nonnegative integer.

17. [*HM20*] Prove that the infinite product

$$\prod_{n\geq 1} \frac{(n+\alpha_1)\cdots(n+\alpha_k)}{(n+\beta_1)\cdots(n+\beta_k)}$$

has the value $\Gamma(1+\beta_1)\cdots\Gamma(1+\beta_k)/\Gamma(1+\alpha_1)\cdots\Gamma(1+\alpha_k)$, if $\alpha_1+\cdots+\alpha_k = \beta_1+\cdots+\beta_k$ and if none of the β's is a negative integer.

18. [*M20*] Assume that $\pi/2 = \frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\cdots$. (This is "Wallis's product," obtained by J. Wallis in 1656, and we will prove it in exercise 1.2.6–43.) Using the previous exercise, prove that $(\frac{1}{2})! = \frac{1}{2}\sqrt{\pi}$.

19. [*HM22*] Denote the quantity appearing after "$\lim_{m\to\infty}$" in Eq. (15) by $\Gamma_m(x)$. Show that

$$\Gamma_m(x) = \int_0^m \left(1-\frac{t}{m}\right)^m t^{x-1}\,dt = m^x \int_0^1 (1-t)^m t^{x-1}\,dt, \qquad \text{if} \qquad x > 0.$$

20. [*HM21*] Using the fact that $0 \leq e^{-t} - (1-t/m)^m \leq t^2 e^{-t}/m$, if $0 \leq t \leq m$, and the previous exercise, show that $\Gamma(x) = \int_0^\infty e^{-t} t^{x-1}\,dt$, if $x > 0$.

21. [*HM25*] (Faa di Bruno's formula.) Let $D_x^k u$ represent the kth derivative of a function u with respect to x. The "chain rule" states that $D_x^1 w = D_u^1 w D_x^1 u$. If we apply this to second derivatives, we find $D_x^2 w = D_u^2 w (D_x^1 u)^2 + D_u^1 w D_x^2 u$. Show that the *general formula* is

$$D_x^n w = \sum_{0\leq j\leq n} \sum_{\substack{k_1+k_2+\cdots+k_n=j \\ k_1+2k_2+\cdots+nk_n=n \\ k_1,k_2,\ldots,k_n\geq 0}} D_u^j w \frac{n!}{k_1!(1!)^{k_1}\cdots k_n!(n!)^{k_n}} (D_x^1 u)^{k_1}\cdots(D_x^n u)^k.$$

▸ **22.** [*HM20*] Try to put yourself in Euler's place, looking for a way to generalize $n!$ to noninteger values of n. Since $(n+\frac{1}{2})!/n!$ times $((n+\frac{1}{2})+\frac{1}{2})!/(n+\frac{1}{2})!$ equals $(n+1)!/n! = n+1$, it seems natural that $(n+\frac{1}{2})!/n!$ should be approximately $\sqrt{n}$. Similarly, $(n+\frac{1}{3})!/n!$ should be approximately $\sqrt[3]{n}$. Invent a hypothesis about the ratio of $(n+x)!/n!$ as n approaches infinity. Is your hypothesis correct when x is an integer? Does it tell anything about the appropriate value of $x!$ when x is not an integer?

1.2.6. Binomial Coefficients

The *combinations of n objects taken k at a time* are the possible choices of k different elements from a collection of n objects. The combinations of the five objects $\{a, b, c, d, e\}$, taken three at a time, are

$$abc, \quad abd, \quad abe, \quad acd, \quad ace, \quad ade, \quad bcd, \quad bce, \quad bde, \quad cde. \tag{1}$$

It is a simple manner to count the total number of k-combinations of n objects: Equation (2) of the previous section told us that there are $n(n-1)\cdots(n-k+1)$ ways to choose the first k objects for a permutation; and every k-combination appears exactly $k!$ times in these arrangements, since each combination appears in all its permutations. Therefore the number of combinations, which we denote by $\binom{n}{k}$, is

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots(1)}. \tag{2}$$

For example,

$$\binom{5}{3} = \frac{5\cdot 4\cdot 3}{3\cdot 2\cdot 1} = 10,$$

which is the number of combinations we found in (1).

The quantity $\binom{n}{k}$ is called a *binomial coefficient*; these numbers have an extraordinary number of applications. They are probably the most important quantities entering into the analysis of algorithms, and so the reader is urged to become familiar with them.

Equation (2) may be used to define $\binom{n}{k}$ even when n is not an integer. We will now define the symbol $\binom{r}{k}$ for all real numbers r and all integers k:

$$\binom{r}{k} = \frac{r(r-1)\cdots(r-k+1)}{k(k-1)\cdots(1)} = \prod_{1\le j\le k}\left(\frac{r+1-j}{j}\right), \qquad \text{integer } k \ge 0; \tag{3}$$

$$\binom{r}{k} = 0, \qquad \text{integer } k < 0.$$

For particular cases we have

$$\binom{r}{0} = 1, \qquad \binom{r}{1} = r, \qquad \binom{r}{2} = \frac{r(r-1)}{2}. \tag{4}$$

Table 1 gives values of the binomial coefficients for small integer values of r and k. The values for $0 \leq r, k \leq 4$ should be memorized.

Table 1

TABLE OF BINOMIAL COEFFICIENTS (PASCAL'S TRIANGLE)

$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$	$\binom{n}{8}$
1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	2	1	0	0	0	0	0	0
1	3	3	1	0	0	0	0	0
1	4	6	4	1	0	0	0	0
1	5	10	10	5	1	0	0	0
1	6	15	20	15	6	1	0	0
1	7	21	35	35	21	7	1	0
1	8	28	56	70	56	28	8	1

The binomial coefficients have a long and interesting history. Table 1 is called "Pascal's triangle" because it appeared in Blaise Pascal's *Traité du triangle arithmétique* in 1653. This treatise was significant because it was one of the first works on probability theory, but Pascal did not invent the binomial coefficients (which were well-known in Europe at that time). Table 1 also appears in the treatise *Szu-yuen Yü-chien* ("The Precious Mirror of the Four Elements") by the Chinese mathematician Chu Shih-chieh in 1303, where they are said to be an old invention. The earliest known appearance of binomial coefficients is in a tenth century commentary, due to Halāyudha, on an ancient Hindu classic, the Chandaḥ-Sûtra. In about 1150 the Hindu mathematician Bháscara Áchárya gave a very clear exposition of binomial coefficients in his book *Líláváti*, Section 6, Chapter 4. For small values of k, they were known much earlier; they appeared in Greek and Roman writings with a geometric interpretation (cf. Fig. 8). The notation $\binom{r}{k}$ was introduced by Andreas von Ettingshausen in his book *Die Combinatorische Analysis* (Vienna, 1826).

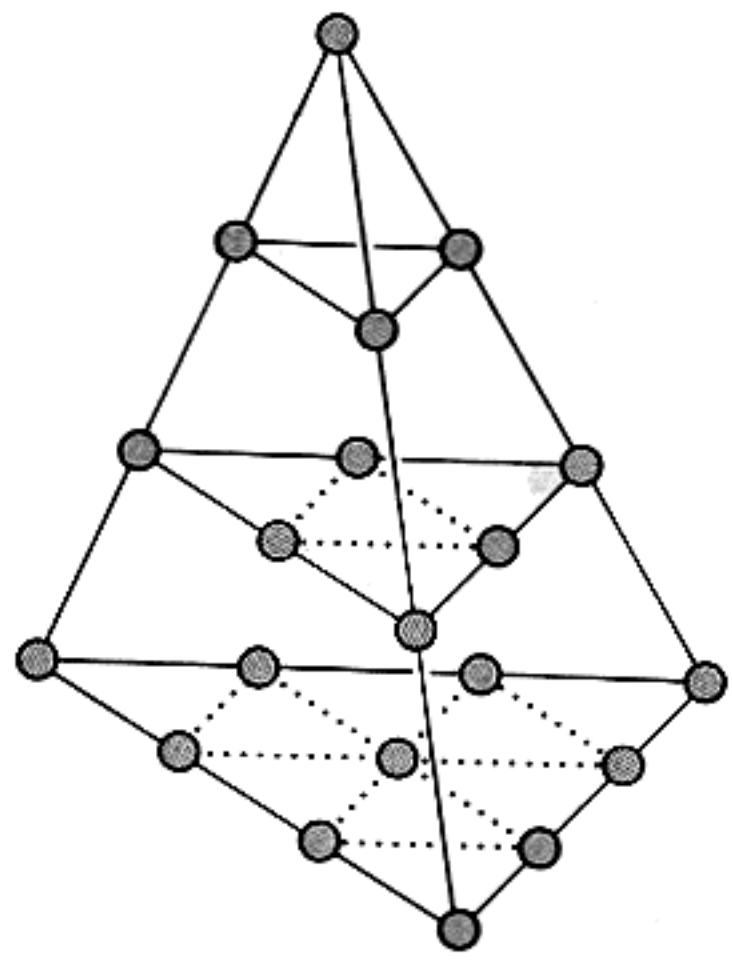


Fig. 8. Geometric interpretation of $\binom{n+2}{3}$, $n = 4$.

The reader has probably noticed several interesting patterns which appear in Table 1. There are literally thousands of identities involving binomial coefficients, and for centuries many people have been pleased to discover them.

However, there are so many relations present that when someone finds a new identity, there aren't many people who get excited about it any more, except the discoverer! In order to manipulate the formulas which arise in the analysis of algorithms, a facility for handling binomial coefficients is a must, and so an attempt has been made in this section to explain in a simple way how to maneuver with these numbers. Mark Twain once tried to reduce all jokes to a dozen or so primitive kinds (e.g., farmer's daughter, mother-in-law, etc.); we will try to condense the thousands of identities into a small set of basic operations with which we can solve nearly every problem involving these numbers that confronts us.

In most applications, *both* the numbers r and k which appear in $\binom{r}{k}$ will be integers. Some of the techniques we will describe are applicable only when both r and k are integers; so we will be careful to list, at the right of each numbered equation, any restrictions on the variables which appear. For example, in Eq. (3) we have mentioned the requirement that k is an integer; there is no restriction on r.

Now let us study the basic techniques for operating on binomial coefficients:

A. Representation by factorials. From Eq. (3) we have immediately

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \qquad \text{integer } n \geq \text{integer } k \geq 0. \tag{5}$$

This allows combinations of factorials to be represented as binomial coefficients and conversely.

B. Symmetry condition. From Eqs. (3) and (5), we have

$$\binom{n}{k} = \binom{n}{n-k}, \qquad \text{integer } n \geq 0, \quad \text{integer } k. \tag{6}$$

This formula holds for all integers k. *When k is negative or greater than n, the binomial coefficient is zero* (provided that n is a positive integer).

C. Moving in and out of brackets. From the definition (3), we have

$$\binom{r}{k} = \frac{r}{k}\binom{r-1}{k-1}, \qquad \text{integer } k \neq 0. \tag{7}$$

This formula is very useful for combining a binomial coefficient with other parts of an expression. By elementary transformation we have the rules

$$k\binom{r}{k} = r\binom{r-1}{k-1}, \qquad \frac{1}{r}\binom{r}{k} = \frac{1}{k}\binom{r-1}{k-1},$$

the first of which is valid for all integers k, and the second is valid when no division by zero has been performed. We also have a similar relation:

$$\binom{r}{k} = \frac{r}{r-k}\binom{r-1}{k}, \qquad \text{integer } k \neq r. \tag{8}$$

Let us illustrate these transformations, by proving Eq. (8) using Eqs. (6) and (7) alternately:

$$\binom{r}{k} = \binom{r}{r-k} = \frac{r}{r-k}\binom{r-1}{r-1-k} = \frac{r}{r-k}\binom{r-1}{k}.$$

[*Note:* This derivation is valid only when r is a positive integer $\neq k$, because of the constraints involved in Eqs. (6) and (7); yet Eq. (8) claims to be valid for *arbitrary* $r \neq k$. This can be proved in a simple and important manner: we have verified that

$$r\binom{r-1}{k} = (r-k)\binom{r}{k}$$

for *infinitely many values of* r. Both sides of this equation are *polynomials* in r. A nonzero polynomial of degree n can have at most n distinct zeros; so (by subtraction) *if two polynomials of degree* $\leq n$ *agree at* $n+1$ *or more different points, the polynomials are identically equal.* This principle may be used to extend the validity of many identities from integers to all real numbers.]

D. Addition formula. The basic relation

$$\binom{r}{k} = \binom{r-1}{k} + \binom{r-1}{k-1}, \qquad \text{integer } k, \tag{9}$$

is clearly valid in Table 1 (every value is the sum of the two values above and to the left) and we may easily verify it in general from Eq. (3). Alternatively, we have by Eqs. (7) and (8),

$$r\binom{r-1}{k} + r\binom{r-1}{k-1} = (r-k)\binom{r}{k} + k\binom{r}{k} = r\binom{r}{k}.$$

Equation (9) is often useful in obtaining proofs by induction on r, when r is an integer.

E. Summation formula. Applying Eq. (9) repeatedly, we obtain two important summation formulas:

$$\sum_{0\leq k\leq n}\binom{r+k}{k} = \binom{r}{0} + \binom{r+1}{1} + \cdots + \binom{r+n}{n} = \binom{r+n+1}{n},$$

$$\text{integer } n \geq 0. \tag{10}$$

$$\sum_{0\leq k\leq n}\binom{k}{m} = \binom{0}{m} + \binom{1}{m} + \cdots + \binom{n}{m} = \binom{n+1}{m+1},$$

$$\text{integer } m \geq 0,$$

$$\text{integer } n \geq 0. \tag{11}$$

Equation (11) can easily be proved by induction on n, but it is interesting to see how it can also be derived from Eq. (10) with two applications of Eq. (6):

$$\sum_{0\le k\le n}\binom{k}{m}=\sum_{-m\le k\le n-m}\binom{m+k}{m}=\sum_{-m\le k<0}\binom{m+k}{m}+\sum_{0\le k\le n-m}\binom{m+k}{k}$$

$$=0+\binom{m+(n-m)+1}{n-m}=\binom{n+1}{m+1},$$

assuming that $n\ge m$; and if $n<m$, Eq. (11) is obvious.

Equation (11) occurs very frequently in applications; in fact, we have already derived special cases of it in previous sections. For example, when $m=1$, we have

$$\binom{0}{1}+\binom{1}{1}+\cdots+\binom{n}{1}=0+1+\cdots+n=\binom{n+1}{2}=\frac{(n+1)n}{2},$$

our old friend, the sum of an arithmetic progression.

Suppose that we want the sum $1^2+2^2+\cdots+n^2$. This can be solved by observing that $k^2=2\binom{k}{2}+\binom{k}{1}$; hence

$$\sum_{0\le k\le n}k^2=\sum_{0\le k\le n}\left(2\binom{k}{2}+\binom{k}{1}\right)=2\binom{n+1}{3}+\binom{n+1}{2}.$$

If desired, this answer, obtained in terms of binomial coefficients, can be put back into polynomial notation:

$$1^2+2^2+\cdots+n^2=2\frac{(n+1)n(n-1)}{6}+\frac{(n+1)n}{2}$$

$$=\tfrac{1}{3}n(n+\tfrac{1}{2})(n+1). \tag{12}$$

The sum $1^3+2^3+\cdots+n^3$ can be obtained in a similar way; *any* polynomial $a_0+a_1k+a_2k^2+\cdots+a_mk^m$ can be expressed as $b_0\binom{k}{0}+b_1\binom{k}{1}+\cdots+b_m\binom{k}{m}$ for suitably chosen coefficients $b_0,\ldots,b_m$. We will return to this subject later.

F. The binomial theorem. Of course, the binomial theorem is one of our principal tools:

$$(x+y)^r=\sum_k\binom{r}{k}x^ky^{r-k},\qquad \text{integer } r\ge 0. \tag{13}$$

(At last we are able to justify the name "binomial coefficient" for our numbers.)

It is important to note that we have written "$\sum_k$" in Eq. (13), rather than "$\sum_{0\le k\le r}$" as might have been written. If no restriction is placed on k, we are summing over *all* integers, $-\infty<k<+\infty$; but the two notations are exactly

equivalent in this case, since when $k < 0$ or $k > r$, the terms in Eq. (13) are all zero. The simpler form "$\sum_k$" is to be preferred, since all manipulations with sums are simpler when the conditions of summation are simpler. We save a good deal of tedious effort if we do not need to keep track of the lower and/or upper limits of summation, so the limits should be left as infinity whenever possible. Our notation has another advantage also: If r is not a nonnegative integer, Eq. (13) becomes an *infinite* sum, and the *binomial theorem* of calculus states that *Eq.* (13) *is valid for all* r, *if* $|x/y| < 1$.

It should be noted that formula (13) gives

$$0^0 = 1, \tag{14}$$

and we will use this convention consistently.

The special case $y = 1$ in Eq. (13) is so important we state it specially:

$$\sum_k \binom{r}{k} x^k = (1+x)^r, \qquad \text{integer } r \geq 0, \quad \text{or} \quad |x| < 1. \tag{15}$$

The discovery of the binomial theorem was announced by Isaac Newton in a letter to Oldenburg on June 13, 1676. He apparently had no real proof of the formula (and at that time the necessity for rigorous proof was not fully realized). The first attempted proof was given by L. Euler in 1774, although that also was lacking in rigor; finally, K. F. Gauss gave the first actual proof in 1812. In fact, Gauss's work represented the first time *anything* about infinite sums was proved satisfactorily.

In the early nineteenth century, N. Abel found a surprising generalization of the binomial formula (Eq. 13):

$$(x+y)^r = \sum_k \binom{r}{k} x(x-kz)^{k-1}(y+kz)^{r-k}, \qquad \text{integer } r \geq 0, \quad x \neq 0, \tag{16}$$

which is an identity in *three* variables, x, y, and z (cf. exercises 50 through 52). Abel published and proved this formula in Volume 1 of the German *Journal für die reine und angewandte Mathematik* (1826), pp. 159–160. It is interesting to note that Abel contributed many other papers to the same Volume 1, including his famous memoirs on the unsolvability of algebraic equations of degree 5 or more, and on the binomial theorem. See *AMM* **69** (1962), 572 for a number of references to Eq. (16).

G. Negating the upper index. The basic identity

$$\binom{-r}{k} = (-1)^k \binom{r+k-1}{k}, \qquad \text{integer } k, \tag{17}$$

follows immediately from the definition (Eq. 3) when each term of the numerator is negated. This is often a useful transformation on the upper index.

We will give one example of the use of Eq. (17) here to prove the summation formula

$$\sum_{k\le n}\binom{r}{k}(-1)^k = \binom{r}{0} - \binom{r}{1} + \cdots + (-1)^n\binom{r}{n}$$
$$= (-1)^n\binom{r-1}{n}, \qquad \text{integer } n \ge 0. \tag{18}$$

This identity could be proved by induction using Eq. (9), but we can easily use Eqs. (17) and (10):

$$\sum_{k\le n}\binom{r}{k}(-1)^k = \sum_{k\le n}\binom{-r+k-1}{k} = \binom{-r+n}{n} = (-1)^n\binom{r-1}{n}.$$

An important application of Eq. (17) can be made when r is an integer:

$$\binom{n}{m} = (-1)^{n-m}\binom{-(m+1)}{n-m}, \qquad \text{integer } n \ge 0, \quad \text{integer } m. \tag{19}$$

[Take $n = -r$, $k = n - m$ in Eq. (17).] We have moved n from the upper position to the lower.

H. Simplifying products. When products of binomial coefficients appear, there are usually several different ways to reexpress the products by expanding into factorials and out again using Eq. (5). For example,

$$\binom{r}{m}\binom{m}{k} = \binom{r}{k}\binom{r-k}{m-k}, \qquad \text{integer } m, \quad \text{integer } k. \tag{20}$$

It suffices to prove Eq. (20) when r is an integer $\ge m$ [cf. the remarks after Eq. (8)], and when $0 \le k \le m$. Then

$$\binom{r}{m}\binom{m}{k} = \frac{r!m!}{m!(r-m)!k!(m-k)!}$$
$$= \frac{r!(r-k)!}{k!(r-k)!(m-k)!(r-m)!} = \binom{r}{k}\binom{r-k}{m-k}.$$

Equation (20) is very useful when an index (namely m) appears in both the upper and the lower position, and we wish to have it appear in one place rather than two. Note that Eq. (7) is the special case of Eq. (20) when $k = 1$.

I. Sums of products. To complete our set of binomial-coefficient manipulations, we present the following very general identities, which are proved in the exercises at the end of this section. These formulas show how to sum over a product of two binomial coefficients, considering various places where the

running variable k might appear:

$$\sum_k \binom{r}{k}\binom{s}{n-k} = \binom{r+s}{n}, \qquad \text{integer } n. \tag{21}$$

$$\sum_k \binom{r}{k}\binom{s}{n+k} = \binom{r+s}{r+n}, \qquad \text{integer } n, \quad \text{integer } r \geq 0. \tag{22}$$

$$\sum_k \binom{r}{k}\binom{s+k}{n}(-1)^k = (-1)^r \binom{s}{n-r},$$

$$\text{integer } n, \quad \text{integer } r \geq 0. \tag{23}$$

$$\sum_{0\leq k\leq r} \binom{r-k}{m}\binom{s}{k-t}(-1)^k = (-1)^t \binom{r-t-s}{r-t-m},$$

$$\text{integer } t \geq 0, \quad \text{integer } r \geq 0,$$
$$\text{integer } m \geq 0. \tag{24}$$

$$\sum_{0\leq k\leq r} \binom{r-k}{m}\binom{s+k}{n} = \binom{r+s+1}{m+n+1},$$

$$\text{integer } n \geq \text{integer } s \geq 0,$$
$$\text{integer } m \geq 0, \quad \text{integer } r \geq 0. \tag{25}$$

$$\sum_{k\geq 0} \binom{r-tk}{k}\binom{s-t(n-k)}{n-k}\frac{r}{r-tk} = \binom{r+s-tn}{n},$$

$$\text{integer } n,$$
$$r \neq tk \quad \text{for} \quad 0 \leq k \leq n. \tag{26}$$

Of these identities, Eq. (21) is by far the most important, and it should be memorized. Obviously Eq. (21) is a special case of Eq. (26) with $t = 0$. These formulas are the principal tools we have for working with difficult sums.

We should point out a nonobvious use of Eqs. (23) and (25); it is often helpful to replace the simple binomial coefficient on the right-hand side by the more complicated expression on the left, interchange the order of summation, and simplify. We may regard the left-hand sides as expansions of

$$\binom{s}{n+a} \quad \text{in terms of} \quad \binom{s+k}{n}.$$

Formula (23) is used for negative a, formula (25) for positive a.

This completes our study of "binomial-coefficientology." The reader is advised to learn especially Eqs. (5), (6), (7), (9), (13), (17), (20), and (21)—frame them in black!

With all these methods at our disposal, we should be able to solve "almost any" problem that comes along, in at least three different ways. The following examples illustrate the techniques.

Problem 1. When r is a positive integer, what is the value of

$$\sum_k \binom{r}{k}\binom{s}{k} k?$$

Solution. Formula (7) is useful for disposing of the outside k:

$$\sum_k \binom{r}{k}\binom{s}{k} k = \sum_k \binom{r}{k}\binom{s-1}{k-1} s$$

$$= s \sum_k \binom{r}{k}\binom{s-1}{k-1},$$

and now formula (22) applies, with $n = -1$. The answer is therefore

$$\sum_k \binom{r}{k}\binom{s}{k} k = \binom{r+s-1}{r-1} s, \qquad \text{integer } r \geq 0.$$

Problem 2. What is the value of

$$\sum_{k\geq 0} \binom{n+k}{2k}\binom{2k}{k} \frac{(-1)^k}{k+1},$$

if $n \geq 0$?

Solution. Now the problem is tougher; the summation index k appears in six places! First we apply Eq. (20), and we obtain

$$\sum_{k\geq 0} \binom{n+k}{k}\binom{n}{k} \frac{(-1)^k}{k+1}.$$

We can now breathe more easily, since several of the menacing characteristics of the original formula have now disappeared. The next step should be obvious; we apply Eq. (7) in a manner similar to the technique used in Problem 1:

$$\sum_{k\geq 0} \binom{n+k}{k}\binom{n+1}{k+1} \frac{(-1)^k}{n+1}, \tag{27}$$

and another k has disappeared. There are now two equally promising lines of attack. We can replace

$$\binom{n+k}{k} \quad \text{by} \quad \binom{n+k}{n},$$

and use Eq. (23):

$$\sum_{k\geq 0}\binom{n+k}{n}\binom{n+1}{k+1}\frac{(-1)^k}{n+1}$$

$$= -\frac{1}{n+1}\sum_{k\geq 1}\binom{n-1+k}{n}\binom{n+1}{k}(-1)^k$$

$$= -\frac{1}{n+1}\sum_{k\geq 0}\binom{n-1+k}{n}\binom{n+1}{k}(-1)^k + \frac{1}{n+1}\binom{n-1}{n}$$

$$= -\frac{1}{n+1}(-1)^{n+1}\binom{n-1}{-1} + \frac{1}{n+1}\binom{n-1}{n} = \frac{1}{n+1}\binom{n-1}{n}.$$

Now

$$\binom{n-1}{n}$$

equals zero except when $n = 0$, in which case it equals one.

It is convenient to represent the answer to our problem by using the "Kronecker delta" notation:

$$\delta_{ij} = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{if } i = j \end{cases}. \tag{28}$$

Using the δ-symbol, we have found that the answer is δ_{n0}.

Another way to proceed from Eq. (27) is to use Eq. (17), obtaining

$$\sum_{k}\binom{-(n+1)}{k}\binom{n+1}{k+1}\frac{1}{n+1}.$$

At this point Eq. (22) does not apply (since it requires that $r \geq 0$), but we can use Eq. (6) so that Eq. (21) applies:

$$\sum_{k}\binom{-(n+1)}{k}\binom{n+1}{n-k}\frac{1}{n+1} = \binom{0}{n}\frac{1}{n+1},$$

and once again we have derived the answer:

$$\sum_{k\geq 0}\binom{n+k}{2k}\binom{2k}{k}\frac{(-1)^k}{k+1} = \delta_{n0}, \qquad \text{integer } n \geq 0. \tag{29}$$

Problem 3. What is the value of

$$\sum_{k\geq 0}\binom{n+k}{m+2k}\binom{2k}{k}\frac{(-1)^k}{k+1}, \qquad \text{if} \qquad m, n > 0?$$

Solution. If m were zero, we would have the same formula to work with that we had in Problem 2. However, now the presence of m means that we cannot even begin to use the method of the previous solution, since the first step there was to use Eq. (20), which no longer applies. In this situation it pays to introduce *still further* complication by replacing

$$\binom{n+k}{m+2k}$$

by a sum of terms of the form

$$\binom{x+k}{2k},$$

since our problem then becomes a sum of problems we know how to solve! Accordingly, we use Eq. (25) with

$$r = n + k - 1, \qquad m = 2k, \qquad s = 0, \qquad n = m - 1,$$

and we have

$$\sum_{k\geq 0} \sum_{0\leq j\leq n+k-1} \binom{n+k-1-j}{2k}\binom{2k}{k}\binom{j}{m-1}\frac{(-1)^k}{k+1}. \tag{30}$$

We wish to perform the summation on k first; interchanging the order of summation demands that we sum on the values of k which are ≥ 0 *and* $\geq j - n + 1$. The latter condition raises problems, because if $j \geq n$, we do *not* know the desired sum. Let us save the situation, however, by observing that the terms of (30) are zero when $n \leq j \leq n + k - 1$. This condition implies that $k \geq 1$; thus $0 \leq n + k - 1 - j \leq k - 1 < 2k$, and the first binomial coefficient in (30) vanishes. We may therefore replace the condition on the second sum by "$0 \leq j < n$," and the interchange of summation is done easily. Summing on k by Eq. (29) now gives

$$\sum_{0\leq j<n} \binom{j}{m-1} \delta_{(n-1-j)0},$$

and all terms vanish except $j = n - 1$; hence our final answer is

$$\binom{n-1}{m-1}.$$

The solution to this problem was fairly complicated, but not really mysterious; there was a good reason for each step. The derivation should be studied closely because it illustrates some delicate maneuvering with the conditions in our equations. There is actually a better way to attack this problem, however; it is left for the reader to figure out a way to transform the given problem so that Eq. (26) applies (see exercise 30).

Problem 4. Prove that

$$\sum_k A_k(r, t)A_{n-k}(s, t) = A_n(r + s, t), \qquad \text{integer } n \geq 0,$$

where

$$A_n(x, t) = \binom{x - nt}{n} \frac{x}{x - nt}, \qquad (x \neq nt). \tag{31}$$

Solution. We must assume that the given sum is defined, that is, that $r \neq kt \neq s$ for $0 \leq k \leq n$. Our problem is to evaluate

$$\sum_k \binom{r - kt}{k}\binom{s - (n - k)t}{n - k} \frac{r}{r - kt} \frac{s}{s - (n - k)t},$$

which, if anything, looks much worse than our previous horrible problems! Note the strong similarity to Eq. (26), however, and also note the case $t = 0$.

We are tempted to change

$$\binom{r - kt}{k} \frac{r}{r - kt} \quad \text{to} \quad \binom{r - kt - 1}{k - 1} \frac{r}{k},$$

except that the latter tends to lose the analogy with Eq. (26) and it fails when $k = 0$. The best way to proceed is to use the technique of "partial fractions," i.e., a complicated denominator can often be replaced by a sum of simpler denominators. Indeed, we have

$$\frac{1}{r - kt} \frac{1}{s - (n - k)t} = \frac{1}{r + s - nt}\left(\frac{1}{r - kt} + \frac{1}{s - (n - k)t}\right).$$

Putting this into our sum we get

$$\frac{s}{r + s - nt} \sum_k \binom{r - kt}{k}\binom{s - (n - k)t}{n - k} \frac{r}{r - kt}$$

$$+ \frac{r}{r + s - nt} \sum_k \binom{r - kt}{k}\binom{s - (n - k)t}{n - k} \frac{s}{s - (n - k)t},$$

and Eq. (26) evaluates both of these if we change k to $(n - k)$ in the second formula; the desired result follows immediately. Identities (26) and (31) are due to H. A. Rothe, *Formulae de serierum reversione* (Leipzig, 1793); special cases of these formulas are still being "discovered" frequently. For the interesting history of these identities and some generalizations, see H. W. Gould and J. Kaucký, *Journal of Combinatorial Theory* **1** (1966), 233–248.

Problem 5. Determine the values of $a_0, a_1, a_2, \ldots$ such that

$$n! = a_0 + a_1 n + a_2 n(n - 1) + a_3 n(n - 1)(n - 2) + \cdots \tag{32}$$

for all nonnegative integers n.

Solution. This question came up in the previous section (cf. Eq. 1.2.5–11) and we stated the answer without proof. Let us pretend we do not know the answer. It is clear that the problem *has* a solution, since we can set $n = 0$ and determine a_0, then set $n = 1$ and determine a_1, etc.

First we would like to write Eq. (32) in terms of binomial coefficients:

$$n! = \sum_k \binom{n}{k} k! a_k. \tag{33}$$

The problem of solving implicit equations like this for a_k is called the *inversion problem*, and the technique to be used applies to similar problems as well.

The idea is based on the following special case of Eq. (23) ($s = 0$):

$$\sum_k \binom{r}{k}\binom{k}{n} (-1)^k = (-1)^r \binom{0}{n-r} = (-1)^r \delta_{nr},$$

$$\text{integer } n, \quad \text{integer } r \geq 0. \tag{34}$$

The importance of this formula is that when $n \neq r$, the sum is zero; this enables us to solve our problem (much like we did in Problem 3) as follows:

$$\begin{aligned}\sum_n n! \binom{m}{n} (-1)^n &= \sum_n \sum_k \binom{n}{k} k! a_k \binom{m}{n} (-1)^n \\ &= \sum_k k! a_k \sum_n \binom{n}{k}\binom{m}{n} (-1)^n \\ &= \sum_k k! a_k (-1)^m \delta_{km} = (-1)^m m! a_m.\end{aligned}$$

Note how we were able to get an equation in which only one value a_m appears—by adding together suitable multiples of Eq. (33) for $n = 0, 1, 2, \ldots$. We have now

$$a_m = \sum_{n \geq 0} (-1)^{m+n} \frac{n!}{m!} \binom{m}{n} = \sum_{0 \leq n \leq m} \frac{(-1)^{m+n}}{(m-n)!} = \sum_{0 \leq n \leq m} \frac{(-1)^n}{n!}.$$

This completes the solution to Problem 5. Let us now take a closer look at the implications of Eq. (34): we have

$$\sum_k \binom{r}{k} (-1)^k \left(c_0 \binom{k}{0} + c_1 \binom{k}{1} + \cdots + c_r \binom{k}{r} \right) = (-1)^r c_r,$$

since the first terms vanish after summation. By properly choosing the coefficients c_i, we can represent *any* polynomial in k as a sum of binomial coefficients with upper index k. We therefore find that

$$\sum_k \binom{r}{k} (-1)^k (b_0 + b_1 k + \cdots + b_r k^r) = (-1)^r r! b_r, \qquad \text{integer } r \geq 0, \tag{35}$$

where $b_0 + \cdots + b_r k^r$ represents any polynomial whatever of degree r or less. [This formula will be of no great surprise to students of numerical analysis, since $\sum_k \binom{r}{k}(-1)^{r+k} f(x+k)$ is the "rth difference" of the function $f(x)$.]

Using Eq. (35), we can immediately obtain many other relations which appear complicated at first and which are often given very lengthy proofs, e.g.,

$$\sum_k \binom{r}{k}\binom{s-kt}{r}(-1)^k = t^r, \qquad \text{integer } r \geq 0. \tag{36}$$

It is customary in textbooks such as this to give a lot of impressive examples of neat tricks, etc., but to never mention simple-looking problems where the techniques fail. The above examples may have given the impression that all things are possible with binomial coefficients; it should be mentioned, however, that in spite of Eqs. (10), (11), and (18), there seems to be no simple formula for the analogous sum

$$\sum_{0 \leq k \leq n} \binom{m}{k} = \binom{m}{0} + \binom{m}{1} + \cdots + \binom{m}{n},$$

when $n < m$. (For $n = m$ the answer is simple; what is it? See exercise 36.)

There are several generalizations of the concept of binomial coefficients, which we will discuss briefly. First, we can consider arbitrary real values of the lower index k in $\binom{r}{k}$; see exercises 40 through 45. We also have the generalization

$$\binom{r}{k}_q = \frac{(1-q^r)(1-q^{r-1})\ldots(1-q^{r-k+1})}{(1-q^k)(1-q^{k-1})\ldots(1-q^1)}, \tag{37}$$

which, as q approaches the limiting value one, becomes the ordinary binomial coefficient $\binom{r}{k}_1 = \binom{r}{k}$. [This can be seen by dividing each term in numerator and denominator by $(1-q)$.]

However, for our purposes the most important generalization is the *multinomial coefficient*

$$\binom{k_1 + k_2 + \cdots + k_m}{k_1, k_2, \ldots, k_m} = \frac{(k_1 + k_2 + \cdots + k_m)!}{k_1! k_2! \cdots k_m!}, \qquad \text{integer } k_i \geq 0. \tag{38}$$

The principal property of multinomial coefficients is the generalization of Eq. (13):

$$(x_1 + x_2 + \cdots + x_m)^n = \sum_{k_1+k_2+\cdots+k_m=n} \binom{n}{k_1, k_2, \ldots, k_m} x_1^{k_1} x_2^{k_2} \cdots x_m^{k_m}. \tag{39}$$

It is important to observe that any multinomial coefficient can be expressed in terms of binomial coefficients:

$$\binom{k_1+k_2+\cdots+k_m}{k_1, k_2, \ldots, k_m} = \binom{k_1+k_2}{k_1}\binom{k_1+k_2+k_3}{k_1+k_2}\cdots\binom{k_1+k_2+\cdots+k_m}{k_1+\cdots+k_{m-1}},$$

so we may apply the techniques we already know for manipulating binomial coefficients.

We conclude this section with a brief analysis of the transformation from a polynomial 'expressed in powers of k to a polynomial expressed in binomial coefficients. The coefficients involved in this transformation are called *Stirling numbers*, and these numbers will arise several times in later sections of this book.

Stirling numbers come in two flavors: we denote Stirling numbers of the first kind by $\left[{n \atop k}\right]$, and those of the second kind by $\left\{{n \atop k}\right\}$. Table 2 displays "Stirling's triangles," which are in some ways analogous to Pascal's triangle.

There is absolutely no agreement today on notation for Stirling's numbers. Some authors define half of the Stirling numbers to be the negatives of the values given here. However, the notation used here, in which all Stirling numbers are nonnegative, makes it much easier to remember the analogies with binomial coefficients.

Stirling numbers of the first kind are used to convert from binomial coefficients to powers:

$$\begin{aligned} n!\binom{x}{n} &= x(x-1)\cdots(x-n+1) \\ &= \begin{bmatrix} n \\ n \end{bmatrix} x^n - \begin{bmatrix} n \\ n-1 \end{bmatrix} x^{n-1} + \cdots + (-1)^n \begin{bmatrix} n \\ 0 \end{bmatrix} \\ &= \sum_k (-1)^{n-k} \begin{bmatrix} n \\ k \end{bmatrix} x^k. \end{aligned} \tag{40}$$

For example, from Table 2,

$$\binom{x}{5} = \tfrac{1}{120}(x^5 - 10x^4 + 35x^3 - 50x^2 + 24x).$$

Stirling numbers of the second kind are used to convert from powers to binomial coefficients:

$$x^n = \left\{{n \atop n}\right\}\binom{x}{n} n! + \cdots + \left\{{n \atop 1}\right\}\binom{x}{1} 1! + \left\{{n \atop 0}\right\}\binom{x}{0} 0! = \sum_k \left\{{n \atop k}\right\}\binom{x}{k} k!. \tag{41}$$

For example, from Table 2,

$$\begin{aligned} x^5 &= \binom{x}{5} 5! + 10\binom{x}{4} 4! + 25\binom{x}{3} 3! + 15\binom{x}{2} 2! + \binom{x}{1} 1! \\ &= 120\binom{x}{5} + 240\binom{x}{4} + 150\binom{x}{3} + 30\binom{x}{2} + \binom{x}{1}. \end{aligned}$$

We shall now list the most important identities involving Stirling numbers. (In these equations, the variables m and n always denote nonnegative integers.)

Table 2

STIRLING NUMBERS OF THE FIRST AND SECOND KINDS*

$\begin{bmatrix} n \\ 0 \end{bmatrix}$	$\begin{bmatrix} n \\ 1 \end{bmatrix}$	$\begin{bmatrix} n \\ 2 \end{bmatrix}$	$\begin{bmatrix} n \\ 3 \end{bmatrix}$	$\begin{bmatrix} n \\ 4 \end{bmatrix}$	$\begin{bmatrix} n \\ 5 \end{bmatrix}$	$\begin{bmatrix} n \\ 6 \end{bmatrix}$	$\begin{bmatrix} n \\ 7 \end{bmatrix}$	$\begin{bmatrix} n \\ 8 \end{bmatrix}$	n	$\begin{Bmatrix} n \\ 0 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 1 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 2 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 3 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 4 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 5 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 6 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 7 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 8 \end{Bmatrix}$
1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	2	0	1	1	0	0	0	0	0	0
0	2	3	1	0	0	0	0	0	3	0	1	3	1	0	0	0	0	0
0	6	11	6	1	0	0	0	0	4	0	1	7	6	1	0	0	0	0
0	24	50	35	10	1	0	0	0	5	0	1	15	25	10	1	0	0	0
0	120	274	225	85	15	1	0	0	6	0	1	31	90	65	15	1	0	0
0	720	1764	1624	735	175	21	1	0	7	0	1	63	301	350	140	21	1	0
0	5040	13068	13132	6769	1960	322	28	1	8	0	1	127	966	1701	1050	266	28	1

* For further values, see *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun, U.S. Government Printing Office, 1964, Tables 24.3 and 24.4, where $\begin{bmatrix} n \\ m \end{bmatrix}$ is denoted by $(-1)^{n+m} S_n^{(m)}$ and $\begin{Bmatrix} n \\ m \end{Bmatrix}$ is denoted by $\mathfrak{S}_n^{(m)}$.

Table 2

STIRLING NUMBERS OF THE FIRST AND SECOND KINDS*

$\begin{bmatrix} n \\ 0 \end{bmatrix}$	$\begin{bmatrix} n \\ 1 \end{bmatrix}$	$\begin{bmatrix} n \\ 2 \end{bmatrix}$	$\begin{bmatrix} n \\ 3 \end{bmatrix}$	$\begin{bmatrix} n \\ 4 \end{bmatrix}$	$\begin{bmatrix} n \\ 5 \end{bmatrix}$	$\begin{bmatrix} n \\ 6 \end{bmatrix}$	$\begin{bmatrix} n \\ 7 \end{bmatrix}$	$\begin{bmatrix} n \\ 8 \end{bmatrix}$	n	$\begin{Bmatrix} n \\ 0 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 1 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 2 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 3 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 4 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 5 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 6 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 7 \end{Bmatrix}$	$\begin{Bmatrix} n \\ 8 \end{Bmatrix}$
1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	2	0	1	1	0	0	0	0	0	0
0	2	3	1	0	0	0	0	0	3	0	1	3	1	0	0	0	0	0
0	6	11	6	1	0	0	0	0	4	0	1	7	6	1	0	0	0	0
0	24	50	35	10	1	0	0	0	5	0	1	15	25	10	1	0	0	0
0	120	274	225	85	15	1	0	0	6	0	1	31	90	65	15	1	0	0
0	720	1764	1624	735	175	21	1	0	7	0	1	63	301	350	140	21	1	0
0	5040	13068	13132	6769	1960	322	28	1	8	0	1	127	966	1701	1050	266	28	1

* For further values, see *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun, U.S. Government Printing Office, 1964, Tables 24.3 and 24.4, where $\begin{bmatrix} n \\ m \end{bmatrix}$ is denoted by $(-1)^{n+m} S_n^{(m)}$ and $\begin{Bmatrix} n \\ m \end{Bmatrix}$ is denoted by $\mathfrak{S}_n^{(m)}$.

Addition formulas:

$$\begin{bmatrix} n \\ m \end{bmatrix} = (n-1)\begin{bmatrix} n-1 \\ m \end{bmatrix} + \begin{bmatrix} n-1 \\ m-1 \end{bmatrix},$$
$$\begin{Bmatrix} n \\ m \end{Bmatrix} = m\begin{Bmatrix} n-1 \\ m \end{Bmatrix} + \begin{Bmatrix} n-1 \\ m-1 \end{Bmatrix}, \qquad \text{if} \quad n > 0. \tag{42}$$

Inversion formulas (compare with Eq. 34):

$$\sum_k \begin{bmatrix} n \\ k \end{bmatrix}\begin{Bmatrix} k \\ m \end{Bmatrix}(-1)^k = (-1)^n\delta_{mn}, \qquad \sum_k \begin{Bmatrix} n \\ k \end{Bmatrix}\begin{bmatrix} k \\ m \end{bmatrix}(-1)^k = (-1)^n\delta_{mn}. \tag{43}$$

Special values: $$\binom{0}{n} = \begin{bmatrix} 0 \\ n \end{bmatrix} = \begin{Bmatrix} 0 \\ n \end{Bmatrix}, \qquad \binom{n}{n} = \begin{bmatrix} n \\ n \end{bmatrix} = \begin{Bmatrix} n \\ n \end{Bmatrix} = 1; \tag{44}$$

$$\begin{bmatrix} n \\ n-1 \end{bmatrix} = \begin{Bmatrix} n \\ n-1 \end{Bmatrix} = \binom{n}{2}; \tag{45}$$

$$\begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{Bmatrix} n \\ 0 \end{Bmatrix} = 0, \qquad \begin{bmatrix} n \\ 1 \end{bmatrix} = (n-1)!, \qquad \begin{Bmatrix} n \\ 1 \end{Bmatrix} = 1,$$
$$\begin{Bmatrix} n \\ 2 \end{Bmatrix} = 2^{n-1} - 1, \qquad \text{if} \quad n > 0. \tag{46}$$

Expansion formulas:

$$\sum_k \begin{bmatrix} n \\ k \end{bmatrix}\binom{k}{m} = \begin{bmatrix} n+1 \\ m+1 \end{bmatrix}, \qquad \sum_k \begin{bmatrix} n+1 \\ k+1 \end{bmatrix}\binom{k}{m}(-1)^k = \begin{bmatrix} n \\ m \end{bmatrix}(-1)^m; \tag{47}$$

$$\sum_k \begin{Bmatrix} k \\ m \end{Bmatrix}\binom{n}{k} = \begin{Bmatrix} n+1 \\ m+1 \end{Bmatrix}, \qquad \sum_k \begin{Bmatrix} k+1 \\ m+1 \end{Bmatrix}\binom{n}{k}(-1)^k = \begin{Bmatrix} n \\ m \end{Bmatrix}(-1)^n; \tag{48}$$

$$\sum_k \binom{n}{k} k^m (-1)^k = (-1)^n n! \begin{Bmatrix} m \\ n \end{Bmatrix}; \tag{49}$$

$$\sum_k \binom{m-n}{m+k}\binom{m+n}{n+k}\begin{Bmatrix} m+k \\ k \end{Bmatrix} = \begin{bmatrix} n \\ n-m \end{bmatrix},$$
$$\text{if} \quad n \geq m; \tag{50}$$
$$\sum_k \binom{m-n}{m+k}\binom{m+n}{n+k}\begin{bmatrix} m+k \\ k \end{bmatrix} = \begin{Bmatrix} n \\ n-m \end{Bmatrix},$$

$$\sum_k \begin{Bmatrix} n+1 \\ k+1 \end{Bmatrix}\begin{bmatrix} k \\ m \end{bmatrix}(-1)^k = (-1)^m\binom{n}{m}; \tag{51}$$

$$\sum_{k\leq n} \begin{bmatrix} k \\ m \end{bmatrix}\frac{n!}{k!} = \begin{bmatrix} n+1 \\ m+1 \end{bmatrix}, \qquad \sum_{k\leq n} \begin{Bmatrix} k \\ m \end{Bmatrix}(m+1)^{n-k} = \begin{Bmatrix} n+1 \\ m+1 \end{Bmatrix}. \tag{52}$$

Some other fundamental Stirling number identities appear in exercises 1.2.6–61, 1.2.7–6, and in Eqs. (23), (26), (27), and (28) of Section 1.2.9. For further information on Stirling numbers, see Karoly (Charles) Jordan, *Calculus of Finite Differences* (New York: Chelsea, 1947), Chapter 4.

EXERCISES

1. [*00*] How many combinations of n things taken $n - 1$ at a time are possible?

2. [*00*] What is $\binom{0}{0}$?

3. [*00*] How many bridge hands are possible (i.e., 13 cards out of a 52-card deck)?

4. [*10*] Give the answer to Problem 3 as a product of prime numbers.

▶ **5.** [*05*] Explain the fact that $11^4 = 14641$ in terms of Pascal's triangle.

▶ **6.** [*10*] Pascal's triangle (Table 1) can be extended in all directions by use of the addition formula, Eq. (9). Find the three rows which go on *top* of Table 1 (i.e., for $n = -1, -2$, and -3).

7. [*12*] If n is a fixed positive integer, what value of k makes $\binom{n}{k}$ a maximum?

8. [*00*] What property of Pascal's triangle is reflected in the "symmetry condition," Eq. (6)?

9. [*01*] What is the value of $\binom{n}{n}$? (Consider all integers n.)

▶ **10.** [*M25*] If p is prime, show that:

a) $\binom{n}{p} \equiv \left\lfloor \frac{n}{p} \right\rfloor$ (modulo p).

b) $\binom{p}{k} \equiv 0$ (modulo p), for $1 \le k \le p - 1$.

c) $\binom{p-1}{k} \equiv (-1)^k$ (modulo p), for $0 \le k \le p - 1$.

d) $\binom{p+1}{k} \equiv 0$ (modulo p), for $2 \le k \le p - 1$.

e) (E. Lucas, 1877)

$$\binom{n}{k} \equiv \binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor}\binom{n \bmod p}{k \bmod p} \quad \text{(modulo } p\text{)}.$$

f) If the p-ary number system representations of n, k are

$$n = a_r p^r + \cdots + a_1 p + a_0, \qquad k = b_r p^r + \cdots + b_1 p + b_0,$$

then $\binom{n}{k} \equiv \binom{a_r}{b_r} \cdots \binom{a_1}{b_1}\binom{a_0}{b_0}$ (modulo p).

▶ **11.** [*M20*] (E. Kummer, 1852.) Show that the highest power to which a prime p divides

$$\binom{a+b}{a}$$

is equal to the number of *carries* which occur when a is added to b in the p-ary number system. (Cf. exercise 1.2.5–12.)

12. [*M22*] Are there any positive integers n for which all the nonzero entries in the nth row of Pascal's triangle are *odd*? If so, find all such n.

13. [*M13*] Prove the summation formula Eq. (10).

14. [*M21*] Evaluate $\sum_{0\le k\le n} k^4$.

15. [*M15*] Prove the binomial formula, Eq. (13).

16. [*M15*] Given that n, k are positive integers, show that

$$(-1)^n\binom{-n}{k-1} = (-1)^k\binom{-k}{n-1}.$$

▸ **17.** [*M18*] Prove the basic identity, Eq. (21), from Eq. (13), using the idea that $(1+x)^{r+s} = (1+x)^r(1+x)^s$.

18. [*M15*] Prove Eq. (22) using Eqs. (21) and (6).

19. [*M18*] Prove Eq. (23) by induction.

20. [*M20*] Prove Eq. (24) by using Eqs. (21) and (19), then show that another use of Eq. (19) yields Eq. (25).

▸ **21.** [*M05*] Both sides of Eq. (25) are polynomials in s; why isn't that equation an identity in s?

22. [*M20*] Prove Eq. (26) for the special case $s = n - 1 - r + nt$.

23. [*M13*] Assuming that Eq. (26) holds for (r, s, t, n) and $(r, s-t, t, n-1)$, prove it for $(r, s+1, t, n)$.

24. [*M15*] Explain why the results of the previous two exercises combine to give a proof of Eq. (26).

25. [*HM30*] Let the notation $A_n(x, t)$ have the significance of Eq. (31). Let $z = x^{t+1} - x^t$, and assume that $r \neq kt$ for integer $k \geq 0$. Prove that $\sum_k A_k(r, t)z^k = x^r$, provided z is small enough. [*Note:* If $t = 0$, this result is essentially the binomial theorem, and this equation is an important generalization of the binomial theorem. The binomial theorem (Eq. 15) may be assumed in the proof.] *Hint:* Start with the identity

$$\sum_j (-1)^j\binom{k}{j}\binom{r-jt}{k}\frac{r}{r-jt} = \delta_{k0}.$$

26. [*HM25*] Using the assumptions of the previous exercise, prove that

$$\sum_k \binom{r-tk}{k} z^k = \frac{x^{r+1}}{(t+1)x - t}.$$

27. [*HM20*] Solve Problem 4 in the text by using the result of exercise 25; and prove Eq. (26) from the preceding two exercises.

28. [*M25*] Prove that

$$\sum_k \binom{r+tk}{k}\binom{s-tk}{n-k} = \sum_{k\ge 0}\binom{r+s-k}{n-k} t^k,$$

if n is a nonnegative integer.

29. [*M20*] Show that Eq. (35) is just a very special case of the general identity proved in exercise 1.2.3–33.

▶ **30.** [*M24*] Show that there is a better way to solve Problem 3 than the way used in the text, by manipulating the sum so that Eq. (26) applies.

▶ **31.** [*M20*] Evaluate

$$\sum_k \binom{m-r+s}{k}\binom{n+r-s}{n-k}\binom{r+k}{m+n}$$

in terms of r, s, m, and n, given that m and n are nonnegative integers. Begin by replacing

$$\binom{r+k}{m+n} \quad \text{by} \quad \sum_j \binom{r}{m+n-j}\binom{k}{j}.$$

32. [*M20*] Let the notation $x^{\overline{n}}$ stand for $x(x+1)\cdots(x+n-1)$. Show that $\sum_k \left[{n \atop k}\right] x^k = x^{\overline{n}}$.

33. [*M20*] Using the notation of the previous exercise, show that the binomial formula is valid when it involves the modified "powers" of x as well as ordinary powers; i.e., show that $(x+y)^{\overline{n}} = \sum_k \binom{n}{k} x^{\overline{k}} y^{\overline{n-k}}$.

34. [*M23*] (Torelli's sum.) In the light of the previous exercise show that Abel's generalization of the binomial formula is also true for modified "powers":

$$(x+y)^{\overline{n}} = \sum_k \binom{n}{k} x(x-kz+1)^{\overline{k-1}}(y+kz)^{\overline{n-k}}. \qquad \text{(Cf. Eq. 16.)}$$

35. [*M23*] Prove the addition formulas, Eq. (42), for Stirling numbers directly from the definitions, Eqs. (40) and (41).

36. [*M10*] What is the sum $\sum_k \binom{n}{k}$ of the numbers in each row of Pascal's triangle? What is the sum of these numbers with alternating signs, $\sum_k \binom{n}{k}(-1)^k$?

37. [*M10*] From the answers to the preceding exercise, deduce the value of the sum of every other entry in a row, $\binom{n}{0} + \binom{n}{2} + \binom{n}{4} + \cdots$.

38. [*HM30*] (C. Ramus, 1834.) Generalizing the result of the preceding exercise, show that we have the following formula, given that $0 \le k < m$:

$$\binom{n}{k} + \binom{n}{m+k} + \binom{n}{2m+k} + \cdots = \frac{1}{m} \sum_{0 \le j < m} \left(2\cos\frac{j\pi}{m}\right)^n \cos\frac{j(n-2k)\pi}{m}.$$

For example,

$$\binom{n}{1} + \binom{n}{4} + \binom{n}{7} + \cdots = \frac{1}{3}\left(2^n + 2\cos\frac{(n-2)\pi}{3}\right).$$

[*Hint:* Find the right combinations of these coefficients multiplied by mth roots of unity.]

39. [*M10*] What is the sum $\sum_k \left[{n \atop k}\right]$ of the numbers in each row of Stirling's first triangle? What is the sum of these numbers with alternating signs? (Cf. exercise 36.)

40. [*HM17*] The *Beta function* $B(x, y)$ is defined for positive real numbers x, y by the formula $B(x, y) = \int_0^1 t^{x-1}(1-t)^{y-1}\,dt$.

a) Show that $B(x, 1) = B(1, x) = 1/x$.
b) Show that $B(x+1, y) + B(x, y+1) = B(x, y)$.
c) Show that $B(x, y) = \big((x+y)\big)/yB(x, y+1)$.

41. [*HM22*] We showed a relation between the Gamma function and the Beta function in exercise 1.2.5–19, by showing that $\Gamma_m(x) = m^x B(x, m+1)$, if m is a positive integer.

a) Prove that

$$B(x, y) = \frac{\Gamma_m(y)m^x}{\Gamma_m(x+y)}\,B(x, y+m+1).$$

b) Show that

$$B(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}.$$

42. [*HM10*] Express the binomial coefficient $\binom{r}{k}$ in terms of the Beta function defined above. (This gives us a way to extend the definition to all real values of k.)

43. [*HM20*] Show that $B(\frac{1}{2}, \frac{1}{2}) = \pi$. (From exercise 41 we may now conclude that $\Gamma(\frac{1}{2}) = \sqrt{\pi}$.)

44. [*HM2*] Using the generalized binomial coefficient suggested in exercise 42, show that

$$\binom{r}{1/2} = 2^{2r+1} \Big/ \binom{2r}{r}\pi.$$

45. [*HM21*] Using the generalized binomial coefficient suggested in exercise 42, find $\lim_{r\to\infty} \binom{r}{k}/r^k$.

▸ **46.** [*M21*] Using Stirling's approximation (Eq. 1.2.5–7), find an approximate value of

$$\binom{x+y}{y},$$

assuming that both x and y are large. In particular, find the approximate size of $\binom{2n}{n}$ when n is large.

47. [*M21*] Given that k is an integer, show that

$$\binom{n}{k}\binom{n+\frac{1}{2}}{k} = \binom{2n+1}{k}\binom{2n+1-k}{k} \Big/ 4^k.$$

Give a simpler formula for the special case $n = -1$.

▸ **48.** [*M25*] Show that

$$\sum_{k\geq 0}\binom{n}{k}\frac{(-1)^k}{k+x} = \frac{n!}{x(x+1)\cdots(x+n)} = \frac{1}{x\binom{n+x}{n}},$$

if the denominators are not zero. [Note that this formula gives us the reciprocal of a binomial coefficient, as well as the partial fraction expansion of $1/x(x+1)\cdots(x+n)$.]

49. [*M20*] Show that the identity $(1+x)^r = (1-x^2)^r(1-x)^{-r}$ implies a relation on binomial coefficients.

50. [*M20*] Prove Abel's formula, Eq. (16), in the special case $x+y=0$.

51. [*M21*] Prove Abel's formula, Eq. (16), by writing $y=(x+y)-x$, expanding the right-hand side in powers of $(x+y)$, and applying the result of the previous exercise.

52. [*HM50*] For what complex values of x, y, z, and r is Abel's generalized binomial formula, Eq. (16), valid?

53. [*M25*] (a) Prove the following identity by induction on m:

$$\sum_{0\le k\le m}\binom{r}{k}\binom{s}{n-k}\Big(nr-(r+s)k\Big)=(m+1)(n-m)\binom{r}{m+1}\binom{s}{n-m},$$

integer m, n.

(b) Making use of the important relations

$$\binom{-1/2}{n}=\frac{(-1)^n}{2^{2n}}\binom{2n}{n},\quad \binom{1/2}{n}=\frac{(-1)^{n-1}}{2^{2n}(2n-1)}\binom{2n}{n}=\frac{(-1)^{n-1}}{2^{2n-1}(2n-1)}\binom{2n-1}{n}$$

show that the following formula can be obtained as a special case of the identity in part (a):

$$\sum_{0\le k\le m}\binom{2k-1}{k}\binom{2n-2k}{n-k}\frac{-1}{2k-1}=\frac{n-m}{2n}\binom{2m}{m}\binom{2n-2m}{n-m}+\frac{1}{2}\binom{2n}{n}.$$

(This result is considerably more general than Eq. (26) in the case $r=-1$, $s=0$, $t=-2$.)

54. [*HM21*] Consider Pascal's triangle (as shown in Table 1) as a matrix. What is the *inverse* of that matrix?

55. [*M21*] Considering each of Stirling's triangles (Table 2) as matrices, determine their inverses.

▸ **56.** [*20*] (The "binomial number system.") For each integer $n=0, 1, 2, \ldots, 20$, find three integers a, b, c for which $n=\binom{a}{1}+\binom{b}{2}+\binom{c}{3}$ and $0\le a<b<c$. Can you see how this can be continued for higher values of n?

▸ **57.** [*M22*] Show that the coefficient a_m in Stirling's attempt at generalizing the factorial function (Eq. 1.2.5–12) is

$$\frac{(-1)^m}{m!}\sum_{k\ge 1}(-1)^k\binom{m-1}{k-1}\ln k.$$

58. [*M21*] In the notation of Eq. (37), prove the "q-binomial theorem":

$$(1+x)(1+qx)\cdots(1+q^{n-1}x)=\sum_k\binom{n}{k}_q q^{k(k-1)/2}x^k.$$

59. [*M25*] A sequence of numbers A_{nk}, $n\ge 0$, $k\ge 0$, satisfies the relations $A_{n0}=1$, $A_{0k}=\delta_{0k}$, $A_{nk}=A_{(n-1)k}+A_{n(k-1)}+\binom{n}{k}$. Find A_{nk}.

▸ **60.** [*24*] We have seen that $\binom{n}{k}$ is the number of combinations of n things, k at a time, i.e., the number of ways to choose k different things out of a set of n. The *combinations with repetitions* are similar to ordinary combinations, except we may choose each object

any number of times. Thus, the list (1) would be extended to include also *aaa*, *aab*, *aac*, *aad*, *aae*, *abb*, etc., if we were considering combinations with repetition. How many k-combinations of n objects are there, if repetition is allowed?

61. [*M25*] Evaluate the sum

$$\sum_k \begin{bmatrix} n+1 \\ k+1 \end{bmatrix} \begin{Bmatrix} k \\ m \end{Bmatrix} (-1)^k,$$

thereby obtaining a companion formula for Eq. (51).

62. [*M38*] The text gives formulas for sums involving a product of two binomial coefficients. Of the sums involving a product of three binomial coefficients, the following seems to be most useful:

$$\sum_k (-1)^k \binom{2l}{l+k}\binom{2m}{m+k}\binom{2n}{n+k} = \frac{(l+m+n)!(2l)!(2m)!(2n)!}{(l+m)!(m+n)!(n+l)!l!m!n!},$$

integer $l, m, n \geq 0$.

(Note that the sum includes positive and negative values of k.) Prove this identity.

63. [*50*] Develop computer programs for simplifying sums that involve binomial coefficients.

▸ **64.** [*M22*] Show that $\{{n \atop m}\}$ is the number of ways to partition a set of n elements into m nonempty disjoint subsets. For example, the set {1, 2, 3,4} can be partitioned into two subsets in $\{{4 \atop 2}\} = 7$ ways: {1, 2, 3} {4}; {1, 2, 4} {3}; {1, 3, 4} {2}; {2, 3, 4} {1}; {1, 2} {3, 4}; {1, 3} {2, 4}; {1, 4} {2, 3}. *Hint:* Use the fact that

$$\begin{Bmatrix} n \\ m \end{Bmatrix} = m \begin{Bmatrix} n-1 \\ m \end{Bmatrix} + \begin{Bmatrix} n-1 \\ m-1 \end{Bmatrix}.$$

Note that the result of this exercise provides us with a mnemonic device for remembering the difference between the "[ ]" and "{ }" notations for Stirling numbers, since "{ }" is commonly used also for sets. The other Stirling numbers $[{n \atop k}]$ also have a combinatorial interpretation: $[{n \atop k}]$ is the number of permutations on n letters having k "cycles"; see Section 1.3.3.

1.2.7. Harmonic Numbers

The following sum will be of great importance in our later work:

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \sum_{1 \le k \le n} \frac{1}{k}, \qquad n \ge 0. \tag{1}$$

This sum does not occur very frequently in classical mathematics, and there is no standard notation for it; but in the analysis of algorithms it pops up nearly every time we turn around, and we will consistently use the symbol H_n to represent the above quantity. (Besides H_n, the notations h_n and S_n are occasionally used in mathematical literature. The letter H stands for "harmonic," and we call H_n a *harmonic number* because (1) is customarily called the harmonic series.)

It may seem at first that H_n does not get too large when n has a large value, since we are always adding smaller and smaller numbers. But actually it is not hard to see that H_n will get as large as we please if we take n to be big enough, because of the following rule:

$$H_{2^m} \geq 1 + \frac{m}{2}. \tag{2}$$

This rule may be proved by observing that, for $m > 0$,

$$\begin{aligned} H_{2^{m+1}} &= H_{2^m} + \frac{1}{2^m+1} + \frac{1}{2^m+2} + \cdots + \frac{1}{2^{m+1}} \\ &> H_{2^m} + \frac{1}{2^{m+1}} + \frac{1}{2^{m+1}} + \cdots + \frac{1}{2^{m+1}} \\ &= H_{2^m} + \tfrac{1}{2}. \end{aligned}$$

So as m increases by 1, the left-hand side of Eq. (2) increases by at least $\frac{1}{2}$.

It is important to have more detailed information about the value of H_n than is given in Eq. (2). The approximate size of H_n is a well-known quantity (at least in mathematical circles) which may be expressed as follows:

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \epsilon, \qquad 0 < \epsilon < \frac{1}{252n^6}. \tag{3}$$

Here $\gamma = 0.57721\ 56649\ldots$ is *Euler's constant*. Exact values of H_n for small n, and a 40-place value for γ, are given in the tables in Appendix B. We shall prove Eq. (3) in Section 1.2.11.2.

Thus H_n is reasonably close to the natural logarithm of n. Exercise 7 shows that H_n has a somewhat logarithmic behavior.

In a sense, H_n "just barely" goes to infinity as n gets large, because it can be proved that the sum

$$1 + \frac{1}{2^r} + \frac{1}{3^r} + \cdots + \frac{1}{n^r} \tag{4}$$

stays bounded for all n, when r is any real-valued exponent *greater* than unity. (See exercise 3.) We denote the sum in Eq. (4) by $H_n^{(r)}$.

When r in Eq. (4) is two or more, the value of $H_n^{(r)}$ is fairly close to its maximum value $H_\infty^{(r)}$, except for very small n. The quantity $H_\infty^{(r)}$ is very well known in mathematics as Riemann's "zeta function":

$$H_\infty^{(r)} = \zeta(r). \tag{5}$$

When r is an *even integer*, the value of $\zeta(r)$ is known to be equal to

$$H_\infty^{(r)} = \frac{1}{2}|B_r| \frac{(2\pi)^r}{r!}, \tag{6}$$

where B_r is a Bernoulli number (see Section 1.2.11.2 and Appendix B). In

particular,

$$H_\infty^{(2)} = \frac{\pi^2}{6}, \qquad H_\infty^{(4)} = \frac{\pi^4}{90}, \qquad H_\infty^{(6)} = \frac{\pi^6}{945}, \qquad H_\infty^{(8)} = \frac{\pi^8}{9450}. \tag{7}$$

For discussion and proof, see K. Knopp, *Theory and Application of Infinite Series*, tr. by R. C. H. Young (Glasgow: Blackie, 1951), Section 32.4.

Now we will consider a few important properties involving summations. First,

$$\sum_{1\le k\le n} H_k = (n+1)H_n - n. \tag{8}$$

This follows from simple transformation of sums:

$$\sum_{1\le k\le n}\ \sum_{1\le j\le k} \frac{1}{j} = \sum_{1\le j\le n}\ \sum_{j\le k\le n} \frac{1}{j} = \sum_{1\le j\le n} \frac{n+1-j}{j}.$$

Formula (8) is a special case of the sum $\sum_{1\le k\le n} \binom{k}{m} H_k$, which we will now determine. The "trick" to be used here is called summation by parts, and it is a useful technique for determining $\sum a_k b_k$ when the quantities $\sum a_k$ and $(b_{k+1} - b_k)$ have simple forms (see exercise 10). We observe in this case that

$$\binom{k}{m} = \binom{k+1}{m+1} - \binom{k}{m+1},$$

and therefore

$$\binom{k}{m} H_k = \binom{k+1}{m+1}\left(H_{k+1} - \frac{1}{k+1}\right) - \binom{k}{m+1} H_k;$$

hence

$$\begin{aligned}
&\sum_{1\le k\le n} \binom{k}{m} H_k \\
&\quad = \left(\binom{2}{m+1} H_2 - \binom{1}{m+1} H_1\right) + \cdots \\
&\qquad + \left(\binom{n+1}{m+1} H_{n+1} - \binom{n}{m+1} H_n\right) - \sum_{1\le k\le n} \binom{k+1}{m+1} \frac{1}{k+1} \\
&\quad = \binom{n+1}{m+1} H_{n+1} - \binom{1}{m+1} H_1 - \frac{1}{m+1} \sum_{0\le k\le n} \binom{k}{m} + \frac{1}{m+1}\binom{0}{m}.
\end{aligned}$$

Applying Eq. 1.2.6–11 yields the desired formula:

$$\sum_{1\le k\le n} \binom{k}{m} H_k = \binom{n+1}{m+1}\left(H_{n+1} - \frac{1}{m+1}\right). \tag{9}$$

(The above derivation and final result are somewhat analogous to the determination of $\int_1^n x^m \ln x\, dx$ in integral calculus.)

We conclude this section by considering a different kind of sum, $\sum_k \binom{n}{k} x^k H_k$, which we will temporarily denote by S_n for brevity. We find that

$$S_{n+1} = \sum_k \left(\binom{n}{k} + \binom{n}{k-1} \right) x^k H_k = S_n + x \sum_k \binom{n}{k-1} x^{k-1} \left(H_{k-1} + \frac{1}{k} \right)$$

$$= S_n + x\, S_n + \frac{1}{n+1} \sum_{k \geq 1} \binom{n+1}{k} x^k.$$

Hence $S_{n+1} = (x+1)S_n + [(x+1)^{n+1} - 1]/(n+1)$, and we have

$$\frac{S_{n+1}}{(x+1)^{n+1}} = \frac{S_n}{(x+1)^n} + \frac{1}{n+1} - \frac{1}{(n+1)(x+1)^{n+1}}.$$

This equation, together with the fact that $S_1 = x$, shows us that

$$\frac{S_n}{(x+1)^n} = H_n - \sum_{1 \leq k \leq n} \frac{1}{k(x+1)^k}. \tag{10}$$

The remaining sum is part of the infinite series for $\ln(1/(1 - 1/(x+1))) = \ln(1 + 1/x)$, and when $x > 0$, the series is convergent; the difference is

$$\sum_{k>n} \frac{1}{k(x+1)^k} < \frac{1}{(n+1)(x+1)^{n+1}} \sum_{k \geq 0} \frac{1}{(x+1)^k} = \frac{1}{(n+1)(x+1)^n x}.$$

This proves the following theorem:

Theorem A. *If $x > 0$, then*

$$\sum_{1 \leq k \leq n} \binom{n}{k} x^k H_k = (x+1)^n \left(H_n - \ln\left(1 + \frac{1}{x}\right) \right) + \epsilon,$$

where $0 < \epsilon < 1/x(n+1)$. ∎

EXERCISES

1. [*01*] What are H_0, H_1, and H_2?

2. [*13*] Show that the simple argument used in the text to prove that $H_{2^m} \geq 1 + m/2$ can be slightly modified to prove that $H_{2^m} \leq 1 + m$.

3. [*M21*] Generalize the argument used in the previous exercise to show that $H_n^{(r)}$ remains bounded for all n, and find an upper bound, assuming that $r > 1$.

▶ **4.** [*10*] Which of the following statements are true for all positive integers n? (a) $H_n < \ln n$. (b) $H_n > \ln n$. (c) $H_n > \ln n + \gamma$.

5. [*15*] Give the value of H_{10000} to 15 decimal places, using the tables in Appendix B.

6. [*M15*] Prove that the harmonic numbers are directly related to Stirling's numbers, which were introduced in the previous section; in fact,

$$H_n = \begin{bmatrix} n+1 \\ 2 \end{bmatrix} \Big/ n!.$$

7. [*M21*] Let $T(m, n) = H_m + H_n - H_{mn}$. (a) Show that if m or n increases, $T(m, n)$ decreases (assuming that m and n are positive). (b) Compute the minimum and maximum values of $T(m, n)$ for $m, n > 0$.

8. [*M18*] Compare Eq. (8) with $\sum_{1\le k\le n} \ln k$; estimate the difference as a function of n.

▶ **9.** [*M18*] Theorem A applies only when $x > 0$; what is the value of the sum considered when $x = -1$?

10. [*M20*] (Summation by parts.) We have used special cases of the general method of summation by parts in exercise 1.2.4–42 and in the derivation of Eq. (9). Prove the general formula

$$\sum_{1\le k<n} (a_{k+1} - a_k)b_k = a_n b_n - a_1 b_1 - \sum_{1\le k<n} a_{k+1}(b_{k+1} - b_k).$$

▶ **11.** [*M21*] Using summation by parts, evaluate

$$\sum_{1<k\le n} \frac{1}{k(k-1)} H_k.$$

▶ **12.** [*M10*] Evaluate $H_\infty^{(1000)}$ correct to at least 100 decimal places.

13. [*M22*] Prove the identity

$$\sum_{1\le k\le n} \frac{x^k}{k} = H_n + \sum_{1\le k\le n} \binom{n}{k} \frac{(x-1)^k}{k}.$$

(Note in particular the special case $x = 0$, which gives us an identity related to exercise 1.2.6–48.)

14. [*M22*] Show that

$$\sum_{1\le k\le n} \frac{H_k}{k} = \tfrac{1}{2}(H_n^2 + H_n^{(2)}),$$

and evaluate $\sum_{1\le k\le n} H_k/(k+1)$.

▶ **15.** [*M23*] Express $\sum_{1\le k\le n} H_k^2$ in terms of n and H_n.

16. [*18*] Express the sum $1 + \frac{1}{3} + \cdots + 1/(2n+1)$ in terms of harmonic numbers.

17. [*M24*] (E. Waring, 1782.) Let p be an odd prime. Show that the numerator of H_{p-1} is divisible by p.

18. [*M33*] (J. Selfridge.) What is the highest power of 2 which divides the numerator of $1 + \frac{1}{3} + \cdots + 1/(2n-1)$?

▸ **19.** [*M30*] List all nonnegative integers n for which H_n is an integer. [*Hint:* If $H_n =$ odd/even, it cannot be an integer.]

20. [*HM22*] There is an analytic way to approach summation problems such as the one leading to Theorem A in this section: If $f(x) = \sum_{k\geq 0} a_k x^k$, and this series converges for $x = x_0$, then show that

$$\sum_{k\geq 0} a_k x_0^k H_k = \int_0^1 \frac{f(x_0) - f(x_0 y)}{1 - y}\, dy.$$

21. [*M24*] Evaluate $\sum_{1\leq k\leq n} H_k/(n + 1 - k)$.

22. [*M28*] Evaluate $\sum_{1\leq k\leq n} H_k H_{n+1-k}$.

▸ **23.** [*HM20*] By considering the function $\Gamma'(x)/\Gamma(x)$, show how we can get a natural generalization of H_n to noninteger values of n. You may use the fact that $\Gamma'(1) = -\gamma$, anticipating the next exercise.

24. [*HM21*] Show that

$$xe^{\gamma x} \prod_{k\geq 1} \left(\left(1 + \frac{x}{k}\right) e^{-x/k}\right) = \frac{1}{\Gamma(x)}.$$

(Consider the partial products of this infinite product.)

1.2.8. Fibonacci Numbers

The sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots, \tag{1}$$

in which each number is the sum of the preceding two, plays an important role in at least a dozen seemingly unrelated algorithms which we will study later. The numbers of the sequence are denoted by F_n, and we formally define this as

$$F_0 = 0, \qquad F_1 = 1, \qquad F_{n+2} = F_{n+1} + F_n, \quad n \geq 0. \tag{2}$$

This famous sequence was originated in 1202 by Leonardo Pisano (Leonardo of Pisa), who is sometimes called Leonardo Fibonacci (*Filius Bonaccii*, son of Bonaccio). His *Liber Abbaci* (Book of the Abacus) contains the following exercise: "How many pairs of rabbits can be produced from a single pair in a year's time?" To solve this problem, we are told to assume that each pair produces a new pair of offspring every month, each new pair becomes fertile at the age of one month, and furthermore, the rabbits never die. After one month there will be 2 pairs of rabbits; after two months, there will be 3; the following month the original pair and the pair born during the first month will both usher in a new pair and there will be 5 in all; and so on.

Fibonacci was by far the greatest European mathematician before the Renaissance. He studied the work of al-Khowârizmî (after whom "algorithm" is named, see Section 1.1) and he added numerous original contributions to

arithmetic and geometry. The writings of Fibonacci were reprinted in 1857 [B. Boncompagni, *Scritti di Leonardo Pisano* (Rome, 1857), 2 vols.; F_n appears in Vol. 1, pp. 283–285]. The rabbit problem was, of course, not posed as a practical application to biology and the population explosion; it was an exercise in addition. In fact, it still makes a rather good computer exercise in addition (cf. exercise 3); Fibonacci wrote: "It is possible to do [the addition] in this order for an infinite number of months."

The same sequence also appears in the work of Kepler, 1611, in connection with "phyllotaxis," the study of the arrangement of leaves and flowers in plant life. Kepler was presumably unaware of Fibonacci's brief mention of the sequence. Fibonacci numbers have often been observed in nature, probably for reasons similar to the original assumptions of the rabbit problem.

A first indication of the intimate connections between F_n and algorithms came to light in 1844, when G. Lamé used Fibonacci's sequence to study the efficiency of Euclid's algorithm. He proved that if the numbers m, n in algorithm 1.1E are not greater than F_k, step E2 will be executed at most $k + 1$ times. This was the first practical application of Fibonacci's sequence. During the next 50 years the mathematician E. Lucas obtained very profound results about the Fibonacci numbers, and in particular he used them to prove that the 39-digit number $2^{127} - 1$ is prime. Lucas gave the name "Fibonacci numbers" to the sequence F_n, and that name has been used ever since.

We already have examined the Fibonacci sequence briefly in Section 1.2.1 (Eq. (3) and exercise 4), where we found that $\phi^{n-2} \leq F_n \leq \phi^{n-1}$, if n is a positive integer and if

$$\phi = \tfrac{1}{2}(1 + \sqrt{5}). \tag{3}$$

We will see shortly that this quantity, ϕ, is intimately connected with the Fibonacci numbers.

The number ϕ itself has a very interesting history. Euclid called it the "extreme and mean ratio"; the ratio of A to B is the ratio of $(A + B)$ to A, if the ratio of A to B is ϕ. Renaissance writers called it the "divine proportion"; and in the last century it has commonly been called the "golden ratio." In the art world, the ratio of ϕ to 1 is said to be the most pleasing proportion aesthetically, and this opinion is confirmed from the standpoint of computer programming aesthetics as well. For the story of ϕ, see the excellent article "The Golden Section, Phyllotaxis, and Wythoff's Game," by H. S. M. Coxeter, *Scripta Math.* **19** (1953), 135–143, and see also Chapter 8 of *The 2nd Scientific American book of Mathematical Puzzles and Diversions*, by Martin Gardner (New York: Simon and Schuster, 1961).

The notations we are using in this section are a little undignified. In most of the sophisticated mathematical literature, F_n is called u_n instead, and ϕ is called τ. Our notations are almost universally used in recreational mathematics (and some crank literature!) and they are rapidly coming into wider use. The designation ϕ comes from the first letter of the Greek artist Phidias who is said to have

used the golden ratio frequently in his sculpture. The notation F_n is in accordance with that used in the *Fibonacci Quarterly* journal (published 1963–) where the reader may find numerous facts about the Fibonacci sequence. A good reference to the classical literature about Fibonacci's sequence is Chapter 17 of L. E. Dickson's *History of the Theory of Numbers*, Vol. 1 (New York: Chelsea, 1952).

The Fibonacci numbers satisfy many interesting identities, some of which appear in the exercises at the end of this section. One of the most commonly quoted relations is

$$F_{n+1}F_{n-1} - F_n^2 = (-1)^n, \tag{4}$$

which is easily proved by induction. A more esoteric method of proving the same formula starts with a simple inductive proof of the matrix identity

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}. \tag{5}$$

We then take the determinant of both sides of this equation.

Relation (4) shows that F_n and F_{n+1} are relatively prime, since any common divisor would have to be a divisor of $(-1)^n$.

From the definition (2) we find immediately that

$$F_{n+3} = F_{n+2} + F_{n+1} = 2F_{n+1} + F_n; \qquad F_{n+4} = 3F_{n+1} + 2F_n;$$

and, in general, by induction that

$$F_{n+m} = F_mF_{n+1} + F_{m-1}F_n \tag{6}$$

for any positive integer m.

If we take m to be a multiple of n in Eq. (6), we find inductively that

$$F_{nk} \quad \textit{is a multiple of} \quad F_k.$$

Thus every third number is even, every fourth number is a multiple of 3, every fifth is a multiple of 5, and so on.

In fact, much more than this is true. If we write gcd (m, n) to stand for the greatest common divisor of m and n, we have the rather surprising theorem:

Theorem A. (E. Lucas, 1876). *A number divides both F_m and F_n if and only if it is a divisor of F_d, where $d = \gcd(m, n)$; in particular,*

$$\gcd(F_m, F_n) = F_{\gcd(m,n)}. \tag{7}$$

Proof: This result is proved by using Euclid's algorithm. We observe that because of Eq. (6) any common divisor of F_m and F_n is also a divisor of F_{n+m}; and, conversely, any common divisor of F_{n+m} and F_n is a divisor of F_mF_{n+1}. Since F_{n+1} is relatively prime to F_n, a common divisor of F_{n+m} and F_n also

divides F_m. Thus we have proved that for any number d

$$d \text{ divides } F_m \text{ and } F_n \text{ if and only if } d \text{ divides } F_{m+n} \text{ and } F_n. \tag{8}$$

We will now show that *any* sequence, F_n, for which statement (8)holds and for which $F_0 = 0$, satisfies Theorem A.

First it is clear that statement (8) may be extended by induction on k to the rule

$$d \text{ divides } F_m \text{ and } F_n \text{ if and only if } d \text{ divides } F_{m+kn} \text{ and } F_n,$$

where k is any nonnegative integer. This result may be stated more succinctly:

$$d \text{ divides } F_{(m \bmod n)} \text{ and } F_n \text{ if and only if } d \text{ divides } F_m \text{ and } F_n. \tag{9}$$

Now if r is the remainder after division of m by n, that is, if $r = m \bmod n$, then the common divisors of F_m, F_n are the common divisors of F_n, F_r. It follows that throughout the manipulations of Algorithm 1.1E the set of common divisors of F_m, F_n remains unchanged as m and n change; finally, when $r = 0$, the common divisors are simply the divisors of $F_0 = 0$ and $F_{\gcd(m,n)}$. ∎

Most of the important results involving Fibonacci numbers can be deduced from the representation of F_n in terms of ϕ, which we now proceed to derive. The method we shall use in the following derivation is extremely important, and the mathematically oriented reader should study it carefully; we will study the same method in detail in the next section.

We start by setting up the infinite series

$$\begin{aligned} G(z) &= F_0 + F_1 z + F_2 z^2 + F_3 z^3 + F_4 z^4 + \cdots \\ &= z + z^2 + 2z^3 + 3z^4 + \cdots \end{aligned} \tag{10}$$

We have no *a priori* reason to expect that this infinite sum exists or that the function $G(z)$ is at all interesting—but let us be optimistic and see what we can conclude about the function $G(z)$ if it does exist. The advantage of such a procedure is that $G(z)$ is a single quantity which represents the *entire* Fibonacci sequence at once; and if we find out that $G(z)$ is a "known" function, its coefficients can be determined. $G(z)$ is called the *generating function* for the sequence $\langle F_n \rangle$.

We can now proceed to investigate $G(z)$ as follows:

$$\begin{aligned} zG(z) &= F_0 z + F_1 z^2 + F_2 z^3 + F_3 z^4 + \cdots \\ z^2 G(z) &= \qquad F_0 z^2 + F_1 z^3 + F_2 z^4 + \cdots ; \end{aligned}$$

by subtraction,

$$\begin{aligned} (1 - z - z^2)G(z) &= F_0 + (F_1 - F_0)z + (F_2 - F_1 - F_0)z^2 \\ &\quad + (F_3 - F_2 - F_1)z^3 + (F_4 - F_3 - F_2)z^4 + \cdots \\ &= z. \end{aligned}$$

All further terms are zero because of the definition of F_n; and so we see that, if $G(z)$ exists,

$$G(z) = z/(1 - z - z^2). \tag{11}$$

In fact, this function *can* be expanded in an infinite series in z (a Taylor series); working backwards we find the coefficients of the power series expansion of Eq. (11) must be the Fibonacci numbers.

We can now manipulate $G(z)$ and find out more about the Fibonacci sequence. The denominator $1 - z - z^2$ is a quadratic equation with the two roots $\frac{1}{2}(-1 \pm \sqrt{5})$; after a little calculation we find that $G(z)$ can be expanded by the method of partial fractions into the form

$$G(z) = \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z}\right), \tag{12}$$

where

$$\hat{\phi} = 1 - \phi = \tfrac{1}{2}(1 - \sqrt{5}). \tag{13}$$

The quantity $1/(1 - \phi z)$ is the sum of the infinite geometric series $1 + \phi z + \phi^2 z^2 + \cdots$, so we have

$$G(z) = \frac{1}{\sqrt{5}}(1 + \phi z + \phi^2 z^2 + \cdots - 1 - \hat{\phi} z - \hat{\phi}^2 z^2 - \cdots).$$

We now look at the coefficient of z^n, *which must be equal to* F_n, and we find that

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n). \tag{14}$$

This is an important "closed form" expression for the Fibonacci numbers, first discovered by A. de Moivre in 1718. The method we have used to derive Eq. (14) is due to Nicolas Bernoulli (1728).

We could have merely stated Eq. (14) and proved it by induction; the point of the rather long derivation above was to show how it would be possible to *discover* the equation in the first place, using the important method of generating functions, which is a valuable technique for solving so many problems.

Many things can be proved from Eq. (14). First we observe that $\hat{\phi}$ is a *negative* number (−0.61803 . . .) whose magnitude is less than unity, so $\hat{\phi}^n$ gets very small as n gets large. In fact, $\hat{\phi}^n/\sqrt{5}$ is always small enough so that we have

$$F_n = \phi^n/\sqrt{5} \qquad \textit{rounded to the nearest integer.} \tag{15}$$

Other results can be obtained directly from $G(z)$; for example,

$$G(z)^2 = \frac{1}{5}\left(\frac{1}{(1 - \phi z)^2} + \frac{1}{(1 - \hat{\phi} z)^2} - \frac{2}{1 - z - z^2}\right), \tag{16}$$

and the coefficient of z^n in $G(z)^2$ is $\sum_{0\le k\le n} F_k F_{n-k}$. We therefore deduce that

$$\begin{aligned}\sum_{0\le k\le n} F_k F_{n-k} &= \tfrac{1}{5}\big((n+1)(\phi^n+\hat\phi^n) - 2F_{n+1}\big)\\ &= \tfrac{1}{5}\big((n+1)(F_n+2F_{n-1}) - 2F_{n+1}\big)\\ &= \frac{n-1}{5}F_n + \frac{2n}{5}F_{n-1}. \qquad (17)\end{aligned}$$

(The second step in this derivation follows from the result of exercise 11.)

EXERCISES

1. [*10*] In Leonardo Fibonacci's problem, how many pairs of rabbits are present after k months? What is the answer to his question, i.e., how many pairs are present after a year?

▶ **2.** [*20*] In lieu of Eq. (15), what is the approximate value of F_{1000}? (Use logarithms found in the table in Appendix B.)

3. [*34*] Write a program for some computer which calculates and prints F_1 through F_{1000}. (Cf. the previous exercise for the size of the numbers which must be handled.)

▶ **4.** [*14*] Find all n for which $F_n = n$.

5. [*20*] Find all n for which $F_n = n^2$.

6. [*HM10*] Prove Eq. (5).

▶ **7.** [*15*] If n is not a prime number, F_n is not a prime number (with one exception). Prove this and find the exception.

8. [*15*] In many cases it is convenient to define F_n for *negative* n, by assuming that $F_{n+2} = F_{n+1} + F_n$ for all integers n. Explore this possibility; what is F_{-1}? What is F_{-2}? Can F_{-n} be expressed in a simple way in terms of F_n?

9. [*M20*] Using the conventions of the preceding exercise, determine whether Eqs. (4), (6), (14), and (15) still hold when the subscripts are allowed to be *any* integer.

10. [*15*] Is $\phi^n/\sqrt{5}$ greater than F_n or less than F_n?

11. [*M20*] Show that $\phi^n = F_n\phi + F_{n-1}$, $\hat{\phi}^n = F_n\hat{\phi} + F_{n-1}$, for *all* integers n.

▶ **12.** [*M26*] The "second order" Fibonacci sequence is defined by the rule

$$\mathfrak{F}_0 = 0, \qquad \mathfrak{F}_1 = 1, \qquad \mathfrak{F}_{n+2} = \mathfrak{F}_{n+1} + \mathfrak{F}_n + F_n.$$

Express $\mathfrak{F}_n$ in terms of F_n. [*Hint:* Use generating functions.]

▶ **13.** [*M22*] Express the following sequences in terms of the Fibonacci numbers:

a) $a_0 = r, \quad a_1 = s, \quad a_{n+2} = a_{n+1} + a_n, \qquad n \geq 0.$

b) $b_0 = 0, \quad b_1 = 1, \quad b_{n+2} = b_{n+1} + b_n + c, \qquad n \geq 0.$

14. [*M28*] Let m be a fixed positive integer. Find a_n given that

$$a_0 = 0, a_1 = 1, a_{n+2} = a_{n+1} + a_n + \binom{n}{m}.$$

15. [*M22*] Let $f(n)$, $g(n)$ be arbitrary functions. Let

$$\begin{aligned} a_0 &= 0, \quad a_1 = 1, \quad a_{n+2} = a_{n+1} + a_n + f(n); \\ b_0 &= 0, \quad b_1 = 1, \quad b_{n+2} = b_{n+1} + b_n + g(n); \\ c_0 &= 0, \quad c_1 = 1, \quad c_{n+2} = c_{n+1} + c_n + xf(n) + yg(n). \end{aligned}$$

Express c_n in terms of x, y, a_n, b_n, and F_n.

▶ **16.** [*M20*] Fibonacci numbers appear implicitly in Pascal's triangle if it is viewed from the right angle. Show that the following sum of binomial coefficients is a Fibonacci number:

$$\sum_{0\le k\le n} \binom{n-k}{k}.$$

17. [*M24*] Using the conventions of exercise 8, prove the following generalization of Eq. (4): $F_{n+k}F_{m-k} - F_nF_m = (-1)^nF_{m-n-k}F_k$.

18. [*20*] Is $F_n^2 + F_{n+1}^2$ always a Fibonacci number?

19. [*M27*] What is $\cos 36°$?

20. [*M16*] Express $\sum_{0\le k\le n} F_k$ in terms of Fibonacci numbers.

21. [*M25*] What is $\sum_{0\le k\le n} F_kx^k$?

▶ **22.** [*M20*] Show that $\sum_k \binom{n}{k}F_{m+k}$ is a Fibonacci number.

23. [*M23*] Generalizing the preceding exercise, show that $\sum_k \binom{n}{k}F_t^kF_{t-1}^{n-k}F_{m+k}$ is always a Fibonacci number.

24. [*HM20*] Evaluate the $n \times n$ determinant

$$\begin{pmatrix} 1 & -1 & 0 & 0 & \ldots & 0 & 0 & 0 \\ 1 & 1 & -1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & 1 & -1 & \ldots & 0 & 0 & 0 \\ \vdots & & & & & & & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 & 1 & -1 \\ 0 & 0 & 0 & 0 & \ldots & 0 & 1 & 1 \end{pmatrix}.$$

25. [*M21*] Show that

$$2^nF_n = 2\sum_{k \text{ odd}} \binom{n}{k} 5^{(k-1)/2}.$$

▶ **26.** [*M20*] Using the previous exercise, show that $F_p \equiv 5^{(p-1)/2}$ (modulo p) if p is an odd prime.

27. [*M20*] Using the previous exercise, show that if p is a prime different from 5, then either F_{p-1} or F_{p+1} (not both) is a multiple of p.

28. [*M21*] What is $F_{n+1} - \phi F_n$?

▶ **29.** [*M23*] (The Fibonacci triangle.) Define

$$\left(\binom{n}{k}\right) = \frac{F_nF_{n-1}\cdots F_{n-k+1}}{F_kF_{k-1}\cdots F_1} = \prod_{1\le j\le k} \left(\frac{F_{n-k+j}}{F_j}\right)$$

in a manner analogous to binomial coefficients. (a) Make a table of $\left(\binom{n}{k}\right)$ for $0 \le n \le 6$. (b) Show that

$$\left(\binom{n}{k}\right) = F_{k-1}\left(\binom{n-1}{k}\right) + F_{n-k+1}\left(\binom{n-1}{k-1}\right).$$

▶ **30.** [*M38*] (D. Jarden.) The sequence of mth powers of Fibonacci numbers satisfies a recurrence relation in which each term depends on the preceding $m+1$ terms. Show that

$$\sum_k \left(\binom{m}{k}\right) (-1)^{\lfloor (m-k)/2 \rfloor} F_{n+k}^{m-1} = 0, \qquad \text{if} \qquad m > 0.$$

For example, when $m = 3$, we get the identity $F_n^2 - 2F_{n+1}^2 - 2F_{n+2}^2 + F_{n+3}^2 = 0$.

31. [*M20*] Let $\psi = \phi - 1 = 1/\phi$. Show that $(F_{2n}\psi) \bmod 1 = 1 - \psi^{2n}$ and $(F_{2n+1}\psi) \bmod 1 = \psi^{2n+1}$.

32. [*M24*] The remainder of one Fibonacci number divided by another is $\pm$ a Fibonacci number: Show that

$$F_{mn+r} \equiv F_r, \quad (-1)^{r+1}F_{n-r}, \quad (-1)^n F_r, \qquad \text{or} \qquad (-1)^{r+1+n}F_{n-r} \qquad (\text{modulo } F_n),$$

depending on whether $m \bmod 4 = 0$, 1, 2, or 3, respectively.

33. [*HM24*] Given that $z = \pi/2 + i \ln \phi$, show that $\sin(nz)/\sin z = i^{1-n}F_n$.

▶ **34.** [*M24*] (The Fibonacci number system.) Let the notation $k \gg m$ mean that $k \ge m + 2$. Show that every positive integer n has a *unique* representation $n = F_{k_1} + F_{k_2} + \cdots + F_{k_r}$, where $k_1 \gg k_2 \gg \cdots \gg k_r \gg 0$.

35. [*M24*] (A phi number system.) Consider real numbers written with the digits 0 and 1 using base ϕ. (Thus $100.1 = \phi^2 + \phi^{-1}$.) Show that there are infinitely many ways to represent the number 1 (for example, $1 = .11 = .011111\ldots$); but if we require that no two adjacent 1's occur and that no infinite sequence "01010101 . . ." appears, every number has a unique representation.

▶ **36.** [*M32*] ("Fibonacci·string sequence.") Let $S_1 =$ "*a*", $S_2 =$ "*b*", and $S_{n+2} = S_{n+1}S_n$, $n > 0$; in other words, S_{n+2} is formed by placing S_n at the right of S_{n+1}. We have $S_3 =$ "*ba*", $S_4 =$ "*bab*", $S_5 =$ "*babba*", etc. Clearly S_n has F_n letters. Explore the properties of S_n. (Where do double letters occur? Can you predict the value of the kth letter of S_n? What is the density of the *b*'s? And so on.)

▶ **37.** [*M35*] (R. E. Gaskell, M. J. Whinihan.) Two players compete in the following game: There is a pile containing n chips; the first player removes any number of chips except that he cannot take the whole pile. From then on, the players alternate moves, each person removing one or more chips but *not more than twice as many chips as the preceding player has taken.* The player who removes the last chip wins. (For example, suppose that $n = 11$; player A removes 3 chips; player B may remove up to 6 chips, and he takes 1. There remain 7 chips; player A may take 1 or 2 chips, and he takes 2; player B may remove up to 4, and he picks up 1. There remain 4 chips; player A now takes 1; player B must take at least one chip and player A wins in the following turn.)

What is the best move for the first player to make if there are initially 1000 chips?

38. [*35*] Write a computer program which plays the game described in the previous exercise and which plays optimally.

39. [*M24*] Find a closed form expression for a_n, given that $a_0 = 0$, $a_1 = 1$, $a_{n+2} = a_{n+1} + 6a_n$.

1.2.9. Generating Functions

Whenever we want to obtain information about a sequence of numbers $\langle a_n \rangle = a_0, a_1, a_2, \ldots$, we can set up an infinite sum in terms of a "parameter" z,

$$G(z) = a_0 + a_1 z + a_2 z^2 + \cdots = \sum_{n \geq 0} a_n z^n. \tag{1}$$

We can then try to obtain information about the function G. This function G is a single quantity which represents the whole sequence $\langle a_n \rangle$; if the sequence $\langle a_n \rangle$ has been defined inductively (that is, if a_n has been defined in terms of $a_0, a_1, \ldots, a_{n-1}$), this is an important advantage. Furthermore, we can recover the values of $a_0, a_1, \ldots$ from the function $G(z)$, assuming that the infinite sum in Eq. (1) exists for some values of z, by using techniques of differential calculus.

$G(z)$ is called the *generating function* for the sequence $a_0, a_1, a_2, \ldots$ The use of generating functions opens up a whole new range of techniques, and it broadly increases our capacity for problem solving. Starting in 1741, L. Euler used generating functions in several number-theoretic investigations. Pierre S. Laplace developed the techniques further in his classic work *Théorie Analytique des Probabilities* (1812). A similar concept, which has become known as the "Laplace transform," was actually introduced by another mathematician N. Abel [*Collected Works* (Oslo, 1881), Vol. 2, pp. 67–81]. (See exercise 13 for a connection between generating functions and Laplace transforms.) Part of the credit for discovering generating functions belongs also to N. Bernoulli (1728) and J. Stirling (1730); see C. Tweedie, *James Stirling*, (London: Oxford University Press, 1922), pp. 37–38, 95.

The question of convergence of the infinite sum, Eq. (1), is of some importance. Any textbook about the theory of infinite series will prove that:

a) If Eq. (1) exists ("converges") for a particular value of $z = z_0$, then it converges for all values of z with $|z| < z_0$.
b) The sequence converges for some $z \neq 0$ if and only if the sequence $\sqrt[n]{|a_n|}$ is bounded. (If this condition is not satisfied, it may be possible to get a convergent series for a related sequence, e.g., for the sequence $\langle a_n/n! \rangle$.)

On the other hand, it often does not pay to worry about convergence of the series when we work with generating functions, since we are only exploring possible approaches to the solution of some problem. When we discover the solution by *any* means, however sloppy they might be, it may be possible to justify the solution independently. For example, in the previous section we used a generating function to deduce Eq. (14); yet once this equation has been found,

it is a simple matter to prove it by induction, and we need not even mention that we used generating functions to discover that relation.

Let us now study the principal techniques used with generating functions.

A. Addition. If $G_1(z)$ is the generating function for $a_0, a_1, \ldots$ and $G_2(z)$ is the generating function for $b_0, b_1, \ldots$, then $\alpha G_1(z) + \beta G_2(z)$ is the generating function for $\alpha a_0 + \beta b_0, \alpha a_1 + \beta b_1, \ldots$:

$$\alpha \sum_{k\geq 0} a_k z^k + \beta \sum_{k\geq 0} b_k z^k = \sum_{k\geq 0} (\alpha a_k + \beta b_k) z^k. \tag{2}$$

B. Shifting. If $G(z)$ is the generating function for $a_0, a_1, \ldots$ then $z^n G(z)$ is the generating function for $0, \ldots, 0, a_0, a_1, \ldots$:

$$z^n \sum_{k\geq 0} a_k z^k = \sum_{k\geq n} a_{k-n} z^k. \tag{3}$$

The last summation may be extended over all $k \geq 0$ if we regard $a_k = 0$ for any negative value of k.

Similarly, $(G(z) - a_0 - a_1 z - \cdots - a_{n-1} z^{n-1})/z^n$ is the generating function for $a_n, a_{n+1}, \ldots$:

$$z^{-n} \sum_{k\geq n} a_k z^k = \sum_{k\geq 0} a_{k+n} z^k. \tag{4}$$

We combined operations A and B to solve the Fibonacci problem in the previous section; $G(z)$ was the generating function for $\langle F_n \rangle$, $zG(z)$ for $\langle F_{n-1} \rangle$, $z^2 G(z)$ for $\langle F_{n-2} \rangle$, and $(1 - z - z^2)G(z)$ for $\langle F_n - F_{n-1} - F_{n-2} \rangle$. The latter sequence is zero when $n \geq 2$, so $(1 - z - z^2)G(z)$ is a polynomial. The same technique applies to any "linearly recurrent" sequence where $a_n = c_1 a_{n-1} + \cdots + c_m a_{n-m}$; the generating function is a polynomial divided by $(1 - c_1 z - \cdots - c_m z^m)$.

Let us consider the simplest example of all: If $G(z)$ is the generating function for the *constant* sequence $1, 1, 1, \ldots$, then $zG(z)$ generates $0, 1, 1, \ldots$, so $(1 - z)G(z) = 1$. This gives us the very special case

$$\frac{1}{1 - z} = 1 + z + z^2 + \cdots. \tag{5}$$

C. Multiplication. If $G_1(z)$ is the generating function for $a_0, a_1, \ldots$ and $G_2(z)$ is the generating function for $b_0, b_1, \ldots$, then

$$\begin{aligned} G_1(z)G_2(z) &= (a_0 + a_1 z + a_2 z^2 + \cdots)(b_0 + b_1 z + b_2 z^2 + \cdots) \\ &= (a_0 b_0) + (a_0 b_1 + a_1 b_0) z + (a_0 b_2 + a_1 b_1 + a_2 b_0) z^2 + \cdots; \end{aligned}$$

thus $G_1(z)G_2(z)$ is the generating function for the sequence $S_0, S_1, \ldots$, where

$$S_n = \sum_{0\leq k\leq n} a_k b_{n-k}. \tag{6}$$

Equation (3) is a very special case of this. Another important special case occurs when each b_n is equal to unity:

$$\frac{1}{1-z}G(z) = a_0 + (a_0 + a_1)z + (a_0 + a_1 + a_2)z^2 + \cdots . \tag{7}$$

Here we have the generating function for the sums of the original sequence.

The rule for a product of *three* functions follows from Eq. (6); $G_1(z)G_2(z)G_3(z)$ generates $S_0, S_1, \ldots$, where

$$S_n = \sum_{\substack{i,j,k\geq 0 \\ i+j+k=n}} a_i b_j c_k. \tag{8}$$

The general rule for products of *any number* of functions (whenever this is meaningful) is

$$\prod_{j\geq 0}\left(\sum_{k\geq 0} a_{jk}z^k\right) = \sum_{n\geq 0} z^n \left(\sum_{\substack{k_0,k_1,\ldots\geq 0 \\ k_0+k_1+\ldots=n}} a_{0k_0}a_{1k_1}\cdots\right). \tag{9}$$

When the recurrence relation for some sequence involves binomial coefficients, we often want to get a generating function for a sequence $c_0, c_1, \ldots$ defined by

$$c_n = \sum_k \binom{n}{k} a_k b_{n-k}. \tag{10}$$

In this case it is usually better to use generating functions for the sequences $\langle a_n/n!\rangle$, $\langle b_n/n!\rangle$, $\langle c_n/n!\rangle$, since we have

$$\left(\frac{a_0}{0!} + \frac{a_1}{1!}z + \frac{a_2}{2!}z^2 + \cdots\right)\left(\frac{b_0}{0!} + \frac{b_1}{1!}z + \frac{b_2}{2!}z^2 + \cdots\right) = \left(\frac{c_0}{0!} + \frac{c_1}{1!}z + \frac{c_2}{2!}z^2 + \cdots\right), \tag{11}$$

where c_n is given by Eq. (10).

D. Change of z. Clearly $G(cz)$ is the generating function for the sequence $a_0, ca_1, c^2a_2, \ldots$. In particular, the generating function for the sequence $1, c, c^2, c^3, \ldots$ is $1/(1 - cz)$.

There is a familiar trick for extracting alternate terms of a series:

$$\begin{aligned} \tfrac{1}{2}[G(z) + G(-z)] &= a_0 + a_2z^2 + a_4z^4 + \cdots \\ \tfrac{1}{2}[G(z) - G(-z)] &= a_1 + a_3z^3 + a_5z^5 + \cdots . \end{aligned} \tag{12}$$

Generally, using complex numbers, we can extract every mth term: Let $\omega = e^{2\pi i/m}$; we have

$$\sum_{k \bmod m = r} a_k z^k = \frac{1}{m}\sum_{1\leq j\leq m} \omega^{-jr}G(\omega^j z), \qquad 0 \leq r < m. \tag{13}$$

For example, if $m = 3$ and $r = 1$, we have $\omega = \cos 120° + i \sin 120°$ (a complex cube root of unity); it follows that

$$a_1 z + a_4 z^4 + a_7 z^7 + \cdots = \tfrac{1}{3}[G(z) + \omega^{-1} G(\omega z) + \omega^{-2} G(\omega^2 z)].$$

Proof is left to the reader (exercise 14).

E. Differentiation and integration. The techniques of calculus give us further operations. If $G(z)$ is given by Eq. (1), the derivative is

$$G'(z) = a_1 + 2a_2 z + 3a_3 z^2 + \cdots = \sum_{k \geq 0} (k+1) a_{k+1} z^k. \tag{14}$$

The generating function for the sequence $\langle n a_n \rangle$ is $zG'(z)$. Hence we can combine the nth term of a sequence with polynomials in n by manipulating the generating function.

Reversing the process, integration gives another useful operation:

$$\int_0^z G(t)\, dt = a_0 z + \tfrac{1}{2} a_1 z^2 + \tfrac{1}{3} a_2 z^3 + \cdots = \sum_{k \geq 1} \frac{1}{k} a_{k-1} z^k. \tag{15}$$

As special cases, we have the derivative and integral of (5):

$$\frac{1}{(1-z)^2} = 1 + 2z + 3z^2 + \cdots = \sum_{k \geq 0} (k+1) z^k. \tag{16}$$

$$\ln \frac{1}{1-z} = z + \tfrac{1}{2} z^2 + \tfrac{1}{3} z^3 + \cdots = \sum_{k \geq 1} \frac{1}{k} z^k. \tag{17}$$

We can combine the second formula with Eq. (7) to get the generating function for the harmonic numbers:

$$\frac{1}{1-z} \ln \frac{1}{1-z} = z + \tfrac{3}{2} z^2 + \tfrac{11}{6} z^3 + \cdots = \sum_{k \geq 0} H_k z^k. \tag{18}$$

F. Known generating functions. Whenever it is possible to determine the power series expansion of a function, we have implicitly found the generating function for a particular sequence. These special functions can be quite useful in conjunction with the operations described above.

The most important power series expansions are given in the following list.

i) *Binomial theorem*

$$(1+z)^r = 1 + rz + \frac{r(r-1)}{2} z^2 + \cdots = \sum_{k \geq 0} \binom{r}{k} z^k. \tag{19}$$

When r is a negative integer, we get a special case already reflected in Eqs. (5) and (16):

$$\frac{1}{(1-z)^{n+1}} = \sum_{k\geq 0} \binom{-n-1}{k} (-z)^k = \sum_{k\geq 0} \binom{n+k}{n} z^k. \tag{20}$$

There is also a generalization, which was proved in exercise 1.2.6–25:

$$x^r = 1 + rz + \frac{r(r-2t-1)}{2} z^2 + \cdots = \sum_{k\geq 0} \binom{r-kt}{k} \frac{r}{r-kt} z^k, \tag{21}$$

if $r \neq kt$ for positive integers k, and if x is the continuous function of z which solves the equation $x^{t+1} = x^t + z$, $x = 1$ when $z = 0$.

ii) *Exponential series*

$$e^z = 1 + z + \frac{1}{2!} z^2 + \cdots = \sum_{k\geq 0} \frac{1}{k!} z^k. \tag{22}$$

In general, we have the following formula involving Stirling numbers:

$$(e^z - 1)^n = z^n + \frac{1}{n+1} \left\{ {n+1 \atop n} \right\} z^{n+1} + \cdots = n! \sum_k \left\{ {k \atop n} \right\} z^k/k!. \tag{23}$$

iii) *Logarithm series*

$$\ln(1+z) = z - \tfrac{1}{2}z^2 + \tfrac{1}{3}z^3 - \cdots = \sum_{k\geq 1} \frac{(-1)^{k+1}}{k} z^k, \tag{24}$$

$$\ln\left(\frac{1}{1-z}\right) = z + \tfrac{1}{2}z^2 + \tfrac{1}{3}z^3 + \cdots = \sum_{k\geq 1} \frac{1}{k} z^k. \tag{25}$$

Using Stirling numbers (cf. Eq. 23), we have a more general equation:

$$\left(\ln\left(\frac{1}{1-z}\right)\right)^n = z^n + \frac{1}{n+1} \left[{n+1 \atop n} \right] z^{n+1} + \cdots = n! \sum_k \left[{k \atop n} \right] z^k/k!. \tag{26}$$

iv) *Miscellaneous*

$$z(z+1)\ldots(z+n-1) = \sum_k \left[{n \atop k} \right] z^k, \tag{27}$$

$$\frac{z^n}{(1-z)(1-2z)\ldots(1-nz)} = \sum_k \left\{ {k \atop n} \right\} z^k, \tag{28}$$

$$\frac{z}{e^z - 1} = 1 - \tfrac{1}{2}z + \tfrac{1}{12}z^2 + \cdots = \sum_{k\geq 0} \frac{B_k z^k}{k!}. \tag{29}$$

The coefficients B_k which appear in the last formula are the *Bernoulli numbers;* they will be examined further in Section 1.2.11.2, and a table of Bernoulli numbers appears in Appendix B.

Another identity, analogous to Eq. (21), is the following (see exercise 2.3.4.4–29):

$$x^r = 1 + rz + \frac{r(r+2t)}{2} z^2 + \cdots = \sum_{k\ge 0} \frac{r(r+kt)^{k-1}}{k!} z^k, \tag{30}$$

if x is the continuous function of z which solves the equation $x = e^{zxt}$, where $x = 1$ when $z = 0$.

We conclude this section by returning to a problem that was only partially solved in Section 1.2.3. We saw (Eq. 1.2.3–13 and exercise 1.2.3–29) that

$$\sum_{1\le i\le j\le n} x_i x_j = \frac{1}{2}\left(\left(\sum_{1\le k\le n} x_k\right)^2 + \left(\sum_{1\le k\le n} x_k^2\right)\right);$$

$$\sum_{1\le i\le j\le k\le n} x_i x_k x_j = \frac{1}{6}\left(\left(\sum_{1\le k\le n} x_k\right)^3 + 3\left(\sum_{1\le k\le n} x_k\right)\left(\sum_{1\le k\le n} x_k^2\right) + 2\left(\sum_{1\le k\le n} x_k^3\right)\right).$$

In general, suppose that we have n numbers $x_1, x_2, \ldots, x_n$ and we want the sum

$$a_m = \sum_{1\le j_1\le\ldots\le j_m\le n} x_{j_1} \ldots x_{j_m}.$$

If possible, this sum should be expressed in terms of $S_1, S_2, \ldots, S_m$, where

$$S_j = \sum_{1\le k\le n} x_k^j, \tag{31}$$

the sum of jth powers. Using this more compact notation, the above formulas become $a_2 = \frac{1}{2}S_1^2 + \frac{1}{2}S_2$; $a_3 = \frac{1}{6}S_1^3 + \frac{1}{2}S_1S_2 + \frac{1}{3}S_3$.

We can attack this problem by setting up the generating function

$$G(z) = 1 + a_1 z + a_2 z^2 + \cdots = \sum_{k\ge 0} a_k z^k. \tag{32}$$

By our rules for multiplying series, we find that

$$\begin{aligned} G(z) &= (1 + x_1 z + x_1^2 z^2 + \cdots) \cdots (1 + x_n z + x_n^2 z^2 + \cdots) \\ &= \frac{1}{(1 - x_1 z) \cdots (1 - x_n z)}. \end{aligned} \tag{33}$$

So $G(z)$ is the reciprocal of a polynomial. It often helps to take the logarithm

of a product, and we find that

$$\ln G(z) = \ln\left(\frac{1}{1-x_1z}\right) + \cdots + \ln\left(\frac{1}{1-x_nz}\right)$$
$$= \left(\sum_{k\ge1}\frac{x_1^kz^k}{k}\right) + \cdots + \left(\sum_{k\ge1}\frac{x_n^kz^k}{k}\right) = \sum_{k\ge1}\frac{S_kz^k}{k}.$$

Now $\ln G(z)$ has been expressed in terms of the S's, so all we must do to obtain the answer to our problem is to compute the power series expansion of $G(z)$ again:

$$G(z) = e^{\ln G(z)} = \exp\left(\sum_{k\ge1}\frac{S_kz^k}{k}\right) = \prod_{k\ge1} e^{S_kz^k/k}$$
$$= \left(1 + S_1z + \frac{S_1^2z^2}{2!} + \cdots\right)\left(1 + \frac{S_2z^2}{2} + \frac{S_2^2z^4}{2^2\cdot 2!} + \cdots\right)\cdots$$
$$= \sum_{m\ge0}\left(\sum_{\substack{k_1,k_2,\ldots,k_m\ge0\\ k_1+2k_2+\cdots+mk_m=m}} \frac{S_1^{k_1}}{1^{k_1}k_1!}\,\frac{S_2^{k_2}}{2^{k_2}k_2!}\cdots\frac{S_m^{k_m}}{m^{k_m}k_m!}\right)z^m. \tag{34}$$

The parenthesized quantity is a_m. This rather imposing sum is really not complicated when it is examined carefully. The number of terms for a particular value of m is $p(m)$, the number of partitions of m (cf. Section 1.2.1). For example, one partition of 12 is

$$12 = 1 + 2 + 2 + 2 + 5;$$

this corresponds to a solution of the equation $k_1 + 2k_2 + \cdots + 12k_{12} = 12$, where k_j is the number of j's in the partition. In our example $k_1 = 1$, $k_2 = 3$, $k_5 = 1$, and the other k's are zero; so we get the term

$$\frac{S_1}{1^11!}\,\frac{S_2^3}{2^33!}\,\frac{S_5}{5^11!} = \tfrac{1}{240}S_1S_2^3S_5$$

as part of the expression for a_{12}.

For a table of the coefficients appearing in Eq. (34), as well as a table of the similar coefficients in Faa di Bruno's formula (exercise 1.2.5–21), see *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun, (U.S. Gov't Printing Office, 1964), Table 24.2.

EXERCISES

1. [*M12*] What is the generating function for the sequence $2, 5, 13, 35, \ldots = \langle 2^n + 3^n \rangle$?

▶ 2. [*M13*] Prove Eq. (11).

3. [*HM21*] Differentiate the generating function (Eq. 18) for $\langle H_n \rangle$, and compare this with the generating function for $\langle \sum_{1 \le k \le n} H_k \rangle$. What relation can you deduce?

4. [*M01*] Explain why Eq. (19) is a special case of Eq. (21).

5. [*M20*] Prove Eq. (23) by induction on n.

▶ **6.** [*HM15*] Find the generating function for

$$\sum_{1\le k<n} \frac{1}{k(n-k)};$$

differentiate it and express the coefficients in terms of harmonic numbers.

7. [*M20*] Verify all the steps leading to Eq. (34).

8. [*M23*] Find the generating function for $p(n)$, the number of partitions of n.

9. [*M11*] In the notation of Eqs. (31) and (32), what is a_4 in terms of S_1, S_2, S_3, and S_4?

▶ **10.** [*M25*] An *elementary symmetric function* is defined by the formula

$$b_m = \sum_{1\le j_1<\cdots<j_m\le n} x_{j_1}\ldots x_{j_m}.$$

(This is the same as a_m of Eq. (32), except that equal subscripts are not allowed.) Find the generating function for b_m and then express b_m in terms of the S_j in Eq. (31). Write out the formulas for b_1, b_2, b_3, and b_4.

11. [*HM30*] Set up the generating function for the sequence $\langle n!\rangle$ and study properties of this function.

▶ **12.** [*M20*] Suppose that we have a doubly subscripted sequence a_{mn} for $m, n = 0, 1, \ldots$; show how this double sequence can be represented by a *single* generating function of two variables, and determine the generating function for the sequence $a_{mn} = \binom{n}{m}$.

13. [*HM22*] The "Laplace transform" of a function $f(x)$ is the function $\mathbf{L}f(s) = \int_0^\infty e^{-st}f(t)\,dt$. Given that $a_0, a_1, a_2, \ldots$ is an infinite sequence having a convergent generating function, let $f(x)$ be the step function $\sum_{0\le k\le x} a_k$. Express the Laplace transform of $f(x)$ in terms of the generating function G for this sequence.

14. [*HM21*] Prove Eq. (13).

15. [*M28*] By considering $H(w) = \sum_{n\ge 0} G_n(z)w^n$, find a "closed form" for the generating function

$$G_n(z) = \sum_{0\le k\le n} \binom{n-k}{k} z^k.$$

16. [*M22*] Give a simple formula for the generating function $G_{nr}(z) = \sum_k a_{nkr}z^k$, where a_{nkr} is the number of ways to choose k things out of n objects, subject to the condition that each object may be chosen at most r times. (If $r = 1$, we have $\binom{n}{k}$ ways, and if $r \ge k$, we have the number of combinations with repetitions (cf. exercise 1.2.6–60).)

17. [*M25*] What are the coefficients of $1/(1-z)^w$ if this function is expanded into a *double* power series in terms of both z and w?

▶ **18.** [*M25*] Given positive integers n and r, find a simple formula for the value of the following sums:

$$\text{(a)} \sum_{1\le k_1<k_2<\cdots<k_r\le n} k_1k_2\ldots k_r; \qquad \text{(b)} \sum_{1\le k_1\le k_2\le\cdots\le k_r\le n} k_1k_2\ldots k_r.$$

(For example, when $n = 3, r = 2$, the sums are $1\cdot 2+1\cdot 3+2\cdot 3$ and $1\cdot 1+1\cdot 2+1\cdot 3+2\cdot 2+2\cdot 3+3\cdot 3$, respectively.)

19. [*HM32*] (K. F. Gauss.) The sums of the following infinite series are well known:

$$1-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+\cdots=\ln 2; \qquad 1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots=\frac{\pi}{4};$$

$$1-\frac{1}{4}+\frac{1}{7}-\frac{1}{10}+\cdots=\frac{\pi}{3\sqrt{3}}+\frac{1}{3}\ln 2.$$

These series may be written respectively as

$$\frac{1}{2}\sum_{n\ge 0}\left(\frac{1}{n+\frac{1}{2}}-\frac{1}{n+1}\right); \qquad \frac{1}{4}\sum_{n\ge 0}\left(\frac{1}{n+\frac{1}{4}}-\frac{1}{n+1}\right)-\frac{1}{4}\sum_{n\ge 0}\left(\frac{1}{n+\frac{3}{4}}-\frac{1}{n+1}\right);$$

and

$$\frac{1}{6}\sum_{n\ge 0}\left(\frac{1}{n+\frac{1}{6}}-\frac{1}{n+1}\right)-\frac{1}{6}\sum_{n\ge 0}\left(\frac{1}{n+\frac{2}{3}}-\frac{1}{n+1}\right).$$

Prove that, in general, the series

$$\sum_{n\ge 0}\left(\frac{1}{n+p/q}-\frac{1}{n+1}\right)$$

has the value

$$\frac{\pi}{2}\cot\frac{p}{q}\pi+\ln 2q-2\sum_{0<k<q/2}\cos\frac{2pk}{q}\pi\cdot\ln\sin\frac{k}{q}\pi,$$

when p and q are integers with $0 < p < q$. [*Hint:* By Abel's limit theorem the sum is

$$\lim_{x\to 1-}\sum_{n\ge 0}\left(\frac{1}{n+p/q}-\frac{1}{n+1}\right)x^{p+nq}.$$

Use Eq. (13) to express this power series in such a way that the limit can be evaluated readily.]

1.2.10. Analysis of an Algorithm

Let us now apply some of the techniques of the preceding sections to the study of a typical algorithm.

Algorithm M (*Find the maximum*). Given n elements $X[1], X[2], \ldots, X[n]$, we will find m and j such that $m = X[j] = \max_{1 \le k \le n} X[k]$, and for which j is as large as possible.

M1. [Initialize.] Set $j \leftarrow n$, $k \leftarrow n - 1$, $m \leftarrow X[n]$.

M2. [All tested?] If $k = 0$, the algorithm terminates.

M3. [Compare.] If $X[k] \le m$, go to M5.

M4. [Change m.] Set $j \leftarrow k$, $m \leftarrow X[k]$. (Now m is the current maximum.)

M5. [Decrease k.] Decrease k by one, return to M2. ∎

This rather obvious algorithm may seem so trivial we should not bother to analyze it in detail; but it actually makes a good demonstration of the manner in which more complicated algorithms may be studied. Analysis of algorithms is quite important in computer programming, because there are usually several algorithms available for a particular application and we would like to know which is best.

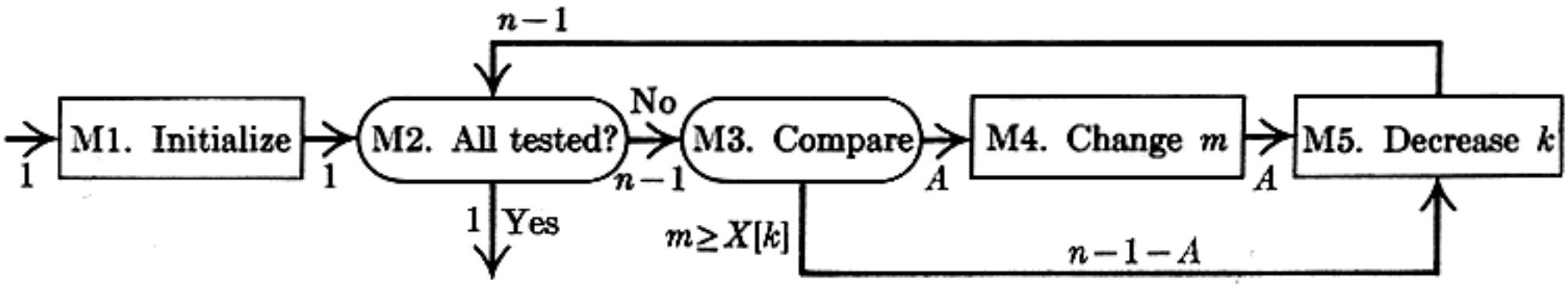


Fig. 9. Algorithm M. Labels on the arrows indicate the number of times each path is taken. Note that "Kirchhoff's law" must be satisfied, i.e., the amount of flow into each node must equal the amount of flow going out.

Algorithm M requires a fixed amount of storage, so we will analyze only the time required to perform it. To do this, we will *count the number of times each step is executed* (cf. Fig. 9):

Step number	Number of times
M1	1
M2	n
M3	$n - 1$
M4	A
M5	$n - 1$

Knowing the number of times each step is executed gives us the information necessary to determine the running time on a particular computer.

In the above table we know everything except the quantity A, which is the number of times we must change the value of the current maximum. To complete the analysis, we will study this interesting quantity A.

The analysis usually consists of finding the *minimum* value of A (for optimistic people), the *maximum* value of A (for pessimistic people), the *average* value of A (for probabilistic people), and the *standard deviation* of A (a quantitative indication of how close to the average we may expect the value to be).

The *minimum* value of A is zero; this happens if $X[n] = \max_{1\le k\le n} X[k]$. The *maximum* value is $n - 1$; this happens in case $X[1] > X[2] > \cdots > X[n]$.

Thus the average value lies between 0 and $n - 1$. Is it $\frac{1}{2}n$? Is it $\frac{1}{3}n$? To answer this question we need to define what we mean by the average; and to properly define the average, we must make some assumptions about the expected characteristics of the input data $X[1], X[2], \ldots, X[n]$. *We will assume that the $X[k]$ are distinct values, and that each of the $n!$ permutations of these values is equally likely.* (This is a reasonable assumption to make in most situations, but the analysis can be carried out under other assumptions, as shown in the exercises at the end of this section.)

The performance of Algorithm M does not depend on what the precise values of the $X[k]$ are; only the relative order is involved. For example, suppose that $n = 3$. We will say that each of the following six possibilities is equally probable:

Situation	Value of A	Situation	Value of A
$X[1] < X[2] < X[3]$	0	$X[2] < X[3] < X[1]$	1
$X[1] < X[3] < X[2]$	1	$X[3] < X[1] < X[2]$	1
$X[2] < X[1] < X[3]$	0	$X[3] < X[2] < X[1]$	2

The average value of A when $n = 3$ comes to $(0+1+0+1+1+2)/6 = 5/6$.

It is clear that we may take $X[1], X[2], \ldots, X[n]$ to be the numbers $1, 2, \ldots, n$ in some order; under our assumption we regard each of the $n!$ permutations as equally likely. The *probability* that A has the value k will be

$$p_{nk} = (\text{number of permutations of } n \text{ objects for which } A = k)/n! \qquad (1)$$

For example, from our table above, $p_{30} = \frac{1}{3}$, $p_{31} = \frac{1}{2}$, $p_{32} = \frac{1}{6}$.

The *average* ("mean") value is defined, as usual, to be

$$A_n = \sum_k k p_{nk}. \qquad (2)$$

The *variance* V_n is defined to be the average value of $(A - A_n)^2$; we have therefore

$$V_n = \sum_k (k - A_n)^2 p_{nk} = \sum_k k^2 p_{nk} - 2A_n \sum_k k p_{nk} + A_n^2 \sum_k p_{nk}$$
$$= \sum_k k^2 p_{nk} - 2A_n A_n + A_n^2 = \sum_k k^2 p_{nk} - A_n^2. \qquad (3)$$

Finally, the *standard deviation* σ_n is defined to be $\sqrt{V_n}$.

We can determine the behavior of A by determining the probabilities p_{nk}. It is not hard to do this inductively: by Eq. (1) we want to count the numbers of permutations on n elements that have $A = k$.

Consider the permutations $x_1 x_2 \ldots x_n$ on $\{1, 2, \ldots, n\}$ (cf. Section 1.2.5). If $x_1 = n$, the value of A is *one higher* than the value obtained on $x_2 \ldots x_n$; if $x_1 \neq n$, the value of A is *exactly the same* as its value on $x_2 \ldots x_n$. Therefore we find that

$$p_{nk} = \frac{1}{n} p_{(n-1)(k-1)} + \frac{n-1}{n} p_{(n-1)k}. \tag{4}$$

This equation will determine p_{nk} if we provide the initial conditions

$$p_{1k} = \delta_{0k}; \qquad \text{and} \qquad p_{nk} = 0 \quad \text{if} \quad k < 0. \tag{5}$$

We can now get information about the quantities p_{nk} by using generating functions. Let

$$G_n(z) = p_{n0} + p_{n1} z + \cdots = \sum_k p_{nk} z^k. \tag{6}$$

(We know that $A \leq n - 1$, so $p_{nk} = 0$ for large values of k; thus $G_n(z)$ is actually a polynomial, even though an infinite sum has been specified for convenience.)

From Eq. (5) we have $G_1(z) = 1$; and from Eq. (4) we have

$$G_n(z) = \frac{z}{n} G_{n-1}(z) + \frac{n-1}{n} G_{n-1}(z) = \frac{z+n-1}{n} G_{n-1}(z). \tag{7}$$

(The reader should study the relation between Eqs. (4) and (7) carefully.) We can now see that

$$\begin{aligned} G_n(z) = \frac{z+n-1}{n} G_{n-1}(z) &= \frac{z+n-1}{n} \frac{z+n-2}{n-1} G_{n-2}(z) = \cdots \\ &= \frac{1}{n!} (z+n-1)(z+n-2) \ldots (z+1) \\ &= \frac{1}{z+n} \binom{z+n}{n}. \end{aligned} \tag{8}$$

So $G_n(z)$ is essentially a binomial coefficient!

This function appears in the previous section (Eq. 1.2.9–27), where we have

$$G_n(z) = \frac{1}{n!} \sum_k \begin{bmatrix} n \\ k \end{bmatrix} z^{k-1}.$$

Therefore p_{nk} can be expressed in terms of Stirling numbers:

$$p_{nk} = \begin{bmatrix} n \\ k+1 \end{bmatrix} \Big/ n! \tag{9}$$

Figure 10 shows the approximate sizes of p_{nk} when $n = 12$.

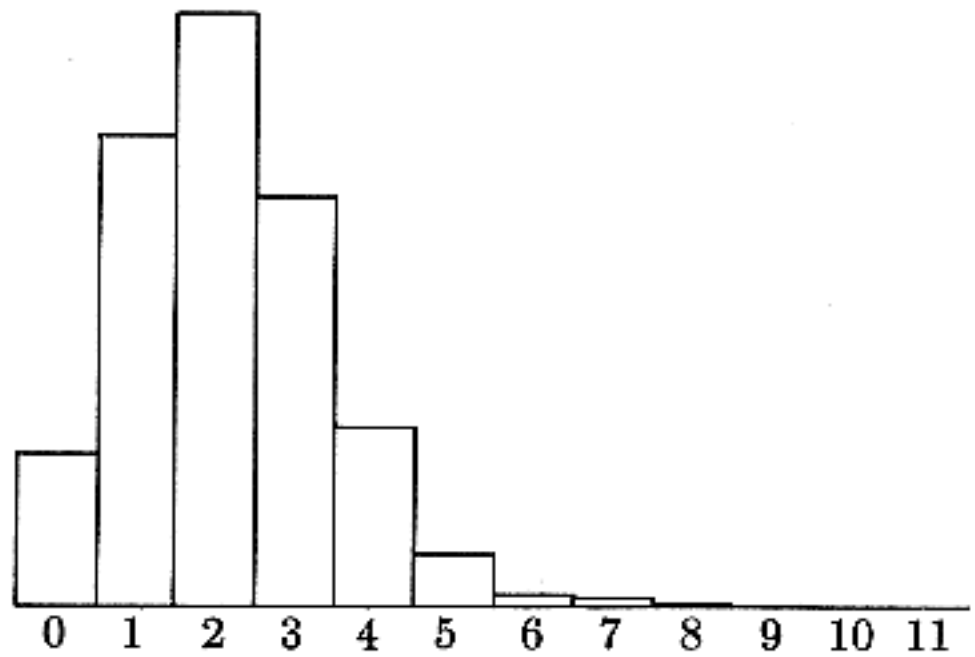


Fig. 10. Probability distribution for step M4, when $n = 12$. The average is $58301/22720 \approx 2.11$.

Now all we must do is plug this value of p_{nk} into Eqs. (2) and (3) and we have the desired average value. But this is easier said than done. It is, in fact, unusual to be able to determine the probabilities p_{nk} explicitly; in most problems we will know the generating function $G_n(z)$, but we will not have any special knowledge about the actual probabilities. The important fact is that *we can determine the mean and variance easily from the generating function itself.*

To see this, let us suppose that we have a generating function whose coefficients represent probabilities:

$$G(z) = p_0 + p_1 z + p_2 z^2 + \cdots .$$

Here p_k is the probability that some event has a value k. We wish to calculate the quantities

$$\text{mean } (G) = \sum_k k p_k, \qquad \text{var } (G) = \sum_k k^2 p_k - (\text{mean } (G))^2. \tag{10}$$

Using differentiation, it is not hard to discover how to do this. Note that

$$G(1) = 1, \tag{11}$$

since $G(1) = p_0 + p_1 + p_2 + \cdots$ is the sum of all possible probabilities. Similarly, since $G'(z) = \sum k p_k z^{k-1}$, we have

$$\text{mean } (G) = \sum_k k p_k = G'(1). \tag{12}$$

Finally, we apply differentiation again and we obtain (see exercise 2)

$$\text{var } (G) = G''(1) + G'(1) - G'(1)^2. \tag{13}$$

Equations (12) and (13) give the desired expressions of the mean and variance in terms of the generating function.

In our case, we wish to calculate $G_n'(1) = A_n$. From Eq. (7) we have

$$G_n'(z) = \frac{1}{n} G_{n-1}(z) + \frac{z+n-1}{n} G_{n-1}'(z);$$

$$G_n'(1) = \quad \frac{1}{n} \quad + G_{n-1}'(1).$$

From the initial condition $G_1'(1) = 0$, we find therefore

$$A_n = G_n'(1) = H_n - 1. \tag{14}$$

This is the desired average number of times step M4 is executed; it is approximately $\ln n$ when n is large. [*Note:* The rth moment is the coefficient of z^n in $1/(1-z) \sum_k \{{r \atop k}\} \ln (1/(1-z)^k)$, and it has the approximate value $(\ln n)^r$; see *CACM* **9** (1966), 342.]

We can proceed similarly to calculate the variance V_n. Before doing this, let us state an important simplification:

Theorem A. *Let G, H be two generating functions with $G(1) = H(1) = 1$. If the quantities* mean (G), var (G) *are defined by Eqs.* (12), (13), *we have*

$$\text{mean } (GH) = \text{mean } (G) + \text{mean } (H); \quad \text{var } (GH) = \text{var } (G) + \text{var } (H). \tag{15}$$

We will prove this theorem later. It tells us that the mean and variance of a product of generating functions may be reduced to a sum. ∎

Letting $Q_n(z) = (z+n-1)/n$, we have $Q_n'(1) = 1/n$, $Q_n''(1) = 0$; hence

$$\text{mean } (Q_n) = \frac{1}{n}, \qquad \text{var } (Q_n) = \frac{1}{n} - \frac{1}{n^2}.$$

Finally, since $G_n(z) = \prod_{2 \le k \le n} Q_k(z)$, it follows that

$$\text{mean } (G_n) = \sum_{2 \le k \le n} \text{mean } (Q_k) = \sum_{2 \le k \le n} \frac{1}{k} = H_n - 1$$

$$\text{var } (G_n) = \sum_{2 \le k \le n} \text{var } (Q_k) = \sum_{1 \le k \le n} \left(\frac{1}{k} - \frac{1}{k^2} \right) = H_n - H_n^{(2)}.$$

Summing up, we have found the desired statistics related to quantity A:

$$A = (\min 0, \quad \text{ave } H_n - 1, \quad \max n - 1, \quad \text{dev } \sqrt{H_n - H_n^{(2)}}). \tag{16}$$

The notation used in Eq. (16) will be used to describe the statistical characteristics of other probabilistic quantities throughout this book.

We have completed the analysis of Algorithm M; the new feature that has appeared in this analysis is the introduction of probability theory. Not much probability theory is required for most of the applications in this book: the

simple counting techniques and the definitions of mean, variance, and standard deviation which have already been given will suffice.

Let us consider some simple probability problems, to get a little more practice using these methods. In all probability the first problem that comes to mind is a coin-tossing problem. Suppose we flip a coin n times and there is a probability p that "heads" turns up at each toss; what is the average number of heads which will occur? What is the standard deviation?

We will consider our coin to be biased, i.e., we will not assume that $p = \frac{1}{2}$. This makes the problem more interesting, and, furthermore, every real coin is biased (else we could not tell one side from the other!).

Proceeding as before, we let p_{nk} be the probability that k heads will occur, and let $G_n(z)$ be the corresponding generating function. We have clearly

$$p_{nk} = p \cdot p_{(n-1)(k-1)} + q \cdot p_{(n-1)k}. \tag{17}$$

Here, $q = 1 - p$ is the probability that "tails" turns up at each toss. As before, we argue from Eq. (17) that $G_n(z) = (q + pz)G_{n-1}(z)$; and from the obvious initial condition that $G_1(z) = q + pz$ we have

$$G_n(z) = (q + pz)^n. \tag{18}$$

Hence

$$\text{mean } (G_n) = n \text{ mean } (G_1) = pn;$$
$$\text{var } (G_n) = n \text{ var } (G_1) = (p - p^2)n = pqn.$$

For the number of heads, we have therefore

$$(\min 0, \quad \text{ave } pn, \quad \max n, \quad \text{dev } \sqrt{pqn}). \tag{19}$$

Figure 11 shows the values of p_{nk} when $p = \frac{3}{5}$, $n = 12$. When the standard deviation is proportional to $\sqrt{n}$ and the difference between maximum and minimum is proportional to n, we may consider the situation "stable" about the average.

Let us work one more simple problem. Suppose that in some process there is *equal* probability of obtaining the values $1, 2, \ldots, n$. The generating function for this situation is

$$G(z) = \frac{1}{n}z + \frac{1}{n}z^2 + \cdots + \frac{1}{n}z^n = \frac{1}{n}\frac{z^{n+1} - z}{z - 1}. \tag{20}$$

We find after some rather laborious calculation that

$$G'(z) = \frac{nz^{n+1} - (n+1)z^n + 1}{n(z-1)^2},$$

$$G''(z) = \frac{n(n-1)z^{n+1} - 2(n+1)(n-1)z^n + n(n+1)z^{n-1} - 2}{n(z-1)^3}. \tag{21}$$

Now to calculate the mean and variance, we need to know $G'(1)$ and $G''(1)$; but the form in which we have expressed these equations reduces to 0/0 when

we substitute $z = 1$. This makes it necessary to find the limit as z approaches unity, and that is a nontrivial task (cf. exercise 6). Here we have a case where it is much easier to compute from the probabilities directly, rather than derive mean and variance from the generating function. The statistics in this case are

$$\left(\min\ 1, \qquad \text{ave}\ \frac{n+1}{2}, \qquad \max\ n, \qquad \text{dev}\ \sqrt{\frac{(n+1)(n-1)}{12}}\right). \tag{22}$$

In this case the deviation of approximately $0.289n$ gives us a recognizably *unstable* situation.

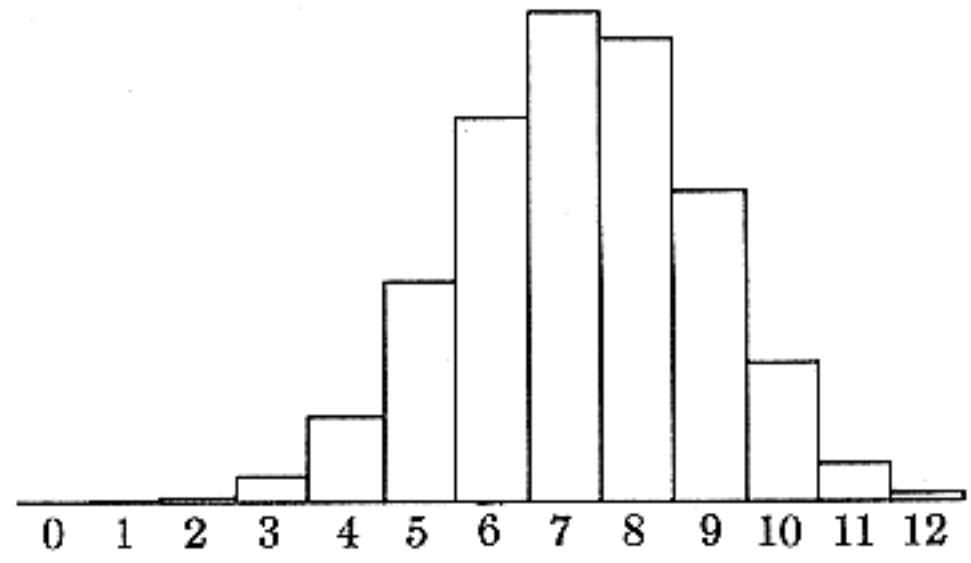


Fig. 11. Probability distribution for coin-tossing; 12 tosses with a chance of success equal to $\frac{3}{5}$ at each toss.

We conclude this section by proving Theorem A and relating our notions to classical probability theory. When $G(z) = p_0 + p_1 z + p_2 z^2 + \cdots$ represents a probability distribution for some quantity X, that is, if p_k is the probability that $X = k$, and X takes on only nonnegative integral values, we have $p_k \geq 0$ and $G(1) = 1$. The quantity $G(e^{it}) = p_0 + p_1 e^{it} + p_2 e^{2it} + \cdots$ is conventionally called the *characteristic function* of this distribution. The distribution given by the product of two such generating functions is called the *convolution* of the two distributions, and it represents the sum of two independent random variables belonging to those distributions.

The mean and variance are just two of the so-called *semi-invariants* or *cumulants* introduced by Thiele in 1903. The semi-invariants $\kappa_1, \kappa_2, \kappa_3, \ldots$ are defined by the rule

$$\frac{\kappa_1 t}{1!} + \frac{\kappa_2 t^2}{2!} + \frac{\kappa_3 t^3}{3!} + \cdots = \ln G(e^t). \tag{23}$$

We have

$$\kappa_n = \frac{d^n}{dt^n} \ln G(e^t)\Big|_{t=0};$$

in particular,

$$\kappa_1 = \frac{e^t G'(e^t)}{G(e^t)}\Big|_{t=0} = G'(1),$$

and

$$\kappa_2 = \frac{e^{2t} G''(e^t)}{G(e^t)} + \frac{e^t G'(e^t)}{G(e^t)} - \frac{e^{2t} G'(e^t)^2}{G(e^t)^2}\Big|_{t=0} = G''(1) + G'(1) - G'(1)^2.$$

Since the semi-invariants are defined in terms of the *logarithm* of a generating function, Theorem A is obvious, and, in fact, it can be generalized to apply to all of the semi-invariants.

A *normal distribution* is one for which all semi-invariants are zero except the mean and variance. In a normal distribution, the difference between a random value and its mean is less than the standard deviation

$$\frac{1}{\sqrt{2\pi}} \int_{-1}^{+1} e^{-t^2/2}\, dt = 68.2689492137\,09\%$$

of the time. The difference is less than twice the standard deviation 95.449973610364% of the time, and it is less than three times the standard deviation 99.730020393674% of the time. Both of the distributions specified by Eqs. (8) and (18) are *approximately* normal when n is large (see exercises 13 and 14).

EXERCISES

1. [*10*] Determine the value of p_{n0} from Eqs. (4) and (5) and, considering Algorithm M, interpret this result.

2. [*HM16*] Derive Eq. (13) from Eq. (10).

3. [*M15*] What are the minimum, maximum, average, and standard deviation of the number of times step M4 is executed, if we are using Algorithm M to find the maximum of 1000 randomly ordered, distinct items? (Give your answer as decimal approximations to these quantities.)

4. [*M10*] Give an explicit, closed formula for the values of p_{nk} in the coin-tossing experiment, Eq. (17).

5. [*M13*] What are the mean and the standard deviation of the distribution shown in Fig. 11?

6. [*HM23*] Use L'Hospital's rule to find $G'(1)$ and $G''(1)$ from Eqs. (21).

▶ **7.** [*M27*] In our analysis of Algorithm M, we assumed that all the $X[k]$ were distinct. Suppose, instead, that we make only the weaker assumption that $X[1], X[2], \ldots, X[n]$ contain precisely m distinct values; the values are otherwise random, subject to this constraint. What is the probability distribution of A in this case?

▶ **8.** [*M20*] Suppose that each $X[k]$ is taken at random from a set of M distinct elements, so that each of the M^n possible choices for $X[1], X[2], \ldots, X[n]$ is considered equally likely. What is the probability that all the $X[k]$ will be distinct?

9. [*M25*] Generalize the result of the preceding exercise to find a formula for the probability that exactly m distinct values occur among the X's. Express your answer in terms of Stirling numbers.

10. [*M20*] Combine the results of the preceding three exercises to obtain a formula for the probability that $A = k$ under the assumption that each X is selected at random from a set of M objects.

11. [*HM20*] Given that $G(z) = p_0 + p_1 z + p_2 z^2 + \cdots$ represents a probability distribution, let $M_n = \sum_k k^n p_k$. (M_n is called the "nth *moment*.") Show that $G(e^t) = 1 + M_1 t + M_2 t^2/2! + \cdots$; then using Faa di Bruno's formula (exercise 1.2.5–21), show that

$$\kappa_n = \sum_{\substack{k_1,\cdots,k_n \geq 0 \\ k_1+2k_2+\cdots=n}} \frac{n!(k_1+k_2+\cdots+k_n-1)!(-1)^{k_1+\cdots+k_n-1}}{k_1!(1!)^{k_1}\ldots k_n!(n!)^{k_n}} M_1^{k_1}\ldots M_n^{k_n}.$$

(In particular, $\kappa_1 = M_1$, $\kappa_2 = -M_1^2 + M_2$ (as we already know), $\kappa_3 = 2M_1^3 - 3M_1M_2 + M_3$, $\kappa_4 = -6M_1^4 - 3M_2^2 + 12M_1^2M_2 - 4M_1M_3 + M_4$.)

▸ **12.** [*M15*] What happens to the semi-invariants of a distribution if we change $G(z)$ to $G_1(z) = z^n G(z)$?

13. [*HM38*] A sequence of characteristic functions $G_n(z)$ with means μ_n and deviations σ_n is said to *approach a normal distribution* if

$$\lim_{n\to\infty} e^{-t\mu_n/\sigma_n} G_n(e^{t/\sigma_n}) = e^{t^2/2}$$

for all *imaginary* values of t, that is, whenever $t = ui$ for a real number u. Using $G_n(z)$ as given by Eq. (8), show that $G_n(z)$ approaches a normal distribution. (This is a theorem of Goncharov, *Izv. Akad. Nauk SSSR Ser. Math.* **8** (1944).)

Note: "Approaching the normal distribution," as defined here, can be shown to be equivalent to the fact that

$$\lim_{n\to\infty} \text{probability}\left(\frac{X_n - \mu_n}{\sigma_n} \leq x\right) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2}\, dt,$$

where X_n is a random quantity whose probabilities are specified by $G_n(z)$. This is a special case of P. Levy's important "continuity theorem," which is a basic result in mathematical probability theory; a proof of the result would take us rather far afield, although it is not extremely difficult [for example, see *Limit Distributions for Sums of Independent Random Variables* by B. V. Gnedenko and A. N. Kolmogorov, tr. by K. L. Chung (Reading, Mass.: Addison-Wesley, 1954)].

14. [*HM30*] (A. de Moivre.) Using the conventions of the previous exercise, show that the binomial distribution $G_n(z)$ given by Eq. (18) approaches the normal distribution.

▸ **15.** [*M21*] Let z be a positive number. What is the average value of the quantity z^A taken over all permutations of order n, if A is the quantity appearing in the analysis of Algorithm M?

16. [*HM23*] When the probability that some quantity has the value k is $e^{-\mu}(\mu^k/k!)$, it is said to have the "Poisson distribution with mean μ."

a) What is the generating function for this set of probabilities?
b) What are the values of the semi-invariants?
c) Show that as $n \to \infty$ the Poisson distribution with mean np approaches the normal distribution in the sense of exercise 13.

▸ **17.** [*M27*] Let $f(z)$ and $g(z)$ be generating functions which represent probability distributions.

a) Show that $h(z) = g\big(f(z)\big)$ is also a generating function representing a probability distribution.

b) Interpret the significance of $h(z)$ in terms of $f(z)$ and $g(z)$. (What is the *meaning* of the probabilities represented by the coefficients of $h(z)$?)

c) Give formulas for the mean and variance of h in terms of those for f, g.

18. [*M28*] Suppose that the distinct values taken on by $X[1], X[2], \ldots, X[n]$ in Algorithm M include exactly k_1 ones, k_2 twos, $\ldots$, k_n n's, arranged in random order. (Here

$$k_1 + k_2 + \cdots + k_n = n.$$

Note that the text's assumption is $k_1 = k_2 = \cdots = k_n = 1$.) Show that in this generalized situation, the generating function, Eq. (8), becomes

$$\left(\frac{k_{n-1}z + k_n}{k_{n-1} + k_n}\right)\left(\frac{k_{n-2}z + k_{n-1} + k_n}{k_{n-2} + k_{n-1} + k_n}\right) \cdots \left(\frac{k_1 z + k_2 + \cdots + k_n}{k_1 + k_2 + \cdots + k_n}\right),$$

using the convention $0/0 = 1$.

*1.2.11. Asymptotic Representations

We often want to find the approximate value of a quantity, instead of an exact value, in order to compare one number to another. For example, Stirling's approximation to $n!$ is a useful representation of this type, and we also have made use of the fact that $H_n \approx \ln n + \gamma$.

The derivations of such "asymptotic" formulas generally involve higher mathematics, although in the following subsections we use nothing more than elementary calculus to get the results we need.

1.2.11.1. The *O*-notation. A very convenient notation for dealing with approximations was introduced by P. Bachmann in the book *Analytische Zahlentheorie* in 1892. This is the "big-oh" notation which allows us to replace the "$\approx$" sign by "$=$", for example,

$$H_n = \ln n + \gamma + O\left(\frac{1}{n}\right). \tag{1}$$

(Read, "H sub n equals the natural log of n plus Euler's ("Oiler's") constant plus big O of one over n.")

In general, the notation $O(f(n))$ may be used whenever $f(n)$ is a function of the positive integer n; it stands for *a quantity which is not explicitly known*, except that its magnitude isn't too large. Every appearance of $O(f(n))$ means precisely this: there is a positive constant M such that the number x_n represented by $O(f(n))$ satisfies the condition $|x_n| \le M|f(n)|$. We do not say *what* the constant M is, and indeed the constant M is usually different for each appearance of O.

For example, Eq. (1) means that $|H_n - \ln n + \gamma| \le M/n$; the constant M is not specified further, but even if we don't know its value, we do know that the quantity $O(\frac{1}{n})$ will be arbitrarily small if n is large enough.

Let us give some more examples. We know that

$$1^2 + 2^2 + \cdots + n^2 = \tfrac{1}{3}n(n + \tfrac{1}{2})(n + 1) = \tfrac{1}{3}n^3 + \tfrac{1}{2}n^2 + \tfrac{1}{6}n,$$

so it follows that

$$1^2 + 2^2 + \cdots + n^2 = O(n^3), \tag{2}$$

$$1^2 + 2^2 + \cdots + n^2 = \tfrac{1}{3}n^3 + O(n^2). \tag{3}$$

Equation (3) is a stronger statement than Eq. (2). To justify these equations we shall prove that *if* $P(n) = a_0 + a_1 n + \cdots + a_m n^m$ *is any polynomial of degree* m *or less,* $P(n) = O(n^m)$. This follows because

$$\begin{aligned} |P(n)| \le |a_0| + |a_1|n + \cdots + |a_m|n^m &= (|a_0|/n^m + |a_1|/n^{m-1} + \cdots + |a_m|)n^m \\ &\le (|a_0| + |a_1| + \cdots + |a_m|)n^m, \end{aligned}$$

when $n \ge 1$. So we may take $M = |a_0| + \cdots + |a_m|$.

The O-notation is a big help in approximation work, since it briefly describes a concept which occurs frequently and it suppresses detailed information which is usually irrelevant. Furthermore, it can be manipulated algebraically in familiar ways, provided that a little bit of caution is used.

Many of the rules of algebra can be used together with O-notations, but certain important differences should be mentioned. The most important consideration is the idea of *one-way equalities:* We write $\frac{1}{2}n^2 + n = O(n^2)$, but we *never* write $O(n^2) = \frac{1}{2}n^2 + n$. (Or else, since $\frac{1}{4}n^2 = O(n^2)$, we might come up with the absurd relation $\frac{1}{4}n^2 = \frac{1}{2}n^2 + n$.) We always use the convention that *the right-hand side of an equation does not give more information than the left-hand side;* the right-hand side is a "crudification" of the left.

This convention about the use of "=" may be stated more precisely as follows: "If $\alpha(n)$ and $\beta(n)$ are formulas which involve the $O(f(n))$-notation, then $\alpha(n) = \beta(n)$ means that for any positive constants $M_1, M_2, \ldots, M_k$—one for each appearance of O in $\alpha(n)$—there exist positive constants $M_{k+1}, \ldots, M_m$—one for each appearance of O in $\beta(n)$—such that, for any given value of n, $\alpha(n)$ and $\beta(n)$ represents *sets* of augmented real numbers $S(\alpha(n))$ and $S(\beta(n))$, respectively, for which $S(\alpha(n)) \subseteq S(\beta(n))$." By augmented real numbers, we mean real numbers plus the additional quantity Ω which denotes an undefined value [such as $1/0$, $\lim_{x\to 0} \sin(1/x)$, $\Omega - \Omega$, etc.]. The sets $S(\alpha(n))$ and $S(\beta(n))$ are obtained by considering all possible values x_j such that $|x_j| \le M_j|f(n)|$ substituted for the jth appearance of O. It follows that we may perform most of the operations we are accustomed to doing with the "=" sign: If $\alpha(n) = \beta(n)$ and $\beta(n) = \gamma(n)$, then $\alpha(n) = \gamma(n)$. Also, if $\alpha(n) = \beta(n)$ and if $\delta(n)$ is a formula resulting from the substitution of $\beta(n)$ for some occurrence of $\alpha(n)$ in a formula $\gamma(n)$, then $\gamma(n) = \delta(n)$. These two statements imply, for example, that if $g(x_1, x_2, \ldots, x_m)$ is any real function whatever, and if $\alpha_k(n) = \beta_k(n)$ for $1 \le k \le m$, then $g(\alpha_1(n), \alpha_2(n), \ldots, \alpha_m(n)) = g(\beta_1(n), \beta_2(n), \ldots, \beta_m(n))$.

Here are some of the simple operations we can do with the O-notation:

$$f(n) = O(f(n)), \tag{4}$$

$$c \cdot O(f(n)) = O(f(n)), \qquad \text{if } c \text{ is a constant}, \tag{5}$$

$$O(f(n)) + O(f(n)) = O(f(n)), \tag{6}$$

$$O(O(f(n))) = O(f(n)), \tag{7}$$

$$O(f(n))O(g(n)) = O(f(n)g(n)), \tag{8}$$

$$O(f(n)g(n)) = f(n)O(g(n)). \tag{9}$$

The O-notation is also used more generally with functions of a real variable x. A particular range of values of x is specified, for example, $a \le x \le b$, and we write $O(f(x))$ to stand for any quantity $g(x)$, such that $|g(x)| \le M|f(x)|$ whenever $a \le x \le b$. (As before, M is an unspecified constant.) The notation $O(f(n))$ discussed above is the special case where the variable x is restricted to positive integer values; we usually call the variable n instead of x in this case.

Suppose that $g(x)$ is a function given by an infinite series,

$$g(x) = \sum_{k \ge 0} a_k x^k, \qquad |x| \le r,$$

where the sum of absolute values $\sum_{k \le 0} |a_k x^k|$ also exists. We can then always write

$$g(x) = a_0 + a_1 x + \cdots + a_m x^m + O(x^{m+1}), \qquad |x| \le r. \tag{10}$$

For, $g(x) = a_0 + a_1 x + \cdots + a_m x^m + x^{m+1}(a_{m+1} + a_{m+2}x + \cdots)$; we must only show that the parenthesized quantity is bounded when $|x| \le r$, and it is easy to show that $|a_{m+1}| + |a_{m+2}|r + |a_{m+3}|r^2 + \cdots$ is an upper bound.

For example, consider the generating functions given in Section 1.2.9; we have the important relations

$$e^x = 1 + x + \frac{1}{2!}x^2 + \cdots + \frac{1}{m!}x^m + O(x^{m+1}),$$

$$|x| \le r, \qquad \text{any fixed } r; \tag{11}$$

$$\ln(1+x) = x - \tfrac{1}{2}x^2 + \cdots + \frac{(-1)^{m+1}}{m}x^m + O(x^{m+1}),$$

$$|x| \le r, \qquad \text{any fixed } r < 1; \tag{12}$$

$$(1+x)^\alpha = 1 + \alpha x + \binom{\alpha}{2}x^2 + \cdots + \binom{\alpha}{m}x^m + O(x^{m+1}),$$

$$|x| \le r, \qquad \text{any fixed } r < 1. \tag{13}$$

The statement that r is "fixed" means that r must have a definite value when the O-notation is used. We obviously have $e^x = O(1)$ when $|x| \le r$, since $|e^x| \le e^r$, but the constant M implied by the O-notation depends on r. In fact,

it is easy to see that if x is allowed to range over all values $-\infty < x < \infty$, then $e^x \neq O(x^m)$ for *any* m.

Let us give one simple example of the concepts we have introduced so far. Consider the quantity $\sqrt[n]{n}$; as n gets large, the operation of taking an nth root tends to decrease the value, but it is not immediately obvious whether $\sqrt[n]{n}$ decreases or increases. It turns out that $\sqrt[n]{n}$ decreases to unity. Let us consider the slightly more complicated quantity $n(\sqrt[n]{n} - 1)$. Now $(\sqrt[n]{n} - 1)$ gets smaller as n gets bigger; what happens to $n(\sqrt[n]{n} - 1)$?

This problem is rather easily solved by applying the above formulas. We have

$$\sqrt[n]{n} = e^{\ln n/n} = 1 + (\ln n/n) + O((\ln n/n)^2). \tag{14}$$

This equation proves our previous contention that $\sqrt[n]{n} \to 1$. Furthermore, it tells us that

$$\begin{aligned} n(\sqrt[n]{n} - 1) &= n(\ln n/n + O((\ln n/n)^2) \\ &= \ln n + O((\ln n)^2/n). \end{aligned}$$

So we find that $n(\sqrt[n]{n} - 1)$ is approximately equal to $\ln n$; the difference is $O((\ln n)^2/n)$, which approaches zero as n approaches infinity (see exercise 8).

EXERCISES

1. [*HM01*] What is $\lim_{n\to\infty} O(n^{-1/3})$?

▶ **2.** [*M10*] Mr. B. C. Dull obtained astonishing results by using the formula $O(f(n)) - O(f(n)) = 0$; what is his mistake, and what should the right-hand side of his formula be?

3. [*M15*] Multiply $(\ln n + \gamma + O(1/n))$ by $(n + O(\sqrt{n}))$, and express your answer in O-notation.

4. [*M18*] Suppose that $f(n) \neq 0$ for all $n > 0$. Show that if $|x_n| \leq M|f(n)|$ for all $n \geq n_0$, where n_0 is some integer, then $x_n = O(f(n))$. (This is stated differently than the text's definition, which takes $n_0 = 1$.)

5. [*M20*] (a) Given that $r > 0$ and $P(x) = c_0 + c_1 x + \cdots + c_m x^m$, show that $P(x) = O(x^m)$, when $x \geq r$. (b) Prove or disprove: $P(x) = O(x^m)$, when $x > 0$.

▶ **6.** [*M20*] What is wrong with the following argument? "Since $n = O(n)$, $2n = O(n)$, . . . , we have

$$\sum_{1\leq k\leq n} kn = \sum_{1\leq k\leq n} O(n) = O(n^2)."$$

7. [*HM15*] Prove that if the values of x are allowed to be arbitrarily large, $e^x \neq O(x^m)$ for any power m.

8. [*HM20*] Prove that as $n \to \infty$, $(\ln n)^m/n \to 0$.

9. [*HM20*] Show that $e^{O(x^m)} = 1 + O(x^m)$, $|x| \leq r$.

10. [*HM22*] Make a statement similar to that in the previous exercise about $\ln\left(1 + O(x^m)\right)$.

11. [*M11*] Explain why Eq. (14) is true.

▶ **12.** [*M15*] Give an asymptotic expansion of $n(\sqrt[n]{a} - 1)$, if $a > 0$, to terms $O(1/n^3)$.

1.2.11.2. Euler's summation formula. Perhaps the most useful method for obtaining good approximations is the one due to Leonhard Euler in 1732; his method approximates a finite sum by an integral, and gives us a means to get better and better approximations in many cases.

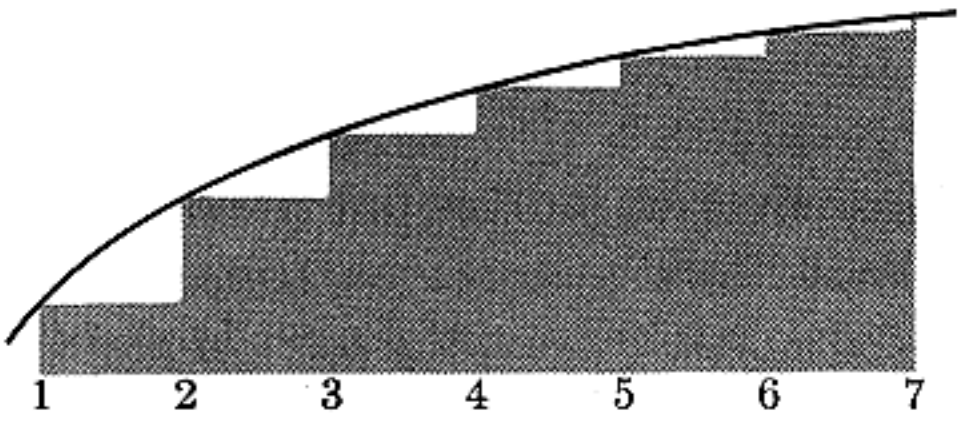


Fig. 12. Comparing a sum with an integral.

Figure 12 shows a comparison of $\int_1^n f(x)\,dx$ and $\sum_{1\le k<n} f(k)$, $n = 7$. Euler's method gives a useful formula for the difference between these two quantities, assuming that $f(x)$ is a differentiable function.

For convenience we use the notation

$$\{x\} = x \bmod 1 = x - \lfloor x \rfloor. \qquad (1)$$

Our derivation starts with the following identity:

$$\int_k^{k+1} (\{x\} - \tfrac{1}{2})f'(x)\,dx = (x - k - \tfrac{1}{2})f(x)\Big|_k^{k+1} - \int_k^{k+1} f(x)\,dx$$
$$= \tfrac{1}{2}\big(f(k+1) + f(k)\big) - \int_k^{k+1} f(x)\,dx. \qquad (2)$$

(This follows from integration by parts.) Adding both sides of this equation for $1 \le k < n$, we find that

$$\int_1^n (\{x\} - \tfrac{1}{2})f'(x)\,dx = \sum_{1\le k<n} f(k) + \tfrac{1}{2}\big(f(n) - f(1)\big) - \int_1^n f(x)\,dx,$$

that is,

$$\sum_{1\le k<n} f(k) = \int_1^n f(x)\,dx - \tfrac{1}{2}\big(f(n) - f(1)\big) + \int_1^n B_1(\{x\})f'(x)\,dx, \qquad (3)$$

where $B_1(x)$ is the polynomial $x - \frac{1}{2}$. This is the desired connection between the sum and the integral.

The approximation can be carried further if we continue to integrate by parts. Before doing this, however, we shall discuss the *Bernoulli numbers*, which

are the coefficients in the following infinite series:

$$\frac{x}{e^x - 1} = B_0 + B_1 x + \frac{B_2 x^2}{2!} + \cdots = \sum_{k \geq 0} \frac{B_k x^k}{k!}. \tag{4}$$

The coefficients of this series, which occur in a wide variety of problems, were introduced by James Bernoulli in 1713. (Some books use a different notation for Bernoulli numbers, but our notation is used in most modern references.) We have

$$B_0 = 1, \qquad B_1 = -\tfrac{1}{2}, \qquad B_2 = \tfrac{1}{6}, \qquad B_3 = 0, \qquad B_4 = -\tfrac{1}{30}. \tag{5}$$

Further values are given in Appendix B. Since

$$\frac{x}{e^x - 1} + \frac{x}{2} = \frac{x}{2}\,\frac{e^x + 1}{e^x - 1} = -\frac{x}{2}\,\frac{e^{-x} + 1}{e^{-x} - 1}$$

is an even function, we see that

$$B_3 = B_5 = B_7 = B_9 = \cdots = 0. \tag{6}$$

The following formula is an immediate consequence of the definition, Eq. (4):

$$\sum_k \binom{n}{k} B_k = B_n + \delta_{n1}. \tag{7}$$

We now define the "Bernoulli polynomial,"

$$B_m(x) = \sum_k \binom{m}{k} B_k x^{m-k}. \tag{8}$$

If $m = 1$, then $B_1(x) = B_0 x + B_1 = x - \frac{1}{2}$, corresponding to the polynomial used above in Eq. (3). If $m > 1$, we have by (7) $B_m(1) = B_m = B_m(0)$; $B_m(\{x\})$ has no discontinuities at integer points x.

The relevance of Bernoulli polynomials and Bernoulli numbers to our problem will soon be clear. We find from Eq. (8) that

$$\begin{aligned} B'_m(x) = \sum_k \binom{m}{k} (m-k) B_k x^{m-k-1} &= m \sum_k \binom{m-1}{k} B_k x^{m-1-k} \\ &= m B_{m-1}(x), \end{aligned} \tag{9}$$

and therefore when $m \geq 1$, we can integrate by parts as follows:

$$\begin{aligned} \frac{1}{m!} \int_1^n B_m(\{x\}) f^{(m)}(x)\, dx = {} & \frac{1}{(m+1)!} \left(B_{m+1}(1) f^{(m)}(n) - B_{m+1}(0) f^{(m)}(1)\right) \\ & - \frac{1}{(m+1)!} \int_1^n B_{m+1}(\{x\}) f^{(m+1)}(x)\, dx. \end{aligned}$$

From this result we can continue to improve the approximation, Eq. (3), and we obtain Euler's general formula:

$$\sum_{1\le k<n} f(k) = \int_1^n f(x)\,dx - \tfrac{1}{2}(f(n)-f(1)) + \frac{B_2}{2!}(f'(n)-f'(1)) + \cdots$$
$$+ \frac{(-1)^m B_m}{m!}(f^{(m-1)}(n) - f^{(m-1)}(1)) + R_m$$
$$= \int_1^n f(x)\,dx + \sum_{1\le k\le m} \frac{B_k}{k!}(f^{(k-1)}(n) - f^{(k-1)}(1)) + R_m, \tag{10}$$

where

$$R_m = \frac{(-1)^{m+1}}{m!}\int_1^n B_m(\{x\})f^{(m)}(x)\,dx. \tag{11}$$

The remainder R_m will be small when $B_m(\{x\})f^{(m)}(x)/m!$ is very small, and in fact, it is known that $|B_m(\{x\})| \le |B_m|$ when m is even, and that

$$\left|\frac{B_m(\{x\})}{m!}\right| < \frac{4}{(2\pi)^m}. \tag{12}$$

[See K. Knopp, *Theory and Application of Infinite Series* (Glasgow: Blackie, 1951), Chapter 14.] On the other hand, it usually turns out that the size of $f^{(m)}(x)$ gets large as m increases, so there is a "best" value of m at which R_m has its least value.

It is known that

$$R_{2k} = \theta\frac{B_{2k+2}}{(2k+2)!}(f^{(2k+1)}(n) - f^{(2k+1)}(1)), \qquad 0<\theta<1, \tag{13}$$

provided that $f^{(2k+1)}(x)$ tends monotonically toward zero as x increases from 1 to n. (So in these circumstances the remainder has the same sign as, and is less than, the first discarded term.) A simpler version of this result appears in exercise 3.

Let us now apply Euler's formula to some important examples. First, we set $f(x) = 1/x$. The derivatives are $f^{(m)}(x) = (-1)^m m!/x^{m+1}$, so we have, by Eq. (10),

$$H_{n-1} = \ln n + \sum_{1\le k\le m}\frac{B_k}{k}(-1)^{k-1}\left(\frac{1}{n^k}-1\right) + R_{mn}. \tag{14}$$

Now we find

$$\gamma = \lim_{n\to\infty}(H_{n-1} - \ln n) = \sum_{1\le k\le m}\frac{B_k}{k}(-1)^k + \lim_{n\to\infty} R_{mn}. \tag{15}$$

The fact that $\lim_{n\to\infty} R_{mn} = -\int_1^\infty B_m(\{x\})\,dx/x^{m+1}$ exists proves that the constant γ does in fact exist; now putting Eqs. (14) and (15) together, we

deduce a general approximation for the harmonic numbers:

$$H_{n-1} = \ln n + \gamma + \sum_{1\le k\le m} \frac{(-1)^{k-1}B_k}{kn^k} + \int_n^\infty \frac{B_m(\{x\})\,dx}{x^{m+1}}$$

$$= \ln n + \gamma + \sum_{1\le k\le m} \frac{(-1)^{k-1}B_k}{kn^k} + O\left(\frac{1}{n^m}\right). \tag{16}$$

Furthermore, by Eq. (13) we see that the error is less than the first term discarded. As a particular case we have (adding $1/n$ to both sides)

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \epsilon, \qquad 0 < \epsilon < \frac{B_6}{6n^6} = \frac{1}{252n^6}.$$

This is Eq. 1.2.7–3. The Bernoulli numbers B_k for large k get very large (approximately $2[k!/(2\pi)^k]$ when k is even), so Eq. (16) cannot be extended to a convergent infinite series for any fixed value of n.

The same technique may be applied to deduce Stirling's approximation. This time we set $f(x) = \ln x$, and applying Eq. (10), we obtain

$$\ln (n-1)! = n\ln n - n + 1 - \tfrac{1}{2}\ln n + \sum_{1<k\le m} \frac{B_k(-1)^k}{k(k-1)}\left(\frac{1}{n^{k-1}} - 1\right) + R_{mn}. \tag{17}$$

Proceeding as above, we find that

$$\lim_{n\to\infty} (\ln n! - n\ln n + n - \tfrac{1}{2}\ln n) = 1 + \sum_{1<k\le m} \frac{B_k(-1)^{k+1}}{k(k-1)} + \lim_{n\to\infty} R_{mn}$$

exists; let it be called σ ("Stirling's constant") temporarily. We get Stirling's result

$$\ln n! = (n+\tfrac{1}{2})\ln n - n + \sigma + \sum_{1<k\le m} \frac{B_k(-1)^k}{k(k-1)n^{k-1}} + O\left(\frac{1}{n^m}\right). \tag{18}$$

In particular, let $m = 5$; we have

$$\ln n! = (n+\tfrac{1}{2})\ln n - n + \sigma + \frac{1}{12n} - \frac{1}{360n^3} + O\left(\frac{1}{n^5}\right).$$

Taking exponentials, we have

$$n! = e^\sigma \sqrt{n}\left(\frac{n}{e}\right)^n \exp\left(\frac{1}{12n} - \frac{1}{360n^3} + O\left(\frac{1}{n^5}\right)\right).$$

Using the fact that $e^\sigma = \sqrt{2\pi}$ (see exercise 5), and expanding the exponential, we get our final result:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \frac{1}{12n} + \frac{1}{288n^2} - \frac{139}{51840n^3} - \frac{571}{2488320n^4} + O\left(\frac{1}{n^5}\right)\right). \tag{19}$$

EXERCISES

1. [*M18*] Prove Eq. (7).

2. [*HM16*] Note that Eq. (9) follows from Eq. (8) for *any* sequence B_n, not only the sequence defined by Eq. (4). Explain why the latter sequence is necessary for the validity of Eq. (10).

3. [*HM20*] If $f^{(2k)}(x)$ has a constant sign for $1 \le x \le n$, show that

$$|R_{2k}| \le \left| \frac{B_{2k}}{(2k)!} \left(f^{(2k-1)}(n) - f^{(2k-1)}(1)\right) \right|,$$

so the remainder has smaller absolute value than the *last* term computed.

▶ **4.** [*HM20*] When $f(x) = x^m$, the high-order derivatives of f are all zero, so Euler's summation formula gives an *exact* value for $\sum_{0 \le k < n} k^m$ in terms of Bernoulli numbers. Express this value in terms of Bernoulli *polynomials*. Check your answer for $m = 0$, 1, 2. (Note that the desired sum runs from 0 to n instead of from 1 to n; Euler's summation formula may be applied with 0 replacing 1 throughout.)

5. [*HM30*] Given that

$$n! = \kappa\sqrt{n}\left(\frac{n}{e}\right)^n \left(1 + O\left(\frac{1}{n}\right)\right),$$

show that $\kappa = \sqrt{2\pi}$ by using Wallis's product (exercise 1.2.5–18). [*Hint:* Consider $\binom{2n}{n}$ for large values of n.]

▶ **6.** [*HM30*] Show that Stirling's approximation holds for noninteger n as well, i.e., that

$$\Gamma(x+1) = \sqrt{2\pi x}\left(\frac{x}{e}\right)^x \left(1 + O\left(\frac{1}{x}\right)\right), \qquad x \ge a > 0.$$

[*Hint:* Let $f(x) = \ln(x + c)$ in Euler's summation formula, and apply the definition of $\Gamma(x)$ given in Section 1.2.5.]

▶ **7.** [*HM32*] What is the approximate value of $1^1 \cdot 2^2 \cdot 3^3 \cdot \cdots \cdot n^n$?

1.2.11.3. Some applications. In this subsection we shall investigate the following three intriguing sums, and we shall deduce their approximate values:

$$P(n) = 1 + \frac{n-1}{n} + \frac{n-2}{n}\,\frac{n-2}{n-1} + \cdots = \sum_{0\le k\le n} \frac{(n-k)^k (n-k)!}{n!}, \tag{1}$$

$$Q(n) = 1 + \frac{n-1}{n} + \frac{n-1}{n}\,\frac{n-2}{n} + \cdots = \sum_{1\le k\le n} \frac{n!}{(n-k)!\,n^k}, \tag{2}$$

$$R(n) = 1 + \frac{n}{n+1} + \frac{n}{n+1}\,\frac{n}{n+2} + \cdots = \sum_{0\le k} \frac{n!\,n^k}{(n+k)!}. \tag{3}$$

These functions, which are similar in appearance yet intrinsically different, arise in several algorithms that we shall encounter later. Both $P(n)$ and $Q(n)$ are

finite sums, while $R(n)$ is an infinite sum. It seems that when n is large, all three of these sums will be nearly equal, although it is not obvious what the approximate value of *any* of these three functions will be. Our quest for approximate values of these functions will lead us through a number of very instructive side results. (The reader may wish to stop reading temporarily and try his hand at studying these functions before going on to see how they are attacked here.)

First, we observe an important connection between $Q(n)$ and $R(n)$:

$$Q(n) + R(n) = \frac{n!}{n^n}\left(\left(1 + n + \cdots + \frac{n^{n-1}}{(n-1)!}\right) + \left(\frac{n^n}{n!} + \frac{n^{n+1}}{(n+1)!} + \cdots\right)\right)$$
$$= \frac{n!\,e^n}{n^n}. \tag{4}$$

To get any further we must therefore consider the partial sums of the series for e^n. By using Taylor's formula with remainder,

$$f(x) = f(0) + f'(0)x + \cdots + \frac{f^{(n)}(0)x^n}{n!} + \int_0^x \frac{t^{n+1}}{(n+1)!} f^{(n+1)}(x-t)\,dt, \tag{5}$$

we are soon led to an important function which is known as the *incomplete gamma function:*

$$\gamma(a, x) = \int_0^x e^{-t} t^{a-1}\,dt. \tag{6}$$

We shall assume that $a > 0$. By exercise 1.2.5–20, we have $\gamma(a, \infty) = \Gamma(a)$; this accounts for the name "incomplete gamma function." It has two series expansions in powers of x (see exercises 2 and 3):

$$\gamma(a, x) = \frac{x^a}{a} - \frac{x^{a+1}}{a+1} + \frac{x^{a+2}}{2!(a+2)} - \cdots = \sum_{k\ge 0} \frac{(-1)^k x^{k+a}}{k!(k+a)}, \tag{7}$$

$$e^x\gamma(a, x) = \frac{x^a}{a} + \frac{x^{a+1}}{a(a+1)} + \frac{x^{a+2}}{a(a+1)(a+2)} + \cdots$$
$$= \sum_{k\ge 0} \frac{x^{k+a}}{a(a+1)\ldots(a+k)}. \tag{8}$$

From the second formula we see the connection with $R(n)$:

$$R(n) = \frac{n!\,e^n}{n^n}\left(\frac{\gamma(n, n)}{(n-1)!}\right). \tag{9}$$

This equation has purposely been written in a more complicated form than necessary, since $\gamma(n, n)$ is a fraction of $\gamma(n, \infty) = \Gamma(n) = (n-1)!$. Thus $R(n)$ lies somewhere between zero and $n!\,e^n/n^n$; by Stirling's formula, $n!\,e^n/n^n$ is approximately $\sqrt{2\pi n}$.

The problem boils down to getting good estimates of $\gamma(n, n)/(n-1)!$. *We shall now determine the approximate value of* $\gamma(x+1, x+y)/\Gamma(x+1)$, when y is fixed and x is large. The methods to be used here are more important than the results, so the reader should study the following derivation carefully.

By definition, we have

$$\begin{aligned}\frac{\gamma(x+1, x+y)}{\Gamma(x+1)} &= \frac{1}{\Gamma(x+1)}\int_0^{x+y} e^{-t}t^x\,dt \\ &= 1 - \frac{1}{\Gamma(x+1)}\int_x^{\infty} e^{-t}t^x\,dt + \frac{1}{\Gamma(x+1)}\int_x^{x+y} e^{-t}t^x\,dt \\ &= 1 - \qquad I_1 \qquad + \qquad I_2. \end{aligned} \tag{10}$$

Now we consider each integral separately.

Estimate of I_1: In the integral I_1, we convert to an integral from 0 to infinity by substituting $t = x(1+u)$:

$$\begin{aligned} I_1 &= \frac{e^{-x}x^x}{\Gamma(x+1)}\int_0^{\infty} xe^{-xu}\,(1+u)^x\,du \\ &= \frac{e^{-x}x^x}{\Gamma(x+1)}\int_0^{\infty} xe^{-xv}\left(1+\frac{1}{u}\right)dv, \end{aligned} \tag{11}$$

$$\text{if} \qquad v = u - \ln(1+u); \quad dv = \left(1 - \frac{1}{1+u}\right)du.$$

This change of variable from u to v is justified, since v is a monotone function of u.

In the last integral we will replace $1 + 1/u$ by a power series in v. We have

$$v = \tfrac{1}{2}u^2 - \tfrac{1}{3}u^3 + \tfrac{1}{4}u^4 - \tfrac{1}{5}u^5 + \cdots = (u^2/2)(1 - \tfrac{2}{3}u + \tfrac{1}{2}u^2 - \tfrac{2}{5}u^3 + \cdots).$$

If $w = \sqrt{2v}$, we have

$$\begin{aligned} w &= u(1 - \tfrac{2}{3}u + \tfrac{1}{2}u^2 - \tfrac{2}{5}u^3 + \cdots)^{1/2} \\ &= u - \tfrac{1}{3}u^2 + \tfrac{7}{36}u^3 - \tfrac{73}{540}u^4 + \tfrac{1331}{12960}u^5 + O(u^6). \end{aligned}$$

(This expansion may be obtained by the binomial theorem; efficient methods for doing this transformation, as well as the other power series manipulations done below, are considered in Section 4.7.) We can now solve for u as a power series in w:

$$u = w + \frac{1}{3}w^2 + \frac{1}{36}w^3 - \frac{1}{270}w^4 + \frac{1}{4320}w^5 + O(w^6);$$

$$\begin{aligned} 1 + \frac{1}{u} &= 1 + \frac{1}{w} - \frac{1}{3} + \frac{1}{12}w - \frac{2}{135}w^2 + \frac{1}{864}w^3 + O(w^4) \\ &= \frac{1}{\sqrt{2}}v^{-1/2} + \frac{2}{3} + \frac{\sqrt{2}}{12}v^{1/2} - \frac{4}{135}v + \frac{\sqrt{2}}{432}v^{3/2} + O(v^2). \end{aligned} \tag{12}$$

In all of these formulas, the O-notation refers to small values of the argument, that is, $|u| \le r$, $|v| \le r$, $|w| \le r$ for sufficiently small positive r. Is this good enough? The substitution of $1 + 1/u$ in terms of v in Eq. (11) is supposed to be valid for $0 \le v < \infty$, not only for $|v| \le r$. Fortunately, it turns out that the value of the integral from 0 to ∞ depends almost entirely on the values of the integrand near zero. In fact, we have (see exercise 4)

$$\int_r^\infty xe^{-xv}\left(1+\frac{1}{u}\right)dv = O(e^{-rx}) \tag{13}$$

for any fixed $r > 0$, and for large x. We are interested in an approximation up to terms $O(x^{-m})$, and since $O((1/e^r)^x)$ is much smaller than $O(x^{-m})$ for any positive r, m, we need integrate only from 0 to r, for any fixed positive r. We therefore take r to be small enough so that all the power series manipulations done above are justified (cf. Eqs. 1.2.11.1–10 and 12).

Now

$$\int_0^\infty xe^{-xv}v^\alpha\,dv = \frac{1}{x^\alpha}\int_0^\infty e^{-q}q^\alpha\,dq = \frac{1}{x^\alpha}\Gamma(\alpha+1), \qquad \text{if} \qquad \alpha > -1; \tag{14}$$

so by putting the series, Eq. (12), into the integral, Eq. (11), we have finally

$$I_1 = \frac{e^{-x}x^x}{\Gamma(x+1)}\left(\sqrt{\frac{\pi}{2}}\,x^{1/2} + \frac{2}{3} + \frac{\sqrt{2\pi}}{24}\,x^{-1/2} - \frac{4}{135}\,x^{-1} + \frac{\sqrt{2\pi}}{576}\,x^{-3/2} + O(x^{-2})\right). \tag{15}$$

Estimate of I_2: In the integral I_2, we substitute $t = u + x$ and obtain

$$I_2 = \frac{e^{-x}x^x}{\Gamma(x+1)}\int_0^y e^{-u}\left(1+\frac{u}{x}\right)^x du. \tag{16}$$

Now

$$e^{-u}\left(1+\frac{u}{x}\right)^x = \exp\left(-u + x\ln\left(1+\frac{u}{x}\right)\right) = \exp\left(\frac{-u^2}{2x} + \frac{u^3}{3x^2} + O(x^{-3})\right)$$

$$= 1 - \frac{u^2}{2x} + \frac{u^4}{8x^2} + \frac{u^3}{3x^2} + O(x^{-3})$$

for $0 \le u \le y$ and large x. Therefore we find that

$$I_2 = \frac{e^{-x}x^x}{\Gamma(x+1)}\left(y - \frac{y^3}{6}\,x^{-1} + \left(\frac{y^4}{12} + \frac{y^5}{40}\right)x^{-2} + O(x^{-3})\right). \tag{17}$$

Finally, we analyze the coefficient $e^{-x}x^x/\Gamma(x+1)$ which appears in both Eqs. (15) and (17). By Stirling's approximation, which is valid for the gamma

function by exercise 1.2.11.2–6, we have

$$\frac{e^{-x}x^x}{\Gamma(x+1)} = \frac{e^{-1/12x+O(x^{-3})}}{\sqrt{2\pi x}}$$

$$= \frac{1}{\sqrt{2\pi}}x^{-1/2} - \frac{1}{12\sqrt{2\pi}}x^{-3/2} + \frac{1}{288\sqrt{2\pi}}x^{-5/2} + O(x^{-7/2}). \quad (18)$$

Now the grand summing up—combining Eqs. (10), (15), (17), and (18), we have

Theorem A. *For large values of x, and fixed y,*

$$\frac{\gamma(x+1, x+y)}{\Gamma(x+1)} = \frac{1}{2} + \left(\frac{y-2/3}{\sqrt{2\pi}}\right)x^{-1/2} + \frac{1}{\sqrt{2\pi}}\left(\frac{23}{270} - \frac{y}{12} - \frac{y^3}{6}\right)x^{-3/2}$$

$$+ O(x^{-5/2}). \quad \blacksquare \quad (19)$$

The method we have used shows how this approximation could be extended to further powers of x as far as we please.

This theorem can be used to obtain the approximate values of $R(n)$ and $Q(n)$, by using Eqs. (4) and (9), but we shall defer that calculation until later. Let us now turn to $P(n)$, for which somewhat different methods seem to be required.

$$P(n) = \sum_{0\le k\le n} \frac{k^{n-k}k!}{n!} = \frac{\sqrt{2\pi}}{n!} \sum_{0\le k\le n} k^{n+1/2}e^{-k}\left(1 + \frac{1}{12k} + O(k^{-2})\right). \quad (20)$$

Thus to get the values of $P(n)$, we must study sums of the form

$$\sum_{0\le k\le n} k^{n+1/2}e^{-k}.$$

Let $f(x) = x^{n+1/2}e^{-x}$ and apply Euler's summation formula:

$$\sum_{0\le k\le n} k^{n+1/2}e^{-k} = \int_0^n x^{n+1/2}e^{-x}\,dx + \tfrac{1}{2}n^{n+1/2}e^{-n} + \tfrac{1}{24}n^{n-1/2}e^{-n} - R. \quad (21)$$

Analysis of the remainder (cf. exercise 5) shows that $R = O(n^{n-1/2}e^{-n})$; and since the integral is an incomplete gamma function, we have

$$\sum_{0\le k\le n} k^{n+1/2}e^{-k} = \gamma(n+\tfrac{3}{2}, n) + \tfrac{1}{2}n^{n+1/2}e^{-n} + O(n^{n-1/2}e^{-n}). \quad (22)$$

Our formula, Eq. (20), also requires an estimate of the sum

$$\sum_{0\le k\le n} k^{n-1/2}e^{-k} = \sum_{0\le k\le n-1} k^{(n-1)+1/2}e^{-k} + n^{n-1/2}e^{-n},$$

and this can also be obtained by Eq. (22).

We now have enough formulas at our disposal to determine the approximate values of $P(n)$, $Q(n)$, and $R(n)$, and it is only a matter of substituting and multiplying, etc. In this process we shall have occasion to use the expansion

$$(n+\alpha)^{n+\beta} = n^{n+\beta}e^{\alpha}\left(1+\alpha\left(\beta-\frac{\alpha}{2}\right)\frac{1}{n}+O(n^{-2})\right), \tag{23}$$

which is proved in exercise 6. The method of (21) yields only the first three terms in the asymptotic series for $P(n)$; further terms can be obtained by using the instructive technique described in exercise 14.

The result of all these calculations gives us the desired asymptotic formulas:

$$P(n) = \sqrt{\frac{\pi n}{2}} - \frac{2}{3} + \frac{11}{24}\sqrt{\frac{\pi}{2n}} + \frac{4}{135n} - \frac{71}{1152}\sqrt{\frac{\pi}{2n^3}} + O(n^{-2}), \tag{24}$$

$$Q(n) = \sqrt{\frac{\pi n}{2}} - \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2n}} - \frac{4}{135n} + \frac{1}{288}\sqrt{\frac{\pi}{2n^3}} + O(n^{-2}), \tag{25}$$

$$R(n) = \sqrt{\frac{\pi n}{2}} + \frac{1}{3} + \frac{1}{12}\sqrt{\frac{\pi}{2n}} + \frac{4}{135n} + \frac{1}{288}\sqrt{\frac{\pi}{2n^3}} + O(n^{-2}). \tag{26}$$

The functions studied here have received only light treatment in the published literature. The first term $\sqrt{\pi n/2}$ in the expansion of $P(n)$ was given by R. Church [see K. Iverson, *A Programming Language* (New York: Wiley, 1962), pp. 239–240]. Using Church's result, a table of $P(n)$ for $n \le 2000$, and a good slide rule, the author deduced an empirical estimate $P(n) \approx \sqrt{\pi n/2} - 0.6667 + 0.575/\sqrt{n}$. It was natural to conjecture that 0.6667 was really an approximation to $\frac{2}{3}$, and that 0.575 would perhaps turn out to be an approximation to $\gamma = 0.57721\ldots$ (why not be optimistic?). Later, as this section was being written, the above expansion of $P(n)$ was developed, and the conjecture $\frac{2}{3}$ was verified; for the 0.575 we have not γ but $\frac{11}{24}\sqrt{\pi/2} \approx 0.574$. This nicely confirms both the theory and the empirical estimates.

Formulas equivalent to the asymptotic values of $Q(n)$ and $R(n)$ were first determined by the brilliant self-taught Indian mathematician S. Ramanujan, who posed the problem of estimating $n!e^n/2n^n - Q(n)$ in *J. Indian Math. Soc.* **3** (1911), 128; **4** (1912), 151–152. In his answer to the problem, he gave the asymptotic series $\frac{1}{3} + \frac{4}{135}n^{-1} - \frac{8}{2835}n^{-2} - \frac{1}{8505}n^{-3} + \ldots$, which goes considerably beyond Eq. (25). His derivation was somewhat more elegant than the method described above; to estimate I_1, he substituted $t = x + u\sqrt{2x}$, and expressed the integrand as a sum of terms of the form $c_{jk}\int_0^\infty \exp(-u^2)u^j n^{-k/2}\,du$. The integral I_2 can be avoided completely, since $a\gamma(a,x) = x^a e^{-x} + \gamma(a+1,x)$ when $a > 0$ (see (8)). The derivation we have used, which is instructive in spite of its unnecessary complications, is due to R. Furch [*Zeitschrift für Physik* **112** (1939), 92–95], who was primarily interested in the value of y which makes $\gamma(x+1, x+y) = \frac{1}{2}\Gamma(x+1)$. For a bibliography of other investigations of $Q(n)$, see H. W. Gould, *AMM* **75** (1968), 1019–1021. The asymptotic properties

of the incomplete gamma function were later extended to complex arguments by F. G. Tricomi ["Asymptotische Eigenschaften der unvollständigen Gammafunktion," *Math. Zeitschrift* **53** (1950), 136–148].

Further study of the functions $P(n)$, $Q(n)$, and $R(n)$ would be interesting. The derivations given above use only simple techniques of elementary calculus; note that we have used different methods for each function!

For detailed information about techniques for solving similar problems, the reader is referred to the book *Asymptotic Methods in Analysis* by N. G. de Bruijn (Amsterdam: North Holland Publ., 1961).

EXERCISES

1. [*HM20*] Prove Eq. (5) by induction on n.
2. [*HM20*] Obtain Eq. (7) from Eq. (6).
3. [*M20*] Derive Eq. (8) from Eq. (7).
4. ▶ [*HM10*] Prove Eq. (13).
5. [*HM24*] Show that R in Eq. (21) is $O(n^{n-1/2}e^{-n})$.
6. ▶ [*HM20*] Prove Eq. (23).
7. ▶ [*HM30*] In the evaluation of I_2, we had to consider

$$\int_0^y e^{-u}\left(1+\frac{u}{x}\right)^x du.$$

Give an asymptotic representation of

$$\int_0^{yx^{1/4}} e^{-u}\left(1+\frac{u}{x}\right)^x du$$

to terms of $O(x^{-2})$, when y is fixed and x is large.

8. [*HM30*] Assume that $0 \le r \le \frac{1}{2}$. Suppose $f(x) = O(x^r)$; show that

$$\int_0^{f(x)} e^{-u}\left(1+\frac{u}{x}\right)^x du = \int_0^{f(x)} \exp\left(\frac{-u^2}{2x}+\frac{u^3}{3x^2}-\cdots+\frac{(-1)^{m-1}u^m}{mx^{m-1}}\right) du + O(x^{-s})$$

if $m = \lceil (s+2r)/(1-r) \rceil$. [This proves in particular a result due to Tricomi: if $f(x) = O(\sqrt{x})$, then

$$\int_0^{f(x)} e^{-u}\left(1+\frac{u}{x}\right)^x du = \sqrt{2x}\int_0^{f(x)/\sqrt{2x}} e^{-t^2} dt + O(1).]$$

9. ▶ [*HM36*] What is the behavior of $\gamma(x+1, px)/\Gamma(x+1)$ for large x? (Here p is a real constant; and if $p < 0$, we assume x is an integer, so that t^x is defined for negative t.) Obtain at least two terms of the asymptotic expansion, before resorting to O-terms.

10. [*HM34*] Under the assumptions of the preceding problem, with $p \neq 1$, obtain the asymptotic expansion of

$$\gamma\left(x+1, px+\frac{p}{p-1}y\right) - \gamma(x+1, px),$$

for fixed y, to terms of the same order as obtained in the previous exercise.

▶ **11.** [*HM35*] Let us generalize the functions $Q(n)$, $R(n)$ by introducing a parameter x so that

$$Q_x(n) = 1+\left(\frac{n-1}{n}\right)x+\left(\frac{n-1}{n}\right)\left(\frac{n-2}{n}\right)x^2+\cdots,$$

and

$$R_x(n) = 1+\left(\frac{n}{n+1}\right)x+\left(\frac{n}{n+1}\right)\left(\frac{n}{n+2}\right)x^2+\cdots.$$

Explore this situation and find asymptotic formulas when $x \neq 1$.

12. [*HM20*] The function $\int_0^x e^{-t^2/2}\,dt$ which appeared in connection with the normal distribution (see Section 1.2.10) can be expressed as a special case of the incomplete gamma function. Find values of a, b, y such that $b\gamma(a, y)$ equals the above function.

13. [*HM46*] (S. Ramanujan.) Prove that $R(n) - Q(n) = \frac{2}{3} + 8/(135(n+\theta(n))$, where $\frac{2}{21} \leq \theta(n) \leq \frac{8}{45}$. (This implies the much weaker result $R(n+1) - Q(n+1) < R(n) - Q(n)$.)

▶ **14.** [*HM39*] (N. G. de Bruijn.) The purpose of this exercise is to find the asymptotic expansion of $\sum_{0\leq k\leq n} k^{n+\alpha}e^{-k}$ for fixed α, as $n \to \infty$. (a) Replacing k by $n-k$, show that the given sum equals $n^{n+\alpha}e^{-n}\sum_{0\leq k\leq n} e^{-k^2/2n}f(k, n)$, where $f(k, n) = (1-k/n)^\alpha \exp(-k^3/3n^2 - k^4/4n^3 - \cdots)$. (b) Show that for all $m \geq 0$ and $\epsilon > 0$, $f(k, n)$ can be written in the form $\sum_{0\leq i\leq j\leq m} c_{ij}k^{2i+j}n^{-i-j} + O(n^{(m+1)(-1/2+3\epsilon)})$, when $0 \leq k \leq n^{1/2+\epsilon}$. (c) Prove that as a consequence of (b), $\sum_{0\leq k\leq n} e^{-k^2/2n}f(k, n) = \sum_{0\leq i\leq j\leq m} c_{ij}n^{-i-j}\sum_{k\geq 0} k^{2i+j}e^{-k^2/2n} + O(n^{-m/2+\delta})$, for all $\delta > 0$. [Hint: Over the range $n^{1/2+\epsilon} < k < \infty$, the sums are $O(n^{-r})$ for all r.] (d) Show that the asymptotic expansion of $\sum_{k\geq 0} k^t e^{-k^2/2n}$ for fixed $t \geq 0$ can be obtained by Euler's summation formula. (e) Finally therefore

$$\sum_{0\leq k\leq n} k^{n+\alpha}e^{-k} = n^{n+\alpha}e^{-n}\left(\sqrt{\frac{\pi n}{2}} - \frac{1}{6} - \alpha + \left(\frac{1}{12}+\frac{1}{2}\alpha+\frac{1}{2}\alpha^2\right)\sqrt{\frac{\pi}{2n}} + O(n^{-1})\right);$$

this computation can in principle be extended to $O(n^{-r})$ for any desired r.

15. [*HM20*] Show that

$$\int_0^\infty \left(1+\frac{z}{n}\right)^n e^{-z}\,dz$$

is related to $Q(n)$.

16. [*M24*] Prove the identity

$$\sum_k (-1)^k \binom{n}{k} k^{n-1}Q(k) = (-1)^n(n-1)!, \qquad \text{when} \qquad n > 0.$$

1.3. MIX

In many places throughout this book we will have occasion to refer to a computer's "machine language." The machine we use is a mythical computer called "MIX." MIX is very much like nearly every computer now in existence (except that it is, perhaps, nicer). The language of MIX has been designed to be powerful enough to allow brief programs to be written for most algorithms, yet simple enough so that its operations are easily learned.

The reader is urged to study this section carefully, since MIX language appears in so many parts of this book. There should be no hesitation about learning a new machine language; indeed, the author has found it not uncommon to be writing programs in a half dozen different machine languages during the same week! Everyone with more than a casual interest in computers will probably get to know several different machine languages in the course of his lifetime. MIX has been specially designed to be so much like most existing machine languages that its characteristics are easy to assimilate.

1.3.1. Description of MIX.

MIX is the world's first polyunsaturated computer. Like most machines, it has an identifying number—the 1009. This number was found by taking 16 actual computers which are very similar to MIX and on which MIX can be easily simulated, then averaging their numbers with equal weight:

$$\lfloor (360 + 650 + 709 + 7070 + \text{U3} + \text{SS80} + 1107 + 1604 + \text{G20} + \text{B220} + \text{S2000} + 920 + 601 + \text{H800} + \text{PDP4} + \text{II})/16 \rfloor = 1009. \quad (1)$$

This number may also be obtained in a more simple way by taking Roman numerals.

MIX has a peculiar property in that it is both binary and decimal at the same time. *The programmer doesn't actually know whether he is programming a machine with base 2 or base 10 arithmetic.* This has been done so that algorithms written in MIX can be used on either type of machine with little change, and so that MIX can be easily simulated on either type of machine. Those programmers accustomed to a binary machine can think of MIX as binary; those accustomed to decimal may regard MIX as decimal. Programmers from another planet might choose to think of MIX as a ternary machine.

Words. The basic unit of information is a *byte*. Each byte contains an *unspecified* amount of information, but it must be capable of holding at least 64 distinct values. That is, we know that any number between 0 and 63, inclusive, can be contained in one byte. Furthermore, each byte contains *at most* 100 distinct values. On a binary computer a byte must therefore be composed of six bits; on a decimal computer we have two digits per byte.

Programs expressed in the MIX language should be written so that no more than sixty-four values are ever assumed for a byte. If we wish to treat the

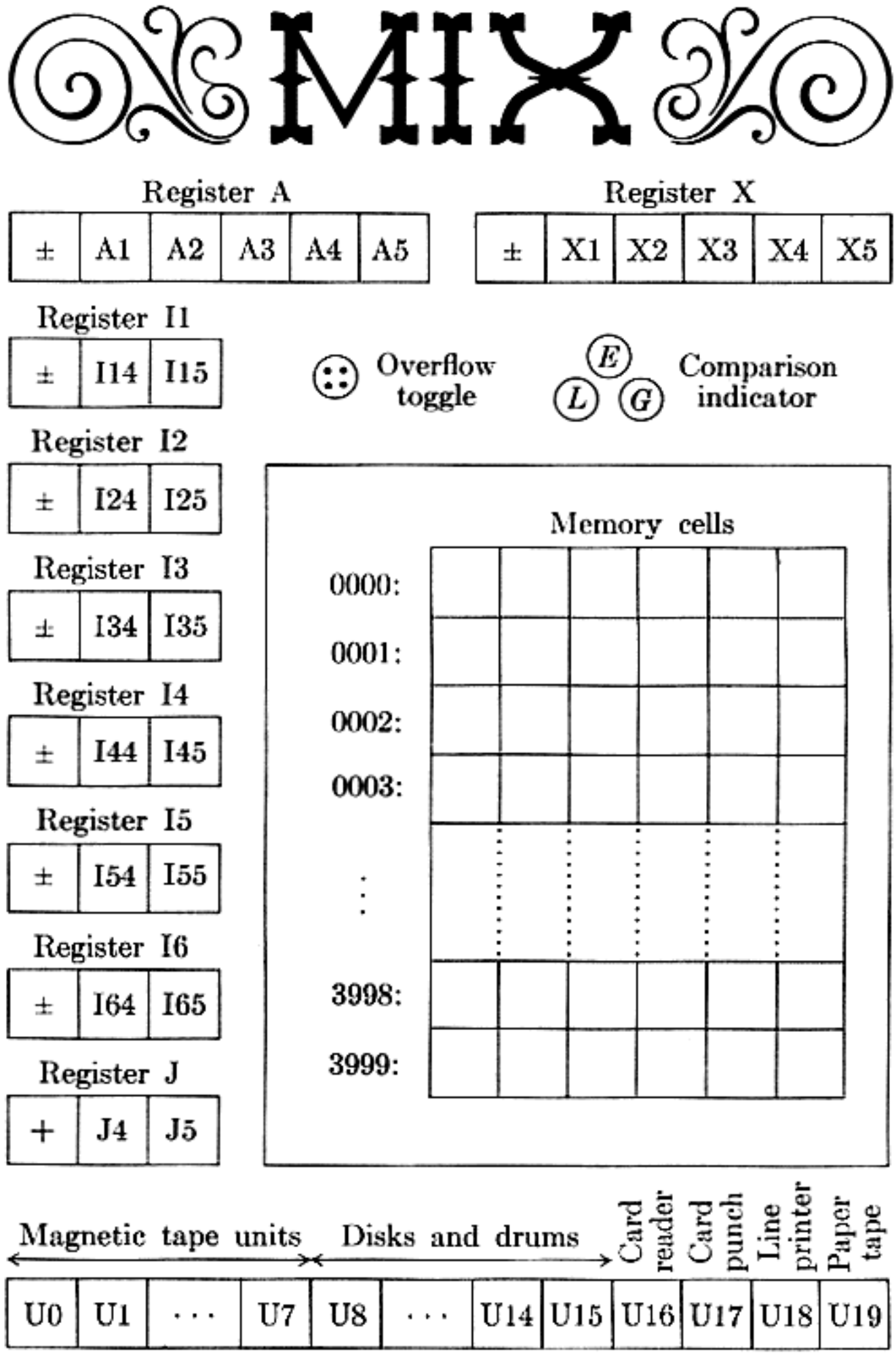


Fig. 13. The MIX computer.

number 80, we should always leave two adjacent bytes for expressing it, even though on a decimal computer one byte is sufficient. *An algorithm in* MIX *should work properly regardless of how big a byte is.* Although it is quite possible to write programs which depend on the byte size, this is an illegal act which will not be tolerated; the only legitimate programs are those which would give correct results with all byte sizes. It is not hard to abide by this ground rule, and we will thereby find that programming a decimal computer isn't so different from programming a binary one after all.

Two adjacent bytes can express the numbers 0 through 4095.
Three adjacent bytes can express the numbers 0 through 262143.
Four adjacent bytes can express the numbers 0 through 16777215.
Five adjacent bytes can express the numbers 0 through 1073741823.

A computer word is five bytes plus a sign. The sign position has only two possible values, + and −.

Registers. There are nine registers in MIX (see Fig. 13):

The A-register (Accumulator) is five bytes plus sign.
The X-register (Extension) is also five bytes plus sign.
The I-registers (Index registers) I1, I2, I3, I4, I5, and I6 each hold two bytes plus sign.
The J-register (Jump address) holds two bytes, and its sign is always +.

We shall use a small letter "r" prefixed to the name, to identify a MIX register. Thus, "rA" means "register A."

The A-register has many uses, especially for arithmetic and operating on data. The X-register is an extension on the "right-hand side" of rA, and it is used in connection with rA to hold ten bytes of a product or dividend, or it can be used to hold information shifted to the right out of rA. The index registers rI1, rI2, rI3, rI4, rI5, and rI6 are used primarily for counting and for referencing variable memory addresses. The J-register always holds the address of the instruction following the preceding "JUMP" instruction, and it is primarily used in connection with subroutines.

Besides these registers, MIX contains

an *overflow toggle* (a single bit which is either "on" or "off"),
a *comparison indicator* (which has three values: less, equal, or greater),
memory (4000 words of storage, each word with five bytes plus sign),
and *input-output devices* (card, tape, etc.).

Partial fields of words. The five bytes and sign of a computer word are numbered as follows:

0	1	2	3	4	5
±	Byte	Byte	Byte	Byte	Byte

(2)

Most of the instructions allow the programmer to use only part of a word if he chooses. In this case a "field specification" is given. The allowable fields are those which are adjacent in a computer word, and they are represented by (L:R), where L is the number of the left-hand part and R is the number of the right-hand part of the field. Examples of field specifications are:

(0:0), the sign only.
(0:2), the sign and the first two bytes.
(0:5), the whole word. This is the most common field specification.
(1:5), the whole word except for the sign.
(4:4), the fourth byte only.
(4:5), the two least significant bytes.

The use of these field specifications varies slightly from instruction to instruction, and it will be explained in detail for each instruction where it applies.

Although it is generally not important to the programmer, the field (L:R) is denoted in the machine by the single number 8L + R, and this number will fit in one byte.

Instruction format. Computer words used for instructions have the following form:

0	1	2	3	4	5
±	A	A	I	F	C

. (3)

The rightmost byte, C, is the operation code telling what operation is to be performed. For example, C = 8 is the operation `LDA`, "load the A register."

The F-byte holds a modification of the operation code. F is usually a field specification (L:R) = 8L + R; for example, if C = 8 and F = 11, the operation is "load the A-register with the (1:3) field." Sometimes F is used for other purposes; on input-output instructions, for example, F is the number of the affected input or output unit.

The left-hand portion of the instruction, ±AA, is the "address." (Note that the sign is part of the address.) The I-field, which comes next to the address, is the "index specification," which may be used to modify the address of an instruction. If I = 0, the address ±AA is used without change; otherwise I should contain a number i between 1 and 6, and the contents of index register Ii are added algebraically to ±AA; the result is used as the address of the instruction. This indexing process takes place on *every* instruction. We will use the letter M to indicate the address after any specified indexing has occurred. (If the addition of the index register to the address ±AA yields a result which does not fit in two bytes, the value of M is undefined.)

In most instructions, M will refer to a memory cell. The terms "memory cell" and "memory location" are used almost interchangeably in this book. We assume that there are 4000 memory cells, numbered from 0 to 3999; hence every memory location can be addressed with two bytes. For every instruction in which M is to refer to a memory cell we must have $0 \le M \le 3999$, and in this case we will write `CONTENTS`(M) to denote the value stored in memory location M.

On certain instructions, the "address" M has another significance, and it may even be negative. Thus one instruction adds M to an index register, and this takes account of the sign of M.

Notation. To discuss instructions in a readable manner, we will use the notation

```
OP  ADDRESS,I(F)                (4)
```

to denote an instruction like (3). Here `OP` is a symbolic name which is given to the operation code (the C-part) of the instruction; `ADDRESS` is the ±AA portion; and `I`, `F` represent the I- and F-fields, respectively.

If I is zero, the ",I" is omitted. If F is the *normal* F-specification for this particular operator, the "(F)" need not be written. The normal F-specification for almost all operators is (0:5), representing a whole word. If a different F is standard, it will be mentioned explicitly when we discuss a particular operator.

For example, the instruction to load a number into the accumulator is called LDA and it is operation code number 8. We have

Conventional representation	Actual numeric instruction			
LDA 2000,2(0:3)	+ 2000	2	3	8
LDA 2000,2(1:3)	+ 2000	2	11	8
LDA 2000(1:3)	+ 2000	0	11	8
LDA 2000	+ 2000	0	5	8
LDA -2000,4	- 2000	4	5	8

(5)

To render these in words, the instruction "LDA 2000,2(0:3)" may be read "Load A with the contents of location 2000 indexed by 2, the zero-three field."

To represent the numerical contents of a MIX word, we will always use a box notation like that above. Note that in the word

+	2000	2	3	8

the number +2000 is shown filling two adjacent bytes and sign; the actual contents of byte (1:1) and of byte (2:2) will vary from one MIX computer to another, since byte size is variable. As a further example of this notation for MIX words, the diagram

-	10000	3000

represents a word with two fields, a three-byte-plus-sign field containing -10000 and a two-byte field containing 3000. When a word is split into more than one field, it is said to be "packed."

Rules for each instruction. The remarks following (3) above have defined the quantities M, F, and C for every word used as an instruction. We will now define the actions corresponding to each instruction.

Loading operators

• LDA (load A). C = 8; F = field.

The specified field of CONTENTS(M) replaces the previous contents of register A.

On all operations where a partial field is used as an input, the sign is used if it is a part of the field, otherwise the sign + is understood. The field is shifted over to the right-hand part of the register as it is loaded.

Examples: If F is the normal field specification (0:5), the entire contents of location M is loaded. If F is (1:5), the absolute value of CONTENTS(M) is loaded with a plus sign. If M contains an *instruction* word and if F is (0:2), the "±AA" field is loaded as

±	0	0	0	A	A

.

Suppose location 2000 contains the word

−	80	3	5	4

; (6)

then we get the following results from loading various partial fields:

Instruction	Contents of rA afterwards					
LDA 2000	−	80		3	5	4
LDA 2000(1:5)	+	80		3	5	4
LDA 2000(3:5)	+	0	0	3	5	4
LDA 2000(0:3)	−	0	0	80		3
LDA 2000(4:4)	+	0	0	0	0	5
LDA 2000(0:0)	−	0	0	0	0	0
LDA 2000(1:1)	+	0	0	0	0	?

(The last example has a partially unknown effect since byte size is variable.)

- LDX (load X). C = 15; F = field.

This is the same as LDA, except that rX is loaded instead of rA.

- LD*i* (load *i*). C = 8 + *i*; F = field.

This is the same as LDA, except that r*I*i is loaded instead of rA. An index register contains only two bytes (not five) plus sign; bytes 1, 2, 3 are always assumed to be zero. The LD*i* instruction is considered undefined if it would result in setting bytes 1, 2, 3 to anything but zero.

In the description of all instructions, "*i*" stands for an integer, $1 \le i \le 6$. Thus, LD*i* stands for six different instructions: LD1, LD2, . . . , LD6.

- LDAN (load A negative). C = 16; F = field.
- LDXN (load X negative). C = 23; F = field.
- LD*i*N (load *i* negative). C = 16 + *i*; F = field.

These eight instructions are the same as LDA, LDX, LD*i*, respectively, except that the *opposite* sign is loaded.

Storing operators.

- STA (store A). C = 24; F = field.

The contents of rA replaces the field of CONTENTS(M) specified by F. The other parts of CONTENTS(M) are unchanged.

On a *store* operation the field F has the opposite significance from the *load* operation. The number of bytes in the field is taken from the right-hand side of the register and shifted *left* if necessary to be inserted in the proper field of CONTENTS(M). The sign is not altered unless it is part of the field. The contents of the register is not affected.

Examples: Suppose that location 2000 contains

−	1	2	3	4	5

and register A contains

+	6	7	8	9	0

.

Then:

Instruction	Contents of location 2000 afterwards
STA 2000	+ 6 7 8 9 0
STA 2000(1:5)	− 6 7 8 9 0
STA 2000(5:5)	− 1 2 3 4 0
STA 2000(2:2)	− 1 0 3 4 5
STA 2000(2:3)	− 1 9 0 4 5
STA 2000(0:1)	+ 0 2 3 4 5

• STX (store X). C = 31; F = field.
Same as STA except rX is stored rather than rA.

• ST*i* (store *i*). C = 24 + *i*; F = field.
Same as STA except rI*i* is stored rather than rA. Bytes 1, 2, 3 of an index register are zero; thus if rI1 contains

±	*m*	*n*

,

this behaves as though it were

±	0	0	0	*m*	*n*

.

• STJ (store J). C = 32; F = field.
Same as ST*i* except rJ is stored, and its sign is always +.

On STJ *the normal field specification for F is* (0:2), *not* (0:5). This is natural, since STJ is almost always done into the address field of an instruction.

• STZ (store zero). C = 33; F = field.
Same as STA except plus zero is stored. In other words, the specified field of CONTENTS(M) is cleared to zero.

Arithmetic operators. On the add, subtract, multiply, and divide operations, a field specification is allowed. A field specification of "(0:6)" can be used to indicate a "floating-point" operation (see Section 4.2), but few of the programs we will write for MIX will use this feature; floating-point instructions will be used primarily in the programs written by the compilers discussed in Chapter 12.

The standard field specification is, as usual, (0:5). Other fields are treated as in LDA. We will use the letter V to indicate the specified field of CONTENTS(M); thus, V is the value which would have been loaded into register A if the operation code were LDA.

• ADD. C = 1; F = field.
V is added to rA. If the magnitude of the result is too large for register A, the overflow toggle is set on, and the remainder of the addition appearing in rA is as though a "1" had been carried into another register to the left of A. (Otherwise the setting of the overflow toggle is unchanged.) If the result is zero, the sign of rA is unchanged.

Example: The sequence of instructions below gives the sum of the five bytes of register A.

```
STA   2000
LDA   2000(5:5)
ADD   2000(4:4)
ADD   2000(3:3)
ADD   2000(2:2)
ADD   2000(1:1)
```

This is sometimes called "sideways addition."

• SUB (subtract). C = 2; F = field.
V is subtracted from rA. Overflow may occur as in ADD.

Note that because of the variable definition of byte size, overflow will occur in some MIX computers when it would not occur in others. We have not said that overflow will occur definitely if the value is greater than 1073741823; overflow occurs when the magnitude of the result is greater than the contents of five bytes, depending on the byte size. One can still write programs which work properly and which give the same final answers, regardless of the byte size.

• MUL (multiply). C = 3; F = field.
The 10-byte product of V times (rA) replaces registers A and X. The signs of rA and rX are both set to the algebraic sign of the result (i.e., + if the signs of V and rA were the same, and − if they were different).

• DIV (divide). C = 4; F = field.
The value of rA and rX, treated as a 10-byte number, with the sign of rA, is divided by the value V. If V = 0 or if the quotient is more than five bytes in magnitude (this is equivalent to the condition that $|rA| \geq |V|$), registers A and X are filled with undefined information and the overflow toggle is set on. Otherwise the quotient is placed in rA and the remainder is placed in rX. The sign of rA afterward is the algebraic sign of the quotient; the sign of rX afterward is the previous sign of rA.

Examples of arithmetic instructions: In most cases, arithmetic is done only with MIX words which are single five-byte numbers, not packed with several fields. It is possible to operate arithmetically on packed MIX words, if some caution is used. The following examples should be studied carefully. (The "?" mark designates an unknown value.)

```
ADD   1000
```

+	1234	1	150		rA before
+	100	5	50		Cell 1000
+	1334	6	200		rA after

```
SUB   1000
```

−	1234	0	0	9	rA before
−	2000	150	0		Cell 1000
+	766	149	?		rA after

```
MUL   1000(1:1)
```

−					112	rA before
?	2	?	?	?	?	Cell 1000
−					0	rA after
−					224	rX after

```
MUL   1000
```

−	50	0	112		4	rA before
−	2	0	0	0	0	Cell 1000
+	100	0	224			rA after
+	8	0	0	0	0	rX after

```
DIV   1000
```

+					0	rA before
+					17	rX before
+					3	Cell 1000
+					5	rA after
+					2	rX after

```
DIV   1000
```

−					0	rA before
+	1235	0	3	0		rX before
−	0	0	0	2	0	Cell 1000
+	0	617	?	?		rA after
−	0	0	0	?	?	rX after

(These examples have been prepared with the philosophy that it is better to give a complete, baffling description than an incomplete, straightforward one.)

Address transfer operators. In the following operations, the (possibly indexed) "address" M is used as a signed number, not as the address of a cell in memory.

- `ENTA` (enter A). C = 48; F = 2.

The quantity M is loaded into rA. The action is equivalent to "`LDA`" from a memory word containing the signed value of M. If M = 0, the sign of the instruction is loaded.

Examples: "`ENTA 0`" sets rA to zeros. "`ENTA 0,1`" sets rA to the current contents of index register 1.

- `ENTX` (enter X). C = 55; F = 2.
- `ENT`*i* (enter *i*). C = 48 + *i*; F = 2.

Analogous to `ENTA`, loading the appropriate register.

- `ENNA` (enter negative A). C = 48; F = 3.
- `ENNX` (enter negative X). C = 55; F = 3.
- `ENN`*i* (enter negative *i*). C = 48 + *i*; F = 3.

Same as `ENTA`, `ENTX`, and `ENT`*i*, except that the opposite sign is loaded.

Example: "`ENN3 0,3`" replaces rI3 by its negative.

- `INCA` (increase A). C = 48; F = 0.

The quantity M is added to rA; the action is equivalent to "`ADD`" from a memory word containing the value of M. Overflow is possible and it is treated just as in `ADD`.

Example: "`INCA 1`" increases the value of rA by one.

- `INCX` (increase X). C = 55; F = 0.

The quantity M is added to rX. If overflow occurs, the action is equivalent to `ADD`, except that rX is used instead of rA. Register A is never affected by this instruction.

- `INC`*i* (increase *i*). *C* = 48 + *i*; *F* = 0.

Add M to rI*i*. Overflow must not occur; if the magnitude of the result is more than two bytes, the result of this instruction is undefined.

- `DECA` (decrease A). C = 48; F = 1.
- `DECX` (decrease X). C = 55; F = 1.
- `DEC`*i* (decrease *i*). C = 48 + *i*; F = 1.

These eight instructions are the same as `INCA`, `INCX`, and `INC`*i*, respectively, except that M is subtracted from the register rather than added.

Note that the operation code C is the same for `ENTA`, `ENNA`, `INCA`, and `DECA`; the F-field is used to distinguish the various operations in this case.

Comparison operators. The comparison operators all compare the value contained in a register with a value contained in memory. The comparison indicator

is then set to LESS, EQUAL, or GREATER according to whether the value of the *register* is less than, equal to, or greater than the value of the *memory cell*. A minus zero is *equal to* a plus zero.

- CMPA (compare A). C = 56; F = field.

The specified field of A is compared with the *same* field of CONTENTS(M). If the field F does not include the sign position, the fields are both thought of as positive; otherwise the sign is taken into account in the comparison. (If F is (0:0) an equal comparison always occurs, since minus zero equals plus zero.)

- CMPX (compare X). C = 63; F = field.

This is analogous to CMPA.

- CMP*i* (compare *i*). $C = 56 + i$; F = field.

Analogous to CMPA. Bytes 1, 2, and 3 of the index register are treated as zero in the comparison.

Jump operators. Ordinarily, instructions are executed in sequential order; i.e., the instruction executed after the one in location P is the instruction found in location P + 1. Several "jump" instructions allow this sequence to be interrupted. Whenever a jump of any kind takes place, the J-register is set to the address of the next instruction (that is, the address of the instruction which would have been next if we hadn't jumped). A "store J" instruction then can be used by the programmer, if desired, to set the address field of another command which will later be used to return to the original place in the program. The J-register is changed whenever a jump actually occurs in a program (except JSJ) and it is never changed except when a jump occurs.

- JMP (jump). C = 39; F = 0.

Unconditional jump: the next instruction is taken from location M.

- JSJ (jump, save J). C = 39; F = 1.

Same as JMP except that the contents of rJ are unchanged.

- JOV (jump on overflow). C = 39; F = 2.

If the overflow toggle is on, it is turned off and a JMP occurs; otherwise nothing happens.

- JNOV (jump on no overflow). C = 39; F = 3.

If the overflow toggle is off, a JMP occurs; otherwise it is turned off.

- JL, JE, JG, JGE, JNE, JLE (jump on less, equal, greater, greater-or-equal, unequal, less-or-equal). C = 39; F = 4, 5, 6, 7, 8, 9, respectively.

Jump if the comparison indicator is set to the condition indicated. For example, JNE will jump if the comparison indicator is LESS or GREATER. The comparison indicator is not changed by these instructions.

- JAN, JAZ, JAP, JANN, JANZ, JANP (jump A negative, zero, positive, nonnegative, nonzero, nonpositive). C = 40; F = 0, 1, 2, 3, 4, 5, respectively.

If the contents of rA satisfy the stated condition, a JMP occurs, otherwise nothing

happens. "Positive" means *greater* than zero (not zero); "nonpositive" means the opposite, i.e., zero or negative.

• JXN, JXZ, JXP, JXNN, JXNZ, JXNP (jump X negative, zero, positive, nonnegative, nonzero, nonpositive). C = 47; F = 0, 1, 2, 3, 4, 5, respectively.

• J*i*N, J*i*Z, J*i*P, J*i*NN, J*i*NZ, J*i*NP (jump i negative, zero, positive, nonnegative, nonzero, nonpositive). C = 40 + i; F = 0, 1, 2, 3, 4, 5, respectively.
These are analogous to the corresponding operations for rA.

Miscellaneous operators.

• MOVE. C = 7; F = number.
The number of words specified by F is moved, starting from location M to the location specified by the contents of index register 1. The transfer occurs one word at a time, and rI1 is increased by the value of F at the end of the operation. If F = 0, nothing happens.

Care must be taken when the groups of locations involved overlap; for example, suppose that F = 3 and M = 1000. Then if (rI1) = 999, we transfer (1000) to (999), (1001) to (1000), and (1002) to (1001). Nothing unusual occurred here; but if (rI1) were 1001 instead, we would move (1000) to (1001), then (1001) to (1002), then (1002) to (1003), so we have moved the *same* word (1000) into three places.

• SLA, SRA, SLAX, SRAX, SLC, SRC (shift left A, shift right A, shift left AX, shift right AX, shift left AX circularly, shift right AX circularly). C = 6; F = 0, 1, 2, 3, 4, 5, respectively.

These are the "shift" commands. Signs of registers A, X are not affected in any way. M specifies the number of *bytes* to be shifted left or right; M must be nonnegative. SLA and SRA do not affect rX; the other shifts affect both registers as though they were a single 10-byte register. With SLA, SRA, SLAX, and SRAX, zeros are shifted into the register at one side, and bytes disappear at the other side. The instructions SLC and SRC call for a "circulating" shift, in which the bytes that leave one end enter in at the other end. Both rA and rX participate in a circulating shift.

Examples:

	Register A						Register X					
Initial contents	+	1	2	3	4	5	−	6	7	8	9	10
SRAX 1	+	0	1	2	3	4	−	5	6	7	8	9
SLA 2	+	2	3	4	0	0	−	5	6	7	8	9
SRC 4	+	6	7	8	9	2	−	3	4	0	0	5
SRA 2	+	0	0	6	7	8	−	3	4	0	0	5
SLC 501	+	0	6	7	8	3	−	4	0	0	5	0

• NOP (no operation). C = 0.
No operation occurs, and this instruction is bypassed. F and M are ignored.

• HLT (halt). C = 5; F = 2.
The machine stops. When the computer operator restarts it, the net effect is equivalent to NOP.

Input-output operators. MIX has a fair amount of input-output equipment (all of which is optional at extra cost). Each device is given a number as follows:

Unit number	Peripheral device	Record size
t	Tape unit no. t $(0 \le t \le 7)$	100 words
d	Disk or drum unit no. d $(8 \le d \le 15)$	100 words
16	Card reader	16 words
17	Card punch	16 words
18	Printer	24 words
19	Typewriter and paper tape	14 words

Not every MIX installation will have all of this equipment available; we will occasionally make appropriate assumptions about the presence of certain devices. Some devices may not be used both for input and for output. The number of words mentioned in the above table is a fixed record size associated with each unit.

Input or output with magnetic tape, disk, or drum units reads or writes full words (five bytes plus sign). Input or output with cards or typewriter, however, is always done in a *character code* where each byte represents one alphameric character. Thus, five characters per MIX word are transmitted. The character code is given at the top of Table 1, which appears at the close of this section and on the end papers of this book. The code 00 corresponds to "␣", which denotes a *blank space*. Codes 01–29 are for the letters A through Z with a few Greek letters thrown in; codes 30–39 represent the digits 0, 1, . . . , 9; and further codes 40, 41, . . . represent punctuation marks and other special characters. It is not possible to read in or write out all possible values a byte may have, since certain combinations are undefined. Not all input-output devices are capable of handling all the symbols in the character set; for example, the symbols Φ and Π which appear amid the letters will perhaps not be acceptable to the card reader. When input of character code is being done, the signs of all words are set to "+"; on output, signs are ignored.

The disk and drum units are large external memory devices each containing b^2 100-word records, where b is the byte size. On every IN, OUT, or IOC instruction as defined below, the particular 100-word record referred to by the instruction is specified by the current contents of the two least significant bytes of rX.

• IN (input). C = 36; F = unit.
This instruction initiates the transfer of information from the input unit specified into sequential locations starting with M. The number of locations transferred

is the record size for this unit (see the table above). The machine will wait at this point if a preceding operation for the same unit is not yet complete. The transfer of information which starts with this instruction will not be complete until a "jump ready" or "jump busy" instruction (see below) indicates the unit is again ready, so a program must not refer to the information in memory until this time. It is improper to attempt to read any record from magnetic tape which follows the latest record written on that tape.

• `OUT` (output). C = 37; F = unit.

This instruction starts the transfer of information from memory locations starting at M to the output unit specified. (The machine waits until the unit is ready, if it is not initially ready.) The transfer will not be complete until a "jump ready" or "jump busy" instruction (see below) indicates the unit is again ready, so a program must not alter the information in memory before this time.

• `IOC` (input-output control). C = 35; F = unit.

The machine waits, if necessary, until the specified unit is not busy. Then a control operation is performed, depending on the particular device being used. The following examples are used in various parts of this book:

Magnetic tape: If M = 0, the tape is rewound. If M < 0 the tape is skipped backward −M records, or to the beginning of the tape, whichever comes first. If M > 0, the tape is skipped forward; it is improper to skip forward over any records following the one last written on that tape.

For example, the sequence "`OUT 1000(3); IOC -1(3); IN 2000(3)`" writes out one hundred words onto tape 3, then reads it back in again. Unless the tape reliability is questioned, the last two instructions of that sequence are only a slow way to move words 1000–1099 to locations 2000–2099. The sequence "`OUT 1000(3); IOC +1(3)`" is improper.

Disk or drum: M should be zero. The effect is to position the device according to rX so that the next `IN` or `OUT` operation on this unit will take less time if it uses the same rX setting.

Printer: M should be zero. "`IOC 0(18)`" skips the printer to the top of the following page.

Paper tape reader: Rewind the tape. (M should be zero.)

• `JRED` (jump ready). C = 38; F = unit.

A jump occurs if the specified unit is ready, i.e., finished with the preceding operation initiated by `IN`, `OUT`, or `IOC`.

• `JBUS` (jump busy). C = 34; F = unit.

Same as `JRED` except the jump occurs under the opposite circumstances, i.e., when the specified unit is *not* ready.

Example: In location 1000, the instruction "`JBUS 1000(16)`" will be executed repeatedly until unit 16 is ready.

The simple operations above complete `MIX`'s repertoire of input-output instructions. There is no "tape check" indicator, etc., to cover exceptional

conditions on the peripheral devices. Any such condition (e.g., paper jam, unit turned off, out of tape, etc.) causes the unit to remain busy, a bell rings, and the skilled computer operator fixes things manually using ordinary maintenance procedures.

Conversion Operators.

• NUM (convert to numeric). C = 5; F = 0.

This operation is used to change the character code into numeric code. M is ignored. Registers A, X are assumed to contain a 10-byte number in character code; the NUM instruction sets the magnitude of rA equal to the numerical value of this number (treated as a decimal number). The value of rX and the sign of rA are unchanged. Bytes 00, 10, 20, 30, 40, . . . convert to the digit zero; bytes 01, 11, 21, . . . convert to the digit one; etc. Overflow is possible, and in this case the remainder modulo the word size is retained.

• CHAR (convert to characters). C = 5; F = 1.

This operation is used to change numeric code into character code suitable for output to cards or printer. The value in rA is converted into a 10-byte decimal number which is put into register A and X in character code. The signs of rA, rX are unchanged. M is ignored.

Examples:

	Register A						Register X					
Initial contents	−	00	00	31	32	39	+	37	57	47	30	30
NUM 0	−					12977700	+	37	57	47	30	30
INCA 1	−					12977699	+	37	57	47	30	30
CHAR 0	−	30	30	31	32	39	+	37	37	36	39	39

Timing. To give quantitative information as to how "good" MIX programs are, each of MIX's operations is assigned an *execution time* typical for present day computers.

ADD, SUB, all LOAD operations, all STORE operations (including STZ), all shift commands, and all comparison operations take *two units* of time. MOVE requires one unit plus two for each word moved. MUL requires 10 and DIV requires 12 units. Execution time for floating-point operations is unspecified. All remaining operations take one unit of time, plus the time the computer may be idle on the IN, OUT, IOC, or HLT instructions.

Note in particular that ENTA takes one unit of time, while LDA takes two units. The timing rules are easily remembered because of the fact that, except for shifts, MUL, and DIV, the number of units equals the number of references to memory (including the reference to the instruction itself).

The "unit" of time is a relative measure which we will denote simply by u. It may be regarded as, say, 10 microseconds (for a relatively inexpensive computer) or as 1 microsecond (for a relatively high-priced machine).

Example: Execution of the sequence

```
LDA  1000
INCA 1
STA  1000
```

takes a time of $5u$.

Summary. We have now discussed all of the features of MIX, except for its "GO button" which is discussed in exercise 26. Although MIX has nearly 150 different operations, they fit into a few simple patterns so they can be easily remembered. Table 1 summarizes the operations for each C-setting. The name of each operator is followed by the standard F-field for that operator in parentheses.

The following exercises give a good review of the material in this section; most of them are very simple, and the reader should try to do nearly all of them.

EXERCISES

1. [*00*] If MIX were a ternary (base 3) computer, how many "trits" would there be per byte?

2. [*02*] If a value to be represented within MIX may get as large as 99999999, how many adjacent bytes should be used to contain this quantity?

3. [*02*] Give the partial field specifications, (L:R), for the (a) address field, (b) index field, (c) field field, and (d) operation code field of a MIX instruction.

4. [*00*] The last example in (5) is "LDA -2000,4"—how can this be legitimate in view of the fact that memory addresses should not be negative?

5. [*10*] What is the symbolic notation [as in (4)] corresponding to the word (6)?

▶ **6.** [*10*] Assume that location 3000 contains

+	5	1	200	15

.

What is the result of the following instructions? (State if any of these are undefined or only partially defined.) (a) LDAN 3000; (b) LD2N 3000(3:4); (c) LDX 3000(1:3); (d) LD6 3000; (e) LDXN 3000(0:0).

7. [*15*] Give a precise definition of the results of the DIV instruction for all cases in which overflow does not occur, using the algebraic operations X mod Y and $\lfloor X \rfloor$.

8. [*15*] The last example of the DIV instruction which appears in the text has "rX before" equal to

+	1235	0	3	0

.

If this were

−	1234	0	3	0

instead, but other parts of that example were unchanged, what would registers A, X contain after the DIV instruction?

Table 1

Character code:

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
␣	A	B	C	D	E	F	G	H	I	Θ	J	K	L	M	N	O	P	Q	R	Φ	Π	S	T	U

00	*1*	**01**	*2*	**02**	*2*	**03**	*10*
No operation NOP(0)		rA ← rA + V ADD(0:5) FADD(6)		rA ← rA − V SUB(0:5) FSUB(6)		rAX ← rA × V MUL(0:5) FMUL(6)	
08	*2*	**09**	*2*	**10**	*2*	**11**	*2*
rA ← V LDA(0:5)		rI1 ← V LD1(0:5)		rI2 ← V LD2(0:5)		rI3 ← V LD3(0:5)	
16	*2*	**17**	*2*	**18**	*2*	**19**	*2*
rA ← −V LDAN(0:5)		rI1 ← −V LD1N(0:5)		rI2 ← −V LD2N(0:5)		rI3 ← −V LD3N(0:5)	
24	*2*	**25**	*2*	**26**	*2*	**27**	*2*
F(M) ← rA STA(0:5)		F(M) ← rI1 ST1(0:5)		F(M) ← rI2 ST2(0:5)		F(M) ← rI3 ST3(0:5)	
32	*2*	**33**	*2*	**34**	*1*	**35**	$1+T$
F(M) ← rJ STJ(0:2)		F(M) ← 0 STZ(0:5)		Unit F busy? JBUS(0)		Control, unit F IOC(0)	
40	*1*	**41**	*1*	**42**	*1*	**43**	*1*
rA:0, jump JA[+]		rI1:0, jump J1[+]		rI2:0, jump J2[+]		rI3:0, jump J3[+]	
48	*1*	**49**	*1*	**50**	*1*	**51**	*1*
rA ← [rA]? ± M INCA(0)DECA(1) ENTA(2)ENNA(3)		rI1 ← [rI1]? ± M INC1(0)DEC1(1) ENT1(2)ENN1(3)		rI2 ← [rI2]? ± M INC2(0)DEC2(1) ENT2(2)ENN2(3)		rI3 ← [rI3]? ± M INC3(0)DEC3(1) ENT3(2)ENN3(3)	
56	*2*	**57**	*2*	**58**	*2*	**59**	*2*
rA(F):V → CI CMPA(0:5) FCMP(6)		rI1(F):V → CI CMP1(0:5)		rI2(F):V → CI CMP2(0:5)		rI3(F):V → CI CMP3(0:5)	

General form:

C	*t*
Description OP(F)	

C = operation code, (5:5) field of instruction
F = op variant, (4:4) field of instruction
M = address of instruction after indexing
V = F(M) = contents of F field of location M
OP = symbolic name for operation
(F) = standard F setting
t = execution time; T = interlock time

25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55
V	W	X	Y	Z	0	1	2	3	4	5	6	7	8	9	.	,	(	)	+	−	*	/	=	$	<	>	@	;	:	'

04	*12*	05	*1*	06	*2*	07	*1 + 2F*
rA ← rAX/V rX ← remainder DIV(0:5) FDIV(6)		Special NUM(0) CHAR(1) HLT(2)		Shift M bytes SLA(0) SRA(1) SLAX(2) SRAX(3) SLC(4) SRC(5)		Move F words from M to rI1 MOVE(1)	
12	*2*	**13**	*2*	**14**	*2*	**15**	*2*
rI4 ← V LD4(0:5)		rI5 ← V LD5(0:5)		rI6 ← V LD6(0:5)		rX ← V LDX(0:5)	
20	*2*	**21**	*2*	**22**	*2*	**23**	*2*
rI4 ← −V LD4N(0:5)		rI5 ← −V LD5N(0:5)		rI6 ← −V LD6N(0:5)		rX ← −V LDXN(0:5)	
28	*2*	**29**	*2*	**30**	*2*	**31**	*2*
F(M) ← rI4 ST4(0:5)		F(M) ← rI5 ST5(0:5)		F(M) ← rI6 ST6(0:5)		F(M) ← rX STX(0:5)	
36	*1 + T*	**37**	*1 + T*	**38**	*1*	**39**	*1*
Input, unit F IN(0)		Output, unit F OUT(0)		Unit F ready? JRED(0)		Jumps JMP(0) JSJ(1) JOV(2)JNOV(3) also [*] below	
44	*1*	**45**	*1*	**46**	*1*	**47**	*1*
rI4:0, jump J4[+]		rI5:0, jump J5[+]		rI6:0, jump J6[+]		rX:0, jump JX[+]	
52	*1*	**53**	*1*	**54**	*1*	**55**	*1*
rI4 ← [rI4]? ± M INC4(0)DEC4(1) ENT4(2)ENN4(3)		rI5 ← [rI5]? ± M INC5(0)DEC5(1) ENT5(2)ENN5(3)		rI6 ← [rI6]? ± M INC6(0)DEC6(1) ENT6(2)ENN6(3)		rX ← [rX]? ± M INCX(0)DECX(1) ENTX(2)ENNX(3)	
60	*2*	**61**	*2*	**62**	*2*	**63**	*2*
rI4(F):V → CI CMP4(0:5)		rI5(F):V → CI CMP5(0:5)		rI6(F):V → CI CMP6(0:5)		rX(F):V → CI CMPX(0:5)	

rA = register A
rX = register X
rAX = registers AX as one
rI*i* = index reg. *i*, $1 \leq i \leq 6$
rJ = register J
CI = comparison indicator

[*]:		[+]:
JL(4)	<	N(0)
JE(5)	=	Z(1)
JG(6)	>	P(2)
JGE(7)	≧	NN(3)
JNE(8)	≠	NZ(4)
JLE(9)	≦	NP(5)

▸ **9.** [*20*] List all the MIX operators which can possibly affect the setting of the overflow toggle. (Do not include floating-point operators.)

10. [*20*] List all the MIX operators which can possibly affect the setting of the comparison indicators.

▸ **11.** [*20*] List all the MIX operators which can possibly affect the setting of rI1.

12. [*10*] Find a single instruction which has the effect of multiplying the current contents of rI3 by two and leaving the result in rI3.

▸ **13.** [*10*] Suppose location 1000 contains the instruction "JOV 1001". This instruction turns off the overflow toggle if it is on (and the next instruction executed will be in location 1001, in any case). If this instruction were changed to "JNOV 1001", would there be any difference? What if it were changed to "JOV 1000" or "JNOV 1000"?

14. [*25*] For each MIX operator, consider whether there is a way to set the ±AA-, I-, and F-portions of the instruction so that the result of the instruction is precisely equivalent to NOP, except that the execution time may be longer. Assume that nothing is known about the contents of any registers or any memory locations. Whenever it is possible to produce a NOP, state how it can be done. *Examples:* INCA is a no-op if the address and index parts are zero. JMP can never be a no-op, since it affects rJ.

15. [*10*] How many *alphameric characters* are there in a typewriter record? in a card-reader or card-punch record? in a printer record?

16. [*20*] Write a program which sets memory cells 0000–0099 all to zero, and which is (a) as short a program as possible; (b) as fast a program as possible. [*Hint:* Consider using the MOVE command.]

17. [*26*] This is the same as the previous exercise, except that locations 0000 through N, inclusive, are to be set to zero, where N is the current contents of rI2. The programs should work for any value $0 \leq N \leq 2999$; they should start in location 3000.

▸ **18.** [*22*] After the following "number one" program has been executed, what changes to registers, toggles, and memory have taken place? (For example, what is the final setting of rI1? of rX? of the overflow and comparison indicators?)

```
STZ  1
ENNX 1
STX  1(0:1)
SLAX 1
ENNA 1
INCX 1
ENT1 1
SRC  1
ADD  1
DEC1 -1
STZ  1
CMPA 1
MOVE -1,1(1)
NUM  1
CHAR 1
HLT  1
```

▸ **19.** [*14*] What is the execution time of the program in the preceding exercise, not counting the HLT instruction?

20. [*20*] Write a program which sets *all 4000* memory cells equal to a "HLT" instruction, and then stops.

▸ **21.** [*24*] (a) Can the J-register ever be zero? (b) Write a program which, given a number N in rI4, sets register J equal to N, assuming $0 < N \leq 3000$. Your program should start in location 3000. When your program has finished its execution, the contents of all memory cells must be unchanged.

▸ **22.** [*28*] Location 2000 contains an integer number, X. Write two programs which compute X^{13} and halt with the result in register A. One program should use the minimum number of MIX memory locations; the other should require the minimum execution time possible. Assume that X^{13} fits into a single word.

23. [*27*] Location 0200 contains a word

+	a	b	c	d	e

;

write two programs which compute the "reflected" word

+	e	d	c	b	a

and halt with the result in register A. One program should do this without using the "partial field" feature of MIX. Both programs should take the minimum possible number of memory locations under the stated conditions (including those locations used for the program and for temporary storage of intermediate results).

24. [*21*] Assuming registers A and X contain

+	0	a	b	c	d

and

+	e	f	g	h	i

,

respectively, write two programs which change the contents of these registers to

+	a	b	c	d	e

and

+	0	f	g	h	i

,

respectively, using (a) minimum memory space and (b) minimum execution time.

▸ **25.** [*30*] Suppose that the manufacturer of MIX wishes to come out with a more powerful computer ("Mixmaster"?), and he wants to convince as many as possible of those people now owning a MIX computer to invest in the more expensive machine. He wants to design this new hardware to be an *extension* of MIX, in the sense that all programs correctly written for MIX will work on the new machines without change. Suggest desirable things which could be incorporated in this extension. (For example, can you make better use of the I-field of an instruction?)

▸ **26.** [*32*] This problem is to write a card-loading routine. Every computer has its own peculiar problems for getting information initially into the machine and correctly started up, etc. In MIX's case, the contents of a card can only be read in character code,

and this *includes* the cards which contain the loading program itself. Not all possible byte values can be read from a card, and each word read in from cards is positive.

MIX has one feature that has not been explained in the text: There is a "GO-button," which is used to get the computer started from scratch when its memory contains arbitrary information. When this button is pushed by the computer operator, the following actions take place:

a) A single card is read into locations 0000–0015; this is essentially equivalent to the instruction "IN 0(16)".

b) When the card has been completely read and the card reader is no longer busy, a JMP to location 0000 occurs. The J-register is also set to zero.

c) The machine now begins to execute the program which it has read from the card.

(Note: Those MIX computers without card readers have their GO-button attached to the paper tape reader, unit 19, but in this problem we will assume the presence of a card reader, unit 16.)

The loading routine to be written must satisfy the following conditions:

a) The input deck begins with the loading routine, followed by information cards containing the numbers to be loaded, then a "transfer card" which shuts down the loading routine and jumps to the beginning of the program. The loading routine must fit onto *two cards*. You are to design a suitable transfer card.

b) The information cards have the following format:

Columns 1–5, ignored by the loading routine.
Column 6, the number of consecutive words to be loaded on this card (1 through 7).
Columns 7–10, the location of word 1, which is always greater than 100 (so it does not overlay the loading routine).
Columns 11–20, word 1.
Columns 21–30, word 2 (if column 6 $\geq$ 2).
. . .

Columns 71–80, word 7 (if column 6 = 7).

The information for word 1, word 2, . . . , is punched numerically as a decimal number. If the word is to be negative, a minus ("11-punch") is *overpunched* over the least significant digit, e.g., in column 20. Assume that this causes the character code input to be 10, 11, 12, . . . , 19, rather than 30, 31, 32, . . . , 39. For example, a card which has

ABCDE3100001234567890000000010000000$1\bar{0}$

punched in columns 1–40, should cause the following information to be loaded:

1000: +0123456789; 1001: +0000000001; 1002: −0000000100.

c) The loading routine should work for all byte sizes without any changes to the cards bearing the loading routine. No card should contain any of the characters corresponding to bytes 20, 21, 49, 50, . . . (i.e., the characters Φ, Π, $, <, . . .) since these characters cannot be read by all card readers. In particular, the instructions ENT1 and INC1 cannot be used (C = 49) since they cannot be punched on the card.

1.3.2. The MIX Assembly Language

A symbolic language is used to make MIX programs considerably easier to read and to write, and to save the programmer from worrying about tedious clerical details which often lead to unnecessary errors. This language, MIXAL ("MIX Assembly Language"), is an extension of the notation used for instructions in the previous section; the main features of this extension are the optional use of alphabetic names to stand for numbers, and a location field for associating names with memory locations.

MIXAL can be readily comprehended if we consider first a simple example. The following code is part of a larger program; it is a subroutine to find the maximum of n elements $X[1], \ldots, X[n]$, according to Algorithm 1.2.10M.

Program M (*Find the maximum*). Register assignments: rA $\equiv m$, rI1 $\equiv n$, rI2 $\equiv j$, rI3 $\equiv k$, $X[i] \equiv \text{cell}(\texttt{X} + i)$.

Assembled instructions						Line no.	LOC	OP	ADDRESS	Times	Remarks
						01	X	EQU	1000		
						02		ORIG	3000		
3000:	+	3009	0	2	32	*03*	MAXIMUM	STJ	EXIT	1	Subroutine linkage
3001:	+	0	1	2	51	*04*	INIT	ENT3	0,1	1	*M1. Initialize.* $k \leftarrow n$.
3002:	+	3005	0	0	39	*05*		JMP	CHANGEM	1	$j \leftarrow n$, $m \leftarrow X[n]$, $k \leftarrow n-1$.
3003:	+	1000	3	5	56	*06*	LOOP	CMPA	X,3	$n-1$	*M3. Compare.*
3004:	+	3007	0	7	39	*07*		JGE	*+3	$n-1$	
3005:	+	0	3	2	50	*08*	CHANGEM	ENT2	0,3	$A+1$	*M4. Change m.* $j \leftarrow k$.
3006:	+	1000	3	5	08	*09*		LDA	X,3	$A+1$	$m \leftarrow X[k]$.
3007:	+	1	0	1	51	*10*		DEC3	1	n	*M5. Decrease k.*
3008:	+	3003	0	2	43	*11*		J3P	LOOP	n	*M2. All tested?*
3009:	+	3009	0	0	39	*12*	EXIT	JMP	*	1	Return to main program. ∎

This program is an example of several things simultaneously:

a) The columns headed "LOC OP ADDRESS" are of principal interest; they contain a program in the MIXAL symbolic machine language, and we shall explain the details of this program below.

b) The column headed "Assembled instructions" shows the actual numeric machine language which corresponds to the MIXAL program. MIXAL has been designed so that it is a relatively simple matter to translate any MIXAL program into numeric machine language; this process may be carried out by another computer program called an *assembly program*. Thus, a programmer may do all of his "machine language" programming in MIXAL, never bothering to determine the equivalent numeric machine language himself. Virtually all MIX programs in this book are written in MIXAL. Chapter 9 includes a complete description of an assembly program which converts MIXAL programs to machine language in a form that is readily loaded into MIX's memory.

c) The column headed "Line no." is not an essential part of the MIXAL program; it is merely incorporated with the MIXAL programs of this book so that the text can refer to parts of the program.

d) The column headed "Remarks" gives explanatory information about the program, and it is cross-referenced to the steps of Algorithm 1.2.10M. The reader should refer to this algorithm. Note that a little "programmer's license" was used during the transcription of that algorithm into a MIX program; for example, step M2 has been put last. Note also the "register assignments" stated at the beginning of Program M; this shows what components of MIX correspond to the variables in the algorithm.

e) The column headed "Times" will be given for many of the MIX programs in this book; it represents the number of times the instruction on that line will be executed during the course of the program. Thus, line 6 will be performed $n - 1$ times, etc. From this information we can determine the length of time required to perform the subroutine; it is $(5 + 5n + 3A)u$, where A is the quantity which was carefully analyzed in Section 1.2.10.

Now we will discuss the MIXAL part of Program M. Line 1, "X EQU 1000", says that symbol X is to be *equivalent* to the number 1000. The effect of this may be seen on line 6, where the numeric equivalent of the instruction "CMPA X,3" appears as

+	1000	3	5	56

i.e., "CMPA 1000,3".

Line 2 says that the locations for succeeding lines should be chosen sequentially, originating with 3000. Therefore the symbol MAXIMUM which appears in the LOC field of line 3 becomes equivalent to the number 3000, INIT is equivalent to 3001, LOOP is equivalent to 3003, etc.

On lines 3 through 12 the OP field contains the symbolic names of MIX instructions STJ, ENT3, etc. The "OP" in lines 1 and 2, on the other hand, contains "EQU" and "ORIG" which are *not* MIX operators. They are called *pseudo-operators* because they appear only in the MIXAL symbolic program. Pseudo-operators are used to specify the form of a symbolic program; they are not instructions of the program itself. Thus the line "X EQU 1000" only talks *about* the program, it does not signify that any variable is to be set equal to 1000 when Program M is run. Note that no instruction is assembled for lines 1 and 2.

Line 3 is a "store J" instruction which stores the contents of register J into the (0:2) field of location "EXIT", i.e., into the address part of the instruction found on line 12.

As mentioned earlier, Program M is intended to be part of a larger program; elsewhere the sequence

```
ENT1 100
JMP  MAXIMUM
STA  MAX
```

would; for example, jump to Program M with n set to 100. Program M would then find the largest of the elements $X[1], \ldots, X[100]$ and would return to the

instruction "STA MAX" with the maximum value in rA and with its position, j, in rI2. (Cf. exercise 3.)

Line 5 jumps the control to line 8. Lines 4, 5, 6 need no further explanation. Line 7 introduces a new notation: an asterisk (read "self") refers to the location of this line; "*+3" ("self plus three") therefore refers to three locations past the current line. Since line 7 is an instruction which corresponds to location 3004, "*+3" appearing there refers to location 3007.

The rest of the symbolic code is self-explanatory; note the appearance of an asterisk again on line 12. (Cf. exercise 2.)

Our next example shows a few more features of the assembly language. The object is to print a table of the first 500 prime numbers, with 10 columns of 50 numbers each. The table should appear as follows:

```
FIRST FIVE HUNDRED PRIMES

    0002 0233 0547 0877 1229 1597 1993 2371 2749 3187
    0003 0239 0557 0881 1231 1601 1997 2377 2753 3191
    0005 0241 0563 0883 1237 1607 1999 2381 2767 3203
      :                                             :
    0229 0541 0863 1223 1583 1987 2357 2741 3181 3571
```

We shall use the following method.

Algorithm P (*Print table of 500 primes*). This algorithm has two distinct parts: steps P1–P8 prepare an internal table of 500 primes, and steps P9–P11 print the answer in the form shown above. The latter part of the program uses two "buffer" areas, i.e., sections of memory in which a line image is formed; while one buffer is being printed, the other is being filled.

P1. [Start table.] Set PRIME[1] ← 2, N ← 3, J ← 1. (N will run through the odd numbers which are candidates for primes; J keeps track of how many primes have been found so far.)

P2. [N is prime.] Set J ← J + 1, PRIME[J] ← N.

P3. [500 found?] If J = 500, go to step P9.

P4. [Advance N.] Set N ← N + 2.

P5. [K ← 2.] Set K ← 2. (PRIME[K] will run through the possible prime divisors of N.)

P6. [PRIME[K]\N?] Divide N by PRIME[K]; let Q be the quotient and R the remainder. If R = 0, N is not prime, so go to P4.

P7. [PRIME[K] large?] If Q ≤ PRIME[K], go to P2. (In such a case, N must be prime; the proof of this fact is interesting and a little unusual—see exercise 6.)

P8. [Advance K.] Increase K by 1, and go to P6.

P9. [Print title.] Now we are ready to print the table. Advance the printer to the next page. Set BUFFER[0] to the title line and print this line. Set B ← 1, M ← 1.

P10. [Set up line.] Put PRIME[M], PRIME[50 + M], . . . , PRIME[450 + M] in proper format into BUFFER[B].

P11. [Print line.] Print BUFFER[B]; set B ← 1 − B (thereby switching to the other buffer); and increase M by 1. If M ≤ 50, return to P10; otherwise the algorithm terminates. ∎

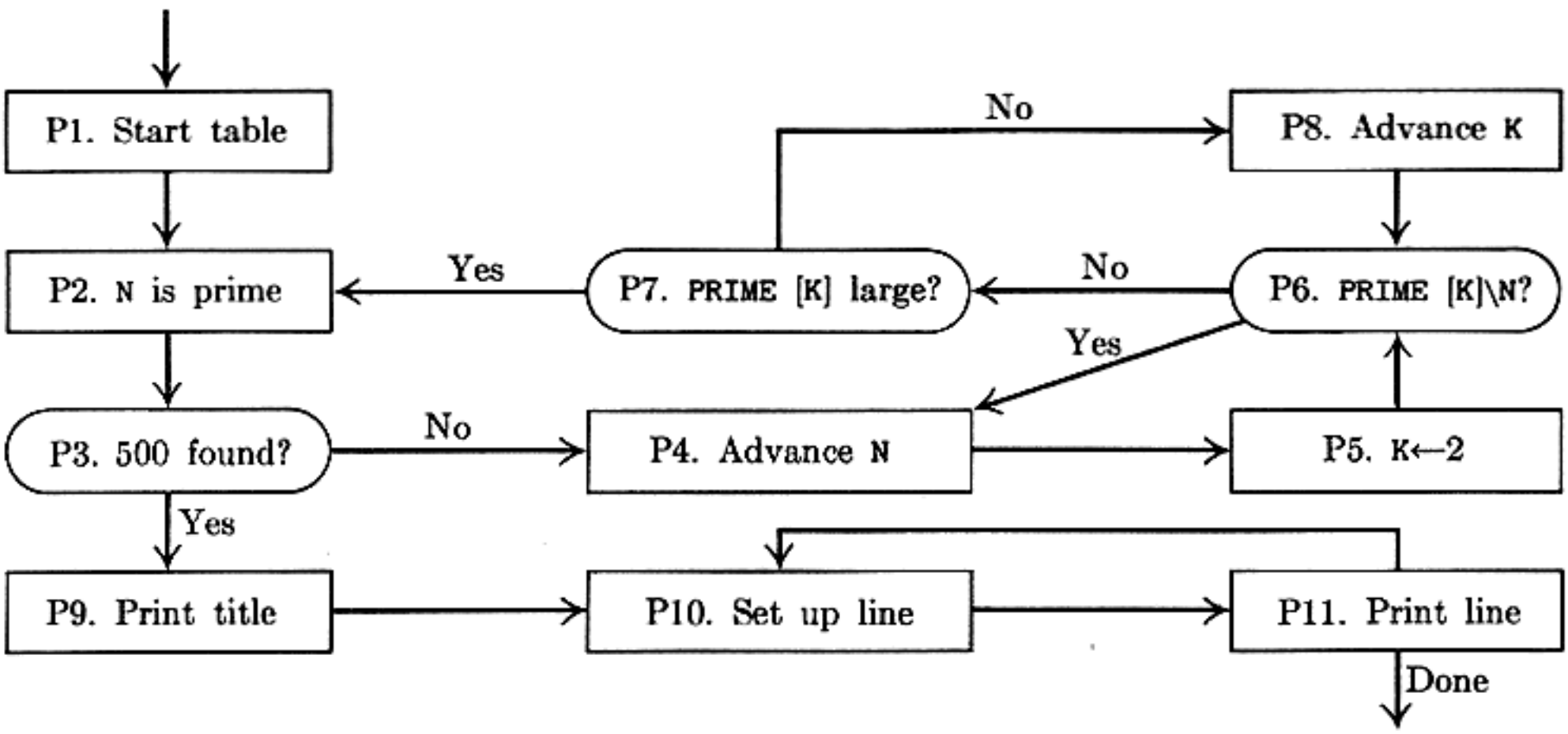


Fig. 14. Algorithm P.

Program P (*Print table of 500 primes*). This program has deliberately been written in a slightly clumsy fashion in order to illustrate most of the features of MIXAL in a single program. rI1 ≡ J − 500; rI2 ≡ N; rI3 ≡ K; rI4 indicates B; rI5 is M plus multiples of 50.

```
01      * EXAMPLE PROGRAM ... TABLE OF PRIMES
02      *
03      L       EQU   500          Number of primes to find
04      PRINTER EQU   18           Unit number of printer
05      PRIME   EQU   -1           Memory area for table of primes
06      BUF0    EQU   2000         Memory area for BUFFER[0]
07      BUF1    EQU   BUF0+25      Memory area for BUFFER[1]
08              ORIG  3000
09      START   IOC   0(PRINTER)   Skip to new page.
10              LD1   =1-L=        P1. Start table. J ← 1.
11              LD2   =3=              N ← 3.
12      2H      INC1  1            P2. N is prime. J ← J + 1.
13              ST2   PRIME+L,1        PRIME[J] ← N.
14              J1Z   2F           P3. 500 found?
```

```
15      4H      INC2    2           P4. Advance N.
16              ENT3    2           P5. K ← 2.
17      6H      ENTA    0           P6. PRIME[K]\N?
18              ENTX    0,2
19              DIV     PRIME,3
20              JXZ     4B              R = 0?
21              CMPA    PRIME,3     P7. PRIME[K] large?
22              INC3    1           P8. Advance K.
23              JG      6B              Jump if Q > PRIME[K].
24              JMP     2B              Otherwise N is prime.
25      2H      OUT     TITLE(PRINTER) P9. Print title.
26              ENT4    BUF1+10         Set B ← 1.
27              ENT5    -50             Set M ← 0.
28      2H      INC5    L+1             Advance M.
29      4H      LDA     PRIME,5     P10. Set up line. (Right to left)
30              CHAR
31              STX     0,4(1:4)
32              DEC4    1
33              DEC5    50              (rI5 goes down by 50 until
34              J5P     4B                  nonpositive)
35              OUT     0,4(PRINTER) P11. Print line.
36              LD4     24,4            Switch buffers.
37              J5N     2B              If rI5 = 0, we are done.
38              HLT
39      * INITIAL CONTENTS OF TABLES AND BUFFERS
40              ORIG    PRIME+1
41              CON     2               First prime is 2.
42              ORIG    BUF0-5
43      TITLE   ALF     FIRST           Alphabetic information for
44              ALF      FIVE               title line
45              ALF      HUND
46              ALF     RED P
47              ALF     RIMES
48              ORIG    BUF0+24
49              CON     BUF1+10         Each buffer refers to the other.
50              ORIG    BUF1+24
51              CON     BUF0+10
52              END     START           End of routine. ∎
```

The following points of interest are to be noted about this program:

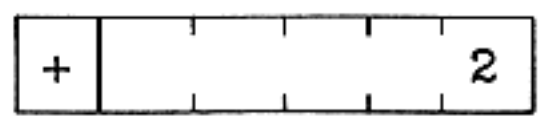
1. Lines 01, 02, and 39 begin with an asterisk: this signifies a "comment" line which is merely explanatory, having no actual effect on the assembled program.

2. As in Program M, the "EQU" in line 03 sets the equivalent of a symbol; in this case, the equivalent of L is set to 500. (In the program of lines 10–24,

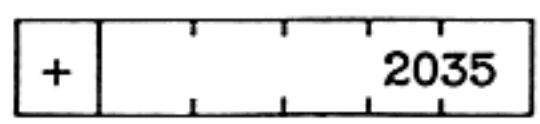
L represents the number of primes to be computed.) Note that in line 05 the symbol PRIME gets a *negative* equivalent; the equivalent of a symbol may be any five-byte-plus-sign number. In line 07 the equivalent of BUF1 is calculated as BUF0+25, namely 2025. MIXAL provides a limited amount of arithmetic on numbers; for another example, see line 13 where the value of PRIME+L (in this case, 499) is calculated by the assembly program.

3. Note that the symbol PRINTER has been used in the F-part on lines 25 and 35. The F-part, which is always enclosed in parentheses, may be a single number or symbol, or two numbers and/or symbols separated by a colon, as in the "1:4" of line 31.

4. MIXAL contains several ways to specify non-instruction words. Line 41 indicates an ordinary constant, "2", using the operation code CON; the result of line 41 is to assemble the word



Line 49 shows a slightly more complicated constant, "BUF1+10", which assembles as the word



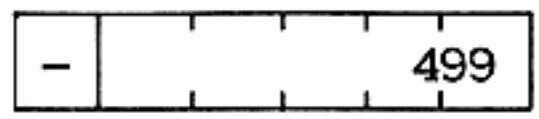
A constant may be enclosed in equal signs and it then becomes a *literal constant* (see lines 10 and 11). The assembler automatically creates internal names and inserts "CON" lines for literal constants. For example, lines 10 and 11 of Program P would effectively be changed to

```
10          LD1   con1
11          LD2   con2
```

and then at the end of the program, between lines 51 and 52, the lines

```
51a  con1   CON   1-L
51b  con2   CON   3
```

are effectively inserted as part of the assembly procedure for literal constants. Line 51a will assemble into the word



The use of literal constants is a decided convenience, because it means that the programmer does not have to invent a name for the constant and that he does not have to insert that constant at the end of the program; he can keep his mind on the central problems and not worry about such routine matters while writing his programs. Of course, in Program P we did not make an

especially good use of literal constants, since lines 10 and 11 would more properly be written "`ENT1 1-L; ENT2 3`"!

5. A good assembly language should mimic the way a programmer *thinks* about writing programs, so he can express himself fluently. One example of this philosophy is the use of literal constants, as we have just mentioned; another example is the use of "`*`", which was explained in Program M. A third example is the idea of *local symbols* such as the symbol `2H`, which appears in the location field of lines 12, 25, and 28.

Local symbols are special symbols whose equivalents can be *redefined* as many times as desired. A symbol like `PRIME` has but one significance throughout a program, and if it were to appear in the location field of more than one line an error would be indicated by the assembly program. Local symbols have a different nature; we write, for example, `2H` ("2 here") in the location field, and `2F` ("2 forward") or `2B` ("2 backward") in the address field of a `MIXAL` line:

`2B` means the closest *previous* location `2H`
`2F` means the closest *following* location `2H`

As examples, the "`2F`" in line 14 refers to line 25; the "`2B`" in line 24 refers back to line 12; and the "`2B`" in line 37 refers to line 28. An address of `2F` or `2B` never refers to the *same* line; e.g., the three lines

```
2H  EQU  10
2H  MOVE 2F(2B)
2H  EQU  2B-3
```

are virtually equivalent to the single line

```
MOVE *-3(10).
```

The symbols `2F`, `2B` are never to be used in the location field, and `2H` is never to be used in the address field. There are ten local symbols, which can be obtained by replacing "2" in the above examples by any digit from 0 to 9.

The idea of local symbols was introduced by M. E. Conway in 1958, in connection with an assembly program for the UNIVAC 1. Local symbols spare the programmer from the necessity to think of a symbolic name for an address, when all he wants to do is refer to an instruction a few lines away. When making reference to a nearby location in the program there often is no appropriate name with much significance, so programmers have tended to use symbols like `X1`, `X2`, `X3`, etc.; this leads to the danger of using the same symbol twice. That is why the reader will soon find that the use of local symbols comes naturally to him when he writes `MIXAL` programs, if he is not already familiar with this idea.

6. In lines 30 and 38 the address part is blank. This means the address is to be zero. Similarly, we could have left the address blank in line 17, but the program would have been less readable.

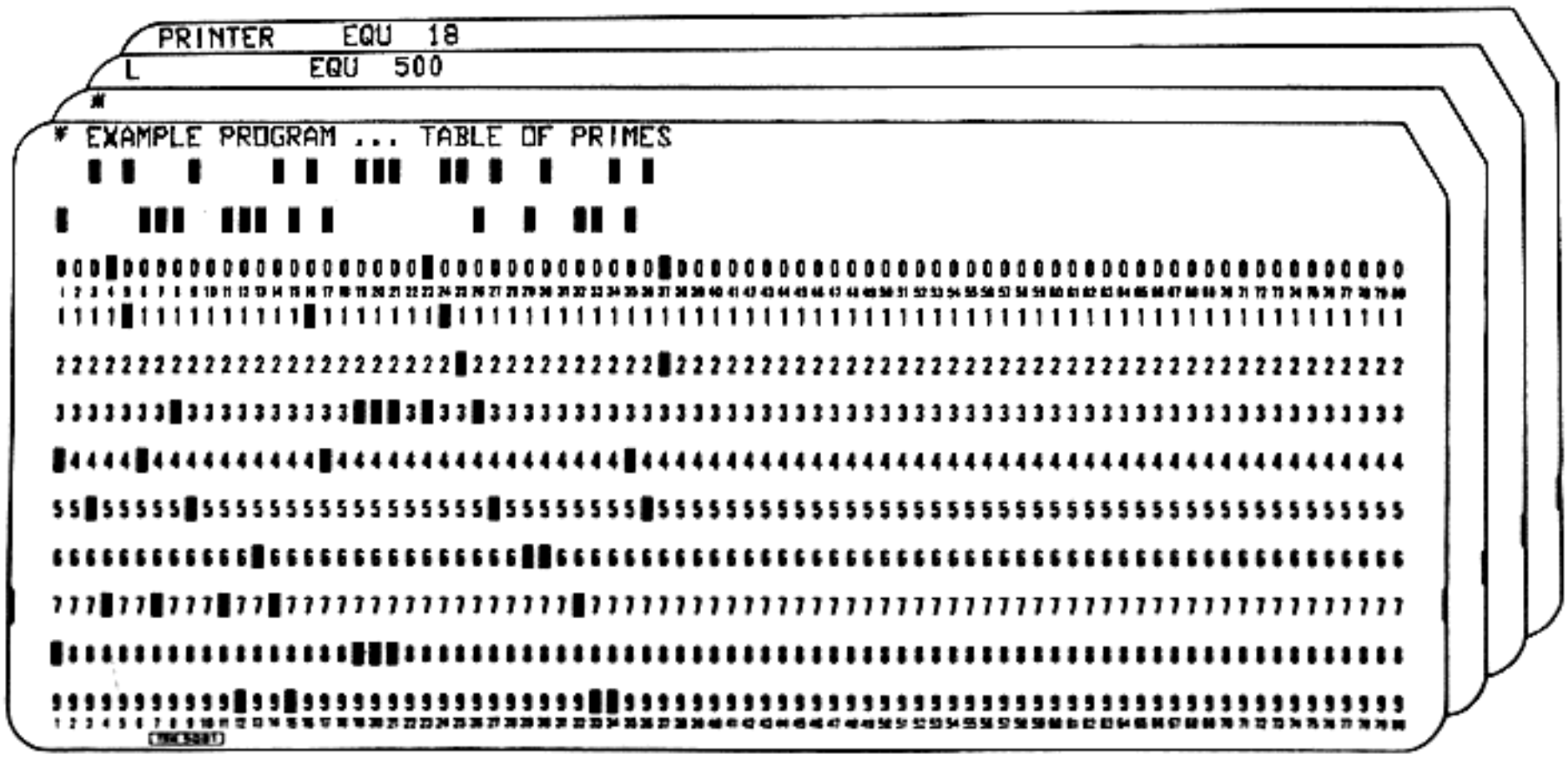


Fig. 15. The first four lines of Program M punched onto cards.

7. Lines 43–47 use the "ALF" operation, which creates a five-byte constant in MIX alphameric character code. For example, line 45 causes the word

+	00	08	24	15	04

to be assembled.

These lines are used as the first 25 characters of the title line. *All locations whose contents are not specified in the* MIXAL *program are ordinarily set to zero* (except the locations which are used by the loading routine, usually 3700–3999); thus there is no need to set the other words of the title line to blanks.

8. Note that arithmetic may be used on ORIG lines, e.g., lines 40, 42, and 48.

9. The last line of a complete MIXAL program always has the operation code END. The address on this line is the location at which the program is to begin once it has been loaded into memory.

10. As a final note about Program P, the reader may observe how the coding has been written so that index registers are counted towards zero, and tested against zero, whenever possible. For example, the quantity J–500, not J, is kept in rI1. Lines 26–34 are particularly noteworthy, although perhaps a bit tricky.

It may be of interest to note a few of the statistics observed when Program P was actually run. The division instruction in line 19 was executed 9538 times; the time to perform lines 10–24 was 182144u.

MIXAL programs can be punched onto cards, as shown in Fig. 15. The following format is used:

Columns 1–10	LOC (location) field.
Columns 12–15	OP field,
Columns 17–80	ADDRESS field and optional remarks,
Columns 11, 16	blank.

However, if column 1 contains an asterisk, the entire card is treated as a comment. The ADDRESS field ends with the first blank column following column 16; any explanatory information may be punched to the right of this first blank column with no effect on the assembled program. (*Exception:* When the OP field is "ALF", the remarks always start in column 22.)

The MIX assembly program (see Section 9.3) accepts card decks prepared in this manner and converts them to machine language programs in loadable form. Under favorable circumstances the reader will have access to a MIX assembly program and MIX simulator, on which various exercises in this book can be worked out.

Now we have seen what can be done in MIXAL. We conclude this section by describing the rules more carefully, and in particular we shall observe what is *not* allowed in MIXAL. The following comparatively few rules define the language.

1. A *symbol* is a string of one to ten letters and/or digits, containing at least one letter. *Examples:* PRIME TEMP 20BY20. The special symbols *d*H, *d*F, *d*B, where *d* is a single digit, will for the purposes of this definition be replaced by other unique symbols according to the "local symbol" convention described above.

2. A *number* is a string of one to ten digits. *Example:* 00052.

3. Each appearance of a symbol in a MIXAL program is said to be either a "defined symbol" or a "future reference." A *defined symbol* is a symbol which has appeared in the LOC field of a preceding line of this MIXAL program. A *future reference* is a symbol which has not yet been defined in this way.

4. An *atomic expression* is either

a) a number, or
b) a defined symbol (denoting the numerical equivalent of that symbol, see rule 13), or
c) an asterisk (denoting the value of ⊛; see rules 10 and 11).

5. An *expression* is either

a) an atomic expression, or
b) a plus or minus sign followed by an atomic expression, or
c) an expression followed by a binary operation followed by an atomic expression.

The six admissible binary operations are +, −, *, /, //, : ; they are defined on numeric MIX words as follows:

```
C = A+B     LDA A; ADD B; STA C
C = A-B     LDA A; SUB B; STA C
C = A*B     LDA A; MUL B; STX C
C = A/B     LDA A; SRAX 5; DIV B; STA C
C = A//B    LDA A; ENTX 0; DIV B; STA C
C = A:B     LDA A; MUL =8=; SLAX 5; ADD B; STA C.
```

Operations within an expression are carried out from left to right. *Examples:*

`-1+5`	equals 4
`-1+5*20/6`	equals 4*20/6 equals 80/6 equals 13 (going from left to right)
`1//3`	equals a `MIX` word whose value is approximately $(b^5/3)$ where b is the byte size; i.e., a word representing the fraction $\frac{1}{3}$ with decimal point at the left.
`1:3`	equals 11 (usually used in partial field specification)
`*-3`	equals ⊛ minus three
`***`	equals ⊛ times ⊛!

6. An *A-part* (which is used to describe the address field of a `MIX` instruction) is either

a) vacuous (denoting the value zero), or
b) an expression, or
c) a future reference (denoting the eventual equivalent of the symbol, see rule 13).

7. An *index part* (which is used to describe the index field of a `MIX` instruction) is either

a) vacuous (denoting the value zero), or
b) a comma followed by an expression (denoting the value of that expression).

8. An *F-part* (which is used to describe the F-field of a `MIX` instruction) is either

a) vacuous (denoting the *standard* F-setting, based on the context), or
b) a left parenthesis followed by an expression followed by a right parenthesis (denoting the value of the expression).

9. A *W-value* (which is used to describe a *full-word* `MIX` constant) is either

a) an expression followed by an F-part [in this case a vacuous F-part denotes (0:5)], or
b) a W-value followed by a comma followed by a W-value of the form (a).

A W-value denotes the value of a numeric `MIX` word determined as follows: Let the W-value have the form "$\mathtt{E}_1(\mathtt{F}_1), \mathtt{E}_2(\mathtt{F}_2), \ldots, \mathtt{E}_n(\mathtt{F}_n)$" where $n \geq 1$, the E's are expressions, and the F's are fields. The desired result is the final value which would appear in memory location `CON` if the following hypothetical program were executed: "`STZ CON; LDA` $\mathtt{C}_1$; `STA CON(`$\mathtt{F}_1$`); . . . ; LDA` $\mathtt{C}_n$; `STA CON(`$\mathtt{F}_n$`)`". Here $\mathtt{C}_1, \ldots, \mathtt{C}_n$ denote locations containing the values of expressions $\mathtt{E}_1, \ldots, \mathtt{E}_n$. Each $\mathtt{F}_i$ must have the form $8\mathtt{L}_i + \mathtt{R}_i$ where $0 \leq \mathtt{L}_i \leq \mathtt{R}_i \leq 5$. *Examples:*

`1`	is the word	+|||||1
`1,-1000(0:2)`	is the word	-|1000|||1
`-1000(0:2),1`	is the word	-|1000|||1

10. The assembly process makes use of a value denoted by ⊛ (called the *location counter*) which is initially zero. The value of ⊛ should always be a

nonnegative number which can fit in two bytes. When the location field of a line is not blank, it must contain a symbol which has not been previously defined. The equivalent of that symbol is then defined to be the current value of ⊛.

11. After processing the LOC field as described in rule 10, the assembly process depends on the value of the OP field. There are six possibilities for OP:

a) OP is a symbolic MIX operator (see Table 1 at the end of the previous section). The chart defines the standard C and F values for this operator. In this case the ADDRESS should be an A-part (rule 6), followed by an index part (rule 7), followed by an F part (rule 8). We thereby obtain four values: C, F, A, and I; the effect is to assemble the word determined by the sequence "LDA C; STA WORD; LDA F; STA WORD(4:4); LDA I; STA WORD(3:3); LDA A; STA WORD(0:2)" into the location specified by ⊛, and to advance ⊛ by 1.

b) OP is "EQU". The ADDRESS should be a W-value (see rule 9); if the LOC field is nonblank, the equivalent of the symbol appearing there is set equal to the value specified in ADDRESS. This rule takes precedence over rule 10. The value of ⊛ is unchanged. (As a nontrivial example, consider the line

```
BYTESIZE  EQU  1(4:4)
```

which allows the programmer to have a symbol whose value depends on the byte size. This is an acceptable situation so long as the resulting program is meaningful with each possible byte size.)

c) OP is "ORIG". The ADDRESS should be a W-value (see rule 9); the location counter, ⊛, is set to this value. (Note that because of rule 10, a symbol appearing in the LOC field of an ORIG card gets as its equivalent the value of ⊛ before it has changed. *Example:*

```
TABLE     ORIG *+100
```

sets the equivalent of TABLE to the *first* of 100 locations.)

d) OP is "CON". The ADDRESS should be a W-value; the effect is to assemble a word, having this value, into the location specified by ⊛, and to advance ⊛ by 1.

e) OP is "ALF". The effect is to assemble the word of character codes formed by columns 17–21 of the card, otherwise behaving like CON.

f) OP is "END". The ADDRESS should be a W-value, which specifies in its (4:5) field the location of the instruction at which the program begins. The END card signals the end of a MIXAL program. Lines are inserted, in arbitrary order, corresponding to all undefined symbols and literal constants (see rules 12 and 13).

12. Literal constants: A W-value of 9 characters or less in length may be enclosed between "=" signs and used as a future reference. The effect is as though a new symbol were created and inserted just before the END card (see remark 4 following Program P).

13. Every symbol has one and only one equivalent value; this is a full-word MIX number which is either determined by the symbol's appearance in LOC according to rule 10 or rule 11(b), or else a line, having the name of the symbol in LOC with OP = "CON" and ADDRESS = "0", is effectively inserted before the END card.

Note: The most significant consequence of the above rules is the restriction on future references. A symbol which has not been defined in the LOC field of a previous card may not be used except as the A-part of an instruction. In particular, it may not be used (a) in connection with arithmetic operations; or (b) in the ADDRESS field of EQU, ORIG, or CON. For example,

```
LDA  2F+1     and     CON  3F
```

are both illegal. This restriction has been imposed in order to allow more efficient assembly of programs, and the experience gained in writing this set of books has shown that it is a very mild restriction which rarely makes much difference.

EXERCISES—First set

1. [*00*] The text remarked that "`X EQU 1000`" does not indicate any instruction which sets the value of a variable. Suppose that you are writing a `MIX` program in which you wish to set the value contained in a certain memory cell (whose symbolic name is `X`) equal to 1000. How could you write this in `MIXAL`?

▶ **2.** [*10*] Line 12 of Program M says "`JMP *`"; since * denotes the location of the line, why doesn't the program go into an infinite loop, endlessly repeating this instruction?

▶ **3.** [*23*] What is the effect of the following program, if it is used in conjunction with Program M?

```
START   IN      X+1
        JBUS    *
        ENT1    100
1H      JMP     MAXIMUM
        LDX     X,1
        STA     X,1
        STX     X,2
        DEC1    1
        J1P     1B
        OUT     X+1(1)
        HLT
        END     START
```

▶ **4.** [*25*] Assemble Program P by hand; i.e., what are the actual numerical contents of memory, corresponding to that symbolic program?

5. [*HM46*] Analyze the time required for steps P1–P8 of Algorithm P to compute the first T primes, as a function of T.

6. [*M20*] (a) Show that if n is not prime, n has a divisor d with $1 < d \leq \sqrt{n}$. (b) Use this fact to show that the test in step P7 of Algorithm P proves that N is not prime.

7. [*10*] What is the meaning of "4B" in line 34 of Program P? What effect, if any, would be caused if the location of line 15 were changed to "2H" and the address of line 20 were changed to "2B"?

▸ **8.** [*24*] What does the following program do? (Do not run it on a computer, figure it out by hand!)

```
* MYSTERY PROGRAM
PRINTER  EQU   18
BUF      ORIG  *+3000
1H       ENT1  1
         ENT2  0
         LDX   4F
2H       ENT3  0,1
3H       STZ   BUF,2
         INC2  1
         DEC3  1
         J3P   3B
         STX   BUF,2
         INC2  1
         INC1  1
         CMP1  =75=
         JL    2B
         ENN2  2400
         OUT   BUF+2400,2(PRINTER)
         INC2  24
         J2N   *-2
         HLT
4H       ALF   AAAAA
         END   1B
```

EXERCISES—Second set

These exercises are short programming problems, representing typical computer applications and covering a wide range of techniques. It is recommended that each reader choose a few of these programs, in order to get some experience using MIX as well as a good review of basic programming skills. If desired, these exercises may be worked concurrently as the rest of Chapter 1 is being read.

The following list indicates the types of programming methods which arise in the exercises below:

Use of switching tables (multiway decisions): exercises 9 and 23.
Use of index registers; two-dimensional arrays: exercises 10, 21, 22, and 23.
Unpacking characters: exercises 13 and 23.

Integer and scaled decimal arithmetic: exercises 14, 16 and 18.
Real-time control: exercise 20.
Graphical display: exercise 23.
Input buffering: exercise 13.
Output buffering: exercises 21 and 23.
Use of subroutines: exercises 14 and 20.

Whenever an exercise in this book says, "write a MIX program" or "write a MIX subroutine", it suffices to write only the symbolic code for what is asked, which will only be a fragment of a larger program; perhaps no input or output is done in this fragment, etc. One need only write LOC, OP, and ADDRESS fields of MIXAL lines (possibly also remarks, especially if someone else is to be grading the solutions!), but not the numeric machine language, line no., or "times" columns unless requested to do so.

If the exercise says, "Write a *complete* MIX program," it implies that an executable program is to be written in MIXAL (including in particular the final END card); hopefully, an assembler and MIX simulator on which complete programs can be tested will be available to most readers.

▸ **9.** [*25*] Location INST contains a MIX word which purportedly is a MIX instruction. Write a program which jumps to location GOOD if the word has a valid C-field, valid ±AA-field, valid I-field, and valid F-field, and which jumps to location BAD otherwise. Remember that the test for a valid F-field depends on the C-field; for example, if C = 7 (MOVE), any F-field is acceptable, but if C = 8 (LDA), the F-field must have the form 8L + R where $0 \le L \le R \le 5$. The "±AA"-field is to be considered valid *unless* C specifies an instruction requiring a memory address, I = 0, and ±AA is not a valid memory address.

Note: Inexperienced programmers tend to tackle a problem like this by writing a long series of tests on C, e.g., `LDA C; JAZ 1F; DECA 5; JAN 2F; JAZ 3F; DECA 2; JAN 4F`; etc. This is *not* good practice! Whenever a multiway decision such as this is to be made, it is best to prepare an auxiliary *table* containing information which facilitates the desired decisions. If there were, for example, a table of 64 entries, we could write "`LD1 C; LD1 TABLE,1; JMP 0,1`"—thereby jumping very speedily to the desired routine. Other information can also be kept in such a table. The tabular approach in this case makes the program only a little bit longer (including the table) and it greatly increases the speed.

▸ **10.** [*31*] Assume that we have a 9×8 matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{18} \\ a_{21} & a_{22} & a_{23} & \ldots & a_{28} \\ \vdots & & & & \vdots \\ a_{91} & a_{92} & a_{93} & \ldots & a_{98} \end{pmatrix}$$

stored in memory so that a_{ij} is in location $1000 + 8i + j$. In memory the matrix therefore appears as follows:

$$\begin{pmatrix} (1009) & (1010) & (1011) & \ldots & (1016) \\ (1017) & (1018) & (1019) & \ldots & (1024) \\ \vdots & & & & \vdots \\ (1073) & (1074) & (1075) & \ldots & (1080) \end{pmatrix}.$$

A matrix is said to have a "saddle point" if some position is the smallest value in its row and the largest value in its column. In symbols, a_{ij} is a saddle point if

$$a_{ij} = \min_{1\le k\le 8} a_{ik} = \max_{1\le k\le 9} a_{kj}.$$

Write a MIX program which computes

a) the location of a saddle point, if there is at least one;
b) zero, if there is no saddle point;
and which then stops with this value in rI1.

11. [*M29*] What is the *probability* that the matrix in the preceding exercise has a saddle point, assuming that the 72 elements are distinct and assuming that all 72! arrangements are equally probable? What is the probability if we assume instead that the elements of the matrix are zeros and ones, and all 2^{72} such matrices are equally probable?

12. [*M50*] The "answers to the exercises" give two solutions to exercise 10, and suggest a third solution, and it is not clear which of the given solutions is better. Analyze the algorithms, using each of the assumptions of exercise 11, and decide which is the better method.

13. [*28*] A cryptanalyst wants a frequency count of the letters in a certain code. The code has been punched on paper tape; the end is signaled by an asterisk. Write a complete program which reads in the tape, counts the frequency of each character up to the first asterisk, and then types out the results in the form

```
A  0010257
B  0000179
D  0794301
```

etc., one character per line. The number of blanks should not be counted, nor should characters for which the count is zero (e.g., C in the above) be printed. For efficiency, "buffer" the input, i.e., while reading a record into one area of memory you can be counting characters from another area. You may assume that an extra record (following that which contains the terminating asterisk) is present on the input tape.

▸ **14.** [*31*] The following algorithm, due to the Neapolitan astronomer Aloysius Lilius and the German Jesuit mathematician Christopher Clavius in the late 16th century, is used by most Western churches to determine the date of Easter Sunday for any year after 1582. [For previous years, see *CACM* **5** (1962), 209–210. The first systematic algorithm for calculating the date of Easter was the *canon paschalis* due to Victorius of Aquitania (457 A.D.). There are many indications that the sole important application of arithmetic in Europe during the Middle Ages was the calculation of Easter date, and so such algorithms are historically significant. For further commentary, see *Puzzles and Paradoxes* by T. H. O'Beirne, (London: Oxford University Press, 1965), Chapter 10.]

Algorithm E. (*Date of Easter*). Let Y be the year for which the date of Easter is desired.

E1. [Golden number.] Set $G \leftarrow (Y \bmod 19) + 1$. ($G$ is the so-called "golden number" of the year in the 19-year Metonic cycle.)

E2. [Century.] Set $C \leftarrow \lfloor Y/100 \rfloor + 1$. (When Y is not a multiple of 100, C is the century number; i.e., 1970 is in the twentieth century.)

E3. [Corrections.] Set $X \leftarrow \lfloor 3C/4 \rfloor - 12$, $Z \leftarrow \lfloor (8C + 5)/25 \rfloor - 5$. ($X$ is the number of years, such as 1900, in which leap year was dropped in order to keep in step with the sun. Z is a special correction designed to synchronize Easter with the moon's orbit.)

E4. [Find Sunday.] Set $D \leftarrow \lfloor 5Y/4 \rfloor - X - 10$. [March $((-D) \bmod 7)$ actually will be a Sunday.]

E5. [Epact.] Set $E \leftarrow (11G + 20 + Z - X) \bmod 30$. If $E = 25$ and the golden number G is greater than 11, or if $E = 24$, then increase E by 1. (E is the so-called "epact," which specifies when a full moon occurs.)

E6. [Find full moon.] Set $N \leftarrow 44 - E$. If $N < 21$ then set $N \leftarrow N + 30$. (Easter is supposedly the "first Sunday following the first full moon which occurs on or after March 21." Actually perturbations in the moon's orbit do not make this strictly true, but we are concerned here with the "calendar moon" rather than the actual moon. The Nth of March is a calendar full moon.)

E7. [Advance to Sunday.] Set $N \leftarrow N + 7 - ((D + N) \bmod 7)$.

E8. [Get month.] If $N > 31$, the date is $(N - 31)$APRIL; otherwise the date is N MARCH. ∎

Write a subroutine to calculate and print Easter date given the year, assuming the year is less than 100000. (The output should have the form "*dd* MONTH, *yyyyy*" where *dd* is the day, *yyyyy* is the year.) Write a complete MIX program which uses this subroutine to prepare a table of the dates of Easter from 1950 through 2000.

15. [*M30*] A fairly common error in the coding of the previous exercise is to fail to realize that the quantity $(11G + 20 + Z - X)$ in step E5 may be negative, and so the positive remainder mod 30 is sometimes not computed. (See *CACM* **5** (1962), 556.) For example, in the year 14250 we would find $G = 1$, $X = 95$, $Z = 40$; so if we had $E = -24$ instead of $E = +6$ we would get the ridiculous answer "37 APRIL". Write a complete program which finds the *earliest* year for which this error would actually cause the wrong date to be calculated for Easter.

16. [*31*] We showed in Section 1.2.7 that the sum $1 + \frac{1}{2} + \frac{1}{3} + \cdots$ becomes infinitely large. But if it is calculated with finite accuracy by a computer, the sum actually exists, in some sense, because the terms eventually get so small they contribute nothing to the sum if added one by one. For example, suppose we calculate the sum by rounding to one decimal place; then we have $1 + 0.5 + 0.3 + 0.3 + 0.2 + 0.2 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 = 3.9$.

More precisely, let $r_n(x)$ be the number x rounded to n decimal places; we define $r_n(x) = \lfloor 10^n x + \frac{1}{2} \rfloor / 10^n$. Then we wish to find

$$S_n = r_n(1) + r_n(\tfrac{1}{2}) + r_n(\tfrac{1}{3}) + \cdots;$$

we know that $S_1 = 3.9$, and the problem is to write a complete MIX program which calculates and prints S_n for $n = 2$, 3, 4, and 5.

Note: There is a much faster way to do this than the simple procedure of adding $r_n(1/m)$, one number at a time, until $r_n(1/m)$ becomes zero. (For example, we have

$r_5(1/m) = 0.00001$ for all values of m from 6667 to 200000. It is a good idea to save calculating $1/m$ all 193,334 times!) An algorithm along the following lines should rather be used:

A. Start with $m_h = 1$, $S = 1$.
B. Set $m_e = m_h + 1$ and calculate $r_n(1/m_e) = r$.
C. Find m_h, the largest m for which $r_n(1/m) = r$.
D. Add $(m_h - m_e + 1)r$ to S and return to step B.

17. [*M38*] Using the notation of the preceding exercise, prove or disprove

$$\lim_{n\to\infty} (S_{n+1} - S_n) = \ln 10.$$

18. [*25*] The ascending sequence of all reduced fractions between 0 and 1 which have denominators $\leq n$ is called the "Farey series of order n." For example, the Farey series of order 7 is

$$\tfrac{0}{1}, \tfrac{1}{7}, \tfrac{1}{6}, \tfrac{1}{5}, \tfrac{1}{4}, \tfrac{2}{7}, \tfrac{1}{3}, \tfrac{2}{5}, \tfrac{3}{7}, \tfrac{1}{2}, \tfrac{4}{7}, \tfrac{3}{5}, \tfrac{2}{3}, \tfrac{5}{7}, \tfrac{3}{4}, \tfrac{4}{5}, \tfrac{5}{6}, \tfrac{6}{7}, \tfrac{1}{1}.$$

If we denote this series by $x_0/y_0, x_1/y_1, x_2/y_2, \ldots$, it can be shown that

$$\begin{aligned} x_0 = 0,\quad y_0 = 1;\qquad x_1 = 1,\quad y_1 = n;\\ x_{k+2} = \lfloor (y_k + n)/y_{k+1} \rfloor x_{k+1} - x_k; \qquad (*)\\ y_{k+2} = \lfloor (y_k + n)/y_{k+1} \rfloor y_{k+1} - y_k. \end{aligned}$$

Write a MIX subroutine which computes the Farey series of order n, by storing the values of x_k and y_k in locations X + k, Y + k, respectively. (The total number of terms in the series is approximately $3n^2/\pi^2$ so you may assume n is rather small.)

19. [*M30*] (a) Show that the numbers x_k, y_k defined by (*) in the preceding exercise satisfy the relation $x_{k+1}y_k - x_k y_{k+1} = 1$. (b) Show that the numbers x_k, y_k given by (*) are indeed the Farey series of order n, using the fact proved in (a).

▶ **20.** [*33*] Assume the X-register and the overflow toggle of MIX have been wired up to the traffic signals at the corner of Del Mar Boulevard and Berkeley Avenue, as follows:

rX(2:2) = Del Mar traffic light
rX(3:3) = Berkeley traffic light
} 0 off, 1 green, 2 amber, 3 red;

rX(4:4) = Del Mar pedestrian light
rX(5:5) = Berkeley pedestrian light
} 0 off, 1 "WALK", 2 "DON'T WALK";

Cars or pedestrians wishing to travel on Berkeley across the boulevard must trip a switch which causes the overflow toggle of MIX to go on. If this condition never occurs, the light for Del Mar should remain green.

Cycle times are as follows:

Del Mar traffic light is green $\geq$ 30 sec, amber 8 sec;
Berkeley traffic light is green 20 sec, amber 5 sec.

When a traffic light is green or amber for one direction, the other direction has a red light. When the traffic light is green, the corresponding "walk" light is on; except that

28	19	10	01	48	39	30
29	27	18	09	07	47	38
37	35	26	17	08	06	46
45	36	34	25	16	14	05
04	44	42	33	24	15	13
12	03	43	41	32	23	21
20	11	02	49	40	31	22

Fig. 16. A magic square.

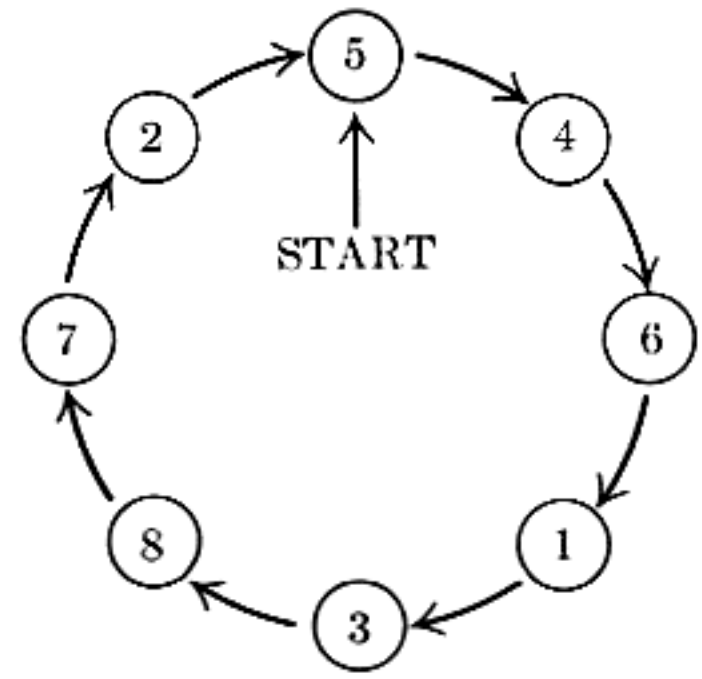


Fig. 17. Josephus' problem, $n = 8$, $m = 4$.

before the green light turns to amber, the "don't walk" light flashes for 12 sec, as follows:

DON'T WALK $\frac{1}{2}$ sec
off $\frac{1}{2}$ sec
} repeat 8 times;
DON'T WALK 4 sec (and remains on through amber and red cycles).

If the overflow is tripped while the Berkeley light is green, the car or pedestrian will pass on that cycle, but if it is tripped during the amber or red portions, another cycle will be necessary after the Del Mar traffic has passed.

Assume that one MIX time unit equals 10 μsec. Write a complete MIX program which controls this traffic light by manipulating rX, according to the input given by the overflow toggle. The stated times are to be followed exactly unless it is impossible to do so. *Note:* The setting of rX changes precisely at the *completion* of a LDX or INCX instruction. *Further note:* Do not worry about the economic unfeasability of the exercise.

21. [*28*] A *magic square of order* n is an arrangement of the numbers 1 through n^2 in a square array so that the sum of each row, column, and diagonal is the same, as well as the sum of the two main diagonals. Figure 16 shows a magic square of order 7. The rule for generating it is easily seen: Start with 1 in the middle of the top row, then go up and to the left diagonally (when running off the edge imagine an entire plane tiled with squares) until reaching a filled square; then drop down one space and continue. This method works whenever n is odd.

Using memory allocated in a fashion like that of exercise 10, write a complete MIX program to generate the 23×23 magic square by the above method; then print out this magic square. [The above algorithm was brought from Siam to France by S. de La Loubère in 1687. For numerous other interesting magic square constructions, many of which are good programming exercises, see W. W. Rouse Ball, *Mathematical Recreations and Essays*, rev. by H. S. M. Coxeter (New York: Macmillan, 1962), Chapter 7.]

22. [*31*] "Josephus' problem." There are n men arranged in a circle. Beginning at a particular position, we count around the circle and brutally execute every mth man (the circle closing as men are decapitated). For example, the execution order when

$n = 8$, $m = 4$ is 54613872, as shown in Fig. 17. Write a complete MIX program which prints out the order of execution when $n = 24$, $m = 11$. Try to design a clever algorithm which works at high speed when n and m are large (it may save your life). *Reference:* W. W. Rouse Ball, as in the previous exercise, pp. 32–36.

23. [*37*] This is an exercise designed to give some experience in the many applications of computers for which the output is to be displayed graphically rather than in the usual tabular form. In this case, the object is to "draw" a crossword puzzle diagram.

You are given as input a matrix of zeros and ones. An entry of zero indicates a white square; a one indicates a black square. The output should be a diagram of the puzzle, with the appropriate squares numbered for words "across" and "down."

For example, given the matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix},$$

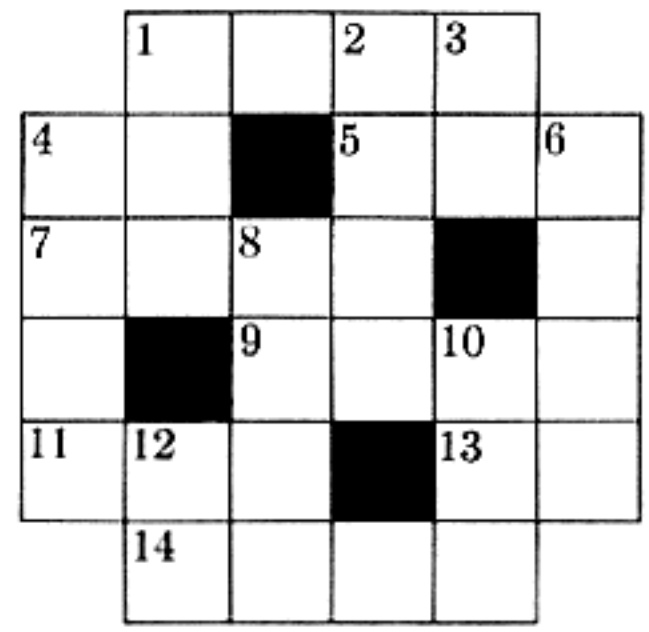


Fig. 18. Diagram corresponding to the matrix in exercise 23.

the corresponding puzzle diagram would be as shown in Fig. 18. A square is numbered if it is a white square and either (a) the square below it is white and there is no white square immediately above, or (b) there is no white square immediately to its left and the square to its right is white. If black squares are given at the edges, they should be removed from the diagram. This is illustrated in Fig. 18, where the black squares at the corners were dropped. A simple way to accomplish this is to artificially insert rows and columns of -1's at the top, bottom, and sides of the given input matrix, and then to change every "$+1$" which is adjacent to a "-1" into "-1" until no "$+1$" remains next to any "-1".

The following method should be used to print the final diagram: Each box of the puzzle should correspond to 5 columns and 3 rows of the output page. These 15 positions should be filled as follows:

Unnumbered white squares: ␣␣␣␣+ / ␣␣␣␣+ / +++++

Number nn white squares: nn␣␣+ / ␣␣␣␣+ / +++++

Black squares: +++++ / +++++ / +++++

"-1" squares, depending on whether there are -1's to the right or below:

␣␣␣␣+ / ␣␣␣␣+ / +++++ ; ␣␣␣␣+ / ␣␣␣␣+ / ␣␣␣␣+ ; ␣␣␣␣␣ / ␣␣␣␣␣ / +++++ ; ␣␣␣␣␣ / ␣␣␣␣␣ / ␣␣␣␣+ ; ␣␣␣␣␣ / ␣␣␣␣␣ / ␣␣␣␣␣

The diagram shown in Fig. 18 would then be printed as shown in Fig. 19.

The width of a printer line—120 characters—is enough to allow up to 23 columns in the crossword puzzle. The data supplied as input will be a 23×23 matrix of zeros and ones, each row punched in columns 1–23 of an input card. In the above example, the first card would have been punched "10000111111111111111111".

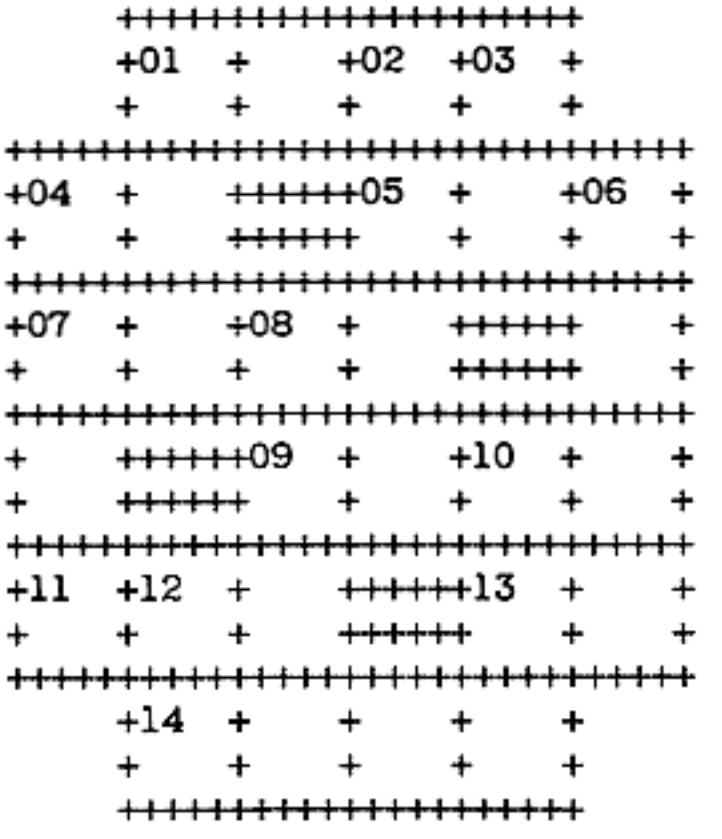


Fig. 19. Representation of Fig. 18 for printer output.

1.3.3. Applications to Permutations

In this section we shall give several more examples of MIX programs, and at the same time some important properties of permutations will be introduced. These investigations will also bring out some interesting aspects of computer programming in general.

Permutations were discussed earlier in Section 1.2.5; we treated the permutation *cdfbea* as an *arrangement* of the six objects a, b, c, d, e, f in a straight line. Another viewpoint is possible: We may think of a permutation as a *rearrangement* or renaming of the objects. With this interpretation it is customary to use a two-line notation, for example,

$$\begin{pmatrix} a & b & c & d & e & f \\ c & d & f & b & e & a \end{pmatrix}, \tag{1}$$

to mean "a becomes c, b becomes d, c becomes f, d becomes b, e becomes e, f becomes a." Considered as a rearrangement, this means object c moves to the place formerly occupied by object a; considered as a renaming, this means object a is renamed c. The two-line notation is unaffected by changes in the order of the columns; i.e., the permutation (1) could also be written

$$\begin{pmatrix} c & d & f & b & a & e \\ f & b & a & d & c & e \end{pmatrix}$$

and 718 other ways.

A *cycle notation* is often used in connection with this interpretation. Permutation (1) could be written

$$(a\ \ c\ \ f)(b\ \ d), \tag{2}$$

again meaning "a becomes c, c becomes f, f becomes a, b becomes d, d becomes b." A cycle $(x_1\ x_2\ \ldots\ x_n)$ means "x_1 becomes x_2, $\ldots$, x_{n-1} becomes x_n, x_n becomes x_1." Since e is "fixed" under the permutation, it does not appear in the cycle notation; that is, singleton cycles like "(e)" are conventionally not written. If

a permutation fixes *all* elements, so there are only singleton cycles present, it is called the *identity permutation*, and it is customarily denoted by "(1)" for no really good reason.

The cycle notation is not unique; for example,

$$(b\ \ d)(a\ \ c\ \ f), \qquad (c\ \ f\ \ a)(b\ \ d), \qquad (d\ \ b)(f\ \ a\ \ c), \tag{3}$$

etc., are all equivalent to (2). However, "$(a\,f\,c)(b\,d)$" is not the same, since it says a goes to f.

It is easy to see why the cycle notation is always possible. Starting with any element x_1, the permutation takes x_1 into x_2, say, and x_2 into x_3, etc., until finally (since there are only finitely many elements) we get to some element x_{n+1} which has already appeared among $x_1, \ldots, x_n$. Now x_{n+1} must equal x_1, for if it were equal to, say, x_3, we already know x_2 goes into x_3 and by assumption $x_n \neq x_2$ goes to x_{n+1}. So we have a cycle $(x_1\, x_2 \ldots x_n)$, $n \geq 1$, as part of our permutation. If this does not account for the entire permutation, we find another element y_1 and in the same way get another cycle $(y_1\, y_2 \ldots y_m)$. None of the y's can equal any of the x's, since $x_i = y_j$ implies that $x_{i+1} = y_{j+1}$, etc., and we would ultimately find $x_k = y_1$ for some k, contradicting the choice of y_1. All cycles will eventually be found in this way.

The application of these concepts to programming comes up whenever some set of n objects is to be rearranged. To rearrange these objects without auxiliary storage, we must essentially follow the cycle structure. For example, to do the rearrangement (1), i.e., to set

$$(a, b, c, d, e, f) \leftarrow (c, d, f, b, e, a),$$

we would essentially follow the cycle structure (2) and successively set

$$t \leftarrow a, \qquad a \leftarrow c, \qquad c \leftarrow f, \qquad f \leftarrow t; \qquad t \leftarrow b, \qquad b \leftarrow d, \qquad d \leftarrow t.$$

It is frequently useful to realize that any such transformation takes place in disjoint cycles like this.

Products of permutations. We can "multiply" two permutations together, with the understanding that multiplication means the application of one permutation after the other. For example, if permutation (1) is followed by the permutation

$$\begin{pmatrix} a & b & c & d & e & f \\ b & d & c & a & f & e \end{pmatrix},$$

we have a becomes c which then becomes c; b becomes d which becomes a; etc.:

$$\begin{pmatrix} a & b & c & d & e & f \\ c & d & f & b & e & a \end{pmatrix} \times \begin{pmatrix} a & b & c & d & e & f \\ b & d & c & a & f & e \end{pmatrix} = \begin{pmatrix} a & b & c & d & e & f \\ c & d & f & b & e & a \end{pmatrix} \times \begin{pmatrix} c & d & f & b & e & a \\ c & a & e & d & f & b \end{pmatrix}$$

$$= \begin{pmatrix} a & b & c & d & e & f \\ c & a & e & d & f & b \end{pmatrix}. \tag{4}$$

It should be clear that multiplication of permutations is not commutative, i.e., $\pi_1 \times \pi_2$ is not necessarily equal to $\pi_2 \times \pi_1$ when π_1 and π_2 are permutations. The reader may verify that the product in (4) gives a different result if the two factors are interchanged (see exercise 3).

Some people multiply permutations from right to left rather than the somewhat more natural left-to-right order shown in (4). In fact, mathematicians are divided into two camps in this regard; should the result of applying transformation T_1, then T_2, be denoted by T_1T_2 or by T_2T_1? Here we use T_1T_2.

Equation (4) would be written as follows, using the cycle notation:

$$(a\ \ c\ \ f)(b\ \ d)(a\ \ b\ \ d)(e\ \ f) = (a\ \ c\ \ e\ \ f\ \ b). \tag{5}$$

Note that the multiplication sign "$\times$" is conventionally dropped; this does not conflict with the cycle notation since it is easy to see that the permutation $(a\ \ c\ \ f)(b\ \ d)$ is really the product of the permutations $(a\ \ c\ \ f)$ and $(b\ \ d)$.

Multiplication of permutations can be done directly in terms of the cycle notation. For example, to compute the product of several permutations

$$(a\ \ c\ \ f\ \ g)(b\ \ c\ \ d)(a\ \ e\ \ d)(f\ \ a\ \ d\ \ e)(b\ \ g\ \ f\ \ a\ \ e), \tag{6}$$

we find (proceeding from left to right) that "a goes to c, then c goes to d, then d goes to a, then a goes to d, then d is unchanged"; so the net result is that a goes to d under (6), and we write down "$(a\ \ d$" as the partial answer. Now we consider the effect on d: "d goes to b goes to g," and we have the partial result "$(a\ \ d\ \ g$". Considering g, we find that "g goes to a, to e, to f, to a" and so the first cycle is closed, "$(a\ \ d\ \ g)$". Now pick a new element which hasn't appeared yet, say c; we find that c goes to e, and the reader may verify that ultimately the answer "$(a\ \ d\ \ g)(c\ \ e\ \ b)$" is obtained for (6).

Let us now try to do this process by computer. The following algorithm formalizes the method described in the preceding paragraph, in a way that is amenable to machine calculation.

Algorithm A (*Multiply permutations in cycle form*). This algorithm takes a product of cycles, such as (6), and computes the resulting permutation in the form of a product of disjoint cycles. We do not describe the removal of singleton cycles, which is a fairly simple extension of the algorithm. As this algorithm is performed, we successively "tag" the elements of the input formula, i.e., mark somehow those symbols of the input formula which have essentially been processed.

A1. [First pass.] Tag all left parentheses, and replace all right parentheses by a tagged copy of the element following their matching left parentheses. (See the example in Table 1.)

A2. [Open.] Searching from left to right, find the first untagged element of the input. (If all elements are tagged, the algorithm terminates.) Set START equal to it; output a left parenthesis; output the element; and tag it.

Table 1

ALGORITHM A APPLIED TO (6).

An "x" indicates a tagged element.

After step no.	START	CURRENT	(	*a*	*c*	*f*	*g*	*a*	(	*b*	*c*	*d*	*b*	(	*a*	*e*	*d*	*a*	(	*f*	*a*	*d*	*e*	*f*	(	*b*	*g*	*f*	*a*	*e*	*b*	Output
A1			x					x	x				x	x				x	x					x	x						x	
A2	*a*		x	x↑				x	x				x	x				x	x					x	x						x	(*a*
A3	*a*	*c*	x	x↑				x	x				x	x				x	x					x	x						x	
A4	*a*	*c*	x	x				x	x		x↑		x	x				x	x					x	x						x	
⋯																																	
A4	*a*	*d*	x	x				x	x		x		x	x			x↑	x	x					x	x						x	
⋯																																	
A4	*a*	*a*	x	x				x	x		x		x	x			x	x	x		x↑			x	x						x	
⋯																																	
A5	*a*	*d*	x	x				x	x		x		x	x			x	x	x		x			x	x						x↑	*d*
⋯																																	
A5	*a*	*g*	x	x				x	x		x	x	x	x			x	x	x		x			x	x	x					x↑	*g*
⋯																																	
A5	*a*	*a*	x	x			x	x	x		x	x	x	x	x		x	x	x		x		x		x	x	x	x		x	x↑	
A6	*a*	*a*	x	x			x	x	x		x	x	x	x	x		x	x	x		x		x		x	x	x	x		x	x↑	)
⋯																																	
A2	*c*	*a*	x	x	x↑		x	x	x		x	x	x	x	x		x	x	x		x		x		x	x	x	x		x		(*c*
⋯																																	
A5	*c*	*e*	x	x	x		x	x	x		x	x	x	x	x		x	x	x	x	x		x		x	x	x	x		x	x↑	*e*
⋯																																	
A5	*c*	*b*	x	x	x		x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x↑	*b*
⋯																																	
A6	*c*	*c*	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x↑	)
⋯																																	
A6	*f*	*f*	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x↑	(*f*)

A3. [Set CURRENT.] Set CURRENT equal to the next element of the formula.

A4. [Scan formula.] Proceed to the right until either reaching the end of the formula, or finding an element equal to CURRENT; in the latter case, tag it and go to step A3.

A5. [CURRENT = START?] If CURRENT ≠ START, output CURRENT and go back to step A4 starting again at the left of the formula (thereby continuing the development of a cycle in the output).

A6. [Close.] (A complete cycle in the output has been found.) Output a right parenthesis, and go back to step A2. ∎

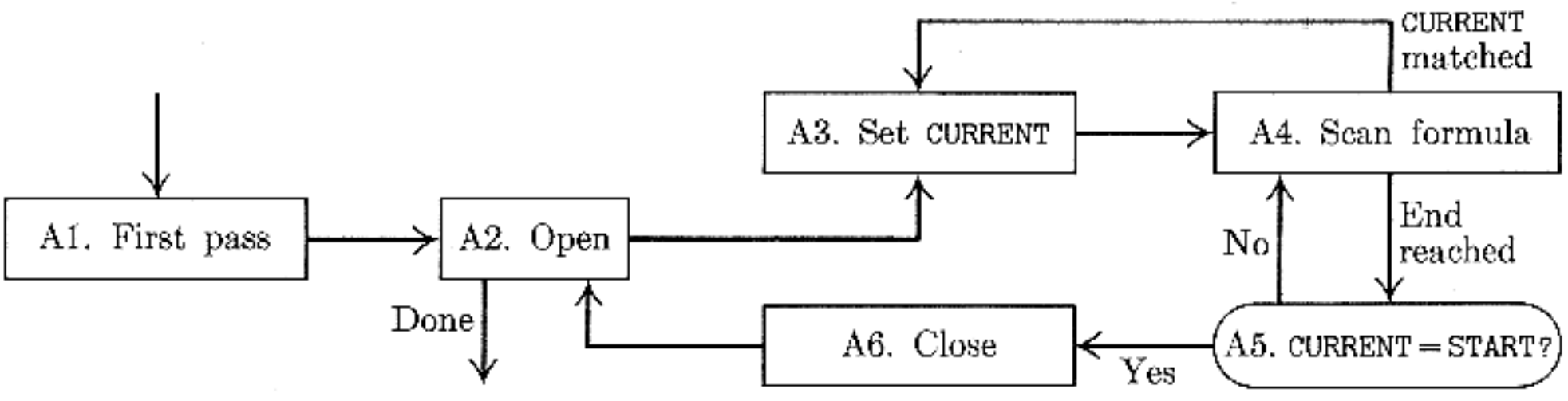


Fig. 20. Algorithm A for multiplying permutations.

For example, consider formula (6); Table 1 shows successive stages in its processing. The first line of that table shows the formula after right parentheses have been replaced by the leading element of the corresponding cycle; succeeding lines of the table show which elements have been tagged. An arrow shows the current point of interest in the formula. The output is "$(a\ d\ g)(c\ e\ d)(f)$"; note that singleton cycles will appear in the output.

A MIX program. To implement this algorithm for MIX, the "tagging" can be done by using the sign of a word. Suppose our input is punched onto cards in the following format: An 80-column card is divided into 16 five-character fields. Each field is either (a) "␣␣␣␣(", representing the left parenthesis beginning a cycle; (b) ")␣␣␣␣", representing the right parenthesis ending a cycle; (c) "␣␣␣␣␣", all blanks, which may be inserted anywhere to fill space; or (d) anything else, representing an element to be permuted. The last card of the input is recognized by having columns 76–80 equal to "␣␣␣␣=". For example, (6) might be punched on two cards as follows:

```
( A   C   F   G )     ( B   C   D )     ( A   E   D )
( F   A   D   E )     ( B   G   F   A   E )           =
```

The output of our program will consist of a copy of the input followed by the answer in essentially the same format.

Program A (*Multiply permutations in cycle form*). This program implements Algorithm A, and it also includes provision for input, output, and the removing of singleton cycles.

```
01  CARDS    EQU   16                Unit number for card reader
02  PRINTER  EQU   18                Unit number for printer
03  ANS      ORIG  *+1000             Place for answer
04  OUTBUF   ORIG  *+24               For copies of input
05  PERM     ORIG  *+1000             The input permutation
06  BEGIN    IN    PERM(CARDS)        Read first card.
07           ENT2  0
08           LDA   EQUALS
09  1H       JBUS  *(CARDS)           Wait for cycle complete.
10           CMPA  PERM+15,2
11           JE    *+2                Is it the last card?
12           IN    PERM+16,2(CARDS)   No, read another.
13           ENT1  OUTBUF
14           JBUS  *(PRINTER)         Print input card.
15           MOVE  PERM,2(16)
16           OUT   OUTBUF(PRINTER)
17           INC2  16
18           JNE   1B                 Repeat until input complete.
```

```
19    *
20          DEC2  1              1      At this point, (rI2) words of
21          ST2   SIZE           1           input are in PERM, PERM + 1, ...
22          ENT3  0              1      A1. First pass.
23    2H    LDAN  PERM,3         A      Get next element of input.
24          CMPA  LPREN(1:5)     A      Is it "("?
25          JNE   1F             A
26          STA   PERM,3         B      Tag it.
27          INC3  1              B      Put next nonblank element
28          LDXN  PERM,3         B           in rX.
29          JXZ   *-2            B
30    1H    CMPA  RPREN(1:5)     C
31          JNE   *+2            C
32          STX   PERM,3         D      Replace ")" by tagged rX.
33          INC3  1              C
34          CMP3  SIZE           C      Have all elements been processed?
35          JL    2B             C
36    *
37          LDA   LPREN          1      Prepare for main program.
38          ENT1  ANS            1      rI1 = place to store next answer
39    OPEN  ENT3  0              E      A2. Open.
40    1H    LDXN  PERM,3         F      Look for untagged element.
41          JXN   GO             F
42          INC3  1              G
43          CMP3  SIZE           G
44          JL    1B             G
45    *                                 All are tagged. Now comes the output.
46    DONE  CMP1  =ANS=
47          JNE   *+2                   Is answer the identity permutation?
48          MOVE  LPREN(3)              If so, change to "(1)".
49          MOVE  =0=                   Put 23 words of blanks after answer.
50          MOVE  -1,1(22)
51          ENT3  0
52          OUT   ANS,3(PRINTER)
53          INC3  24
54          LDX   ANS,3                 Print as many lines as necessary.
55          JXNZ  *-3
56          HLT
57    LPREN ALF     (                   Constants used in program
58          ALF    1
59    RPREN ALF     )
60    EQUALS ALF      =
61    *
62    GO    MOVE  LPREN          H      Open a cycle in the output.
63          MOVE  PERM,3         H
64          STX   START          H
65    SUCC  STX   PERM,3         J      Tag an element.
66          INC3  1              J      Move one step to right.
```

```
67                LDXN   PERM,3(1:5)   J     A3. Set CURRENT (namely rX).
68                JXN    1F            J     Skip past blanks.
69                JMP    *-3           0
70    4H          CMPX   PERM,3(1:5)   K     A4. Scan formula.
71                JE     SUCC          K     Element = CURRENT?
72    1H          INC3   1             L     Move to right.
73                CMP3   SIZE          L     End of formula?
74                JL     4B            L
75                CMPX   START(1:5)    P     A5. CURRENT = START?
76                JE     CLOSE         P
77                STX    0,1           Q     No, output CURRENT.
78                INC1   1             Q
79                ENT3   0             Q     Scan formula again.
80                JMP    4B            Q     Go back to A4.
81    CLOSE       MOVE   RPREN         R     A6. Close.
82                CMPA   -3,1          R     Note: rA = "(".
83                JNE    OPEN          R
84                INC1   -3            S     Suppress singleton cycles.
85                JMP    OPEN          S
86                END    BEGIN
```

This program of approximately 70 instructions is quite a bit longer than the programs of the previous section, and indeed it is longer than most of the programs we will meet in this book. Its length is not formidable, however, since it divides into several small parts which are fairly independent. Lines 06–18 read in the input cards and print a copy of each card; lines 20–35 accomplish step A1 of the algorithm, the preconditioning of the input; lines 37–44 and 62–85 do the main business of Algorithm A; and lines 46–55 output the answer. The reader will find it instructive to study as many of the MIX programs given in this book as he can—it is exceedingly important to acquire skill in reading other people's computer programs, yet such training has been sadly neglected in too many computer courses and it has led to some horribly inefficient uses of computing machinery.

Timing. The parts of Program A which are not concerned with input-output have been given "timing" indications (cf. Program 1.2.10M); thus, line 27 is supposedly executed B times. For convenience it has been assumed that no blank words appear in the input except at the extreme right end; hence line 69 is never executed and the jump in line 29 never occurs.

By simple addition the total time to execute the program is

$$(7 + 5A + 6B + 7C + 2D + E + 3F + 4G + 8H + 6J + 3K + 4L + 3P + 5Q + 6R + 2S)u \qquad (7)$$

plus the time for input and output. In order to understand the meaning of formula (7), we need to examine the fifteen unknowns $A, B, C, D, E, F, G, H, J, K, L, P, Q, R, S$ and we must relate them to pertinent characteristics about

the input. We will now illustrate the general principles of attack for problems of this kind.

First we apply "Kirchhoff's law" of electrical circuit theory: the number of times an instruction is executed must equal the number of times we transfer to that instruction. This seemingly obvious rule often relates several quantities in a nonobvious way. Analyzing the flow of Program A, we get the following equations.

From lines	*We deduce*
23, 35	$A = 1 + (C - 1)$
30, 25	$C = B + (A - B)$
39, 83, 85	$E = 1 + R$
40, 44	$F = E + (G - 1)$
62, 41	$H = F - G$
65, 68, 71	$J = H + (K - (L - J))$
70, 74, 80	$K = (L - P) + Q$
81, 76	$R = P - Q$

As usual, not all of the equations given by Kirchhoff's law will be independent; in the above case, the first and second equations are obviously equivalent. Furthermore, the last two equations are equivalent, since the third, fourth, and fifth imply that $H = R$; hence the sixth says that $K = L - R$. At any rate we have already eliminated six of our fifteen unknowns:

$$\begin{aligned} A = C, \quad & E = R + 1, \quad & F = R + G, \\ H = R, \quad & K = L - R, \quad & Q = P - R. \end{aligned} \tag{8}$$

Kirchhoff's law is an effective tool which is analyzed more closely in Section 2.3.4.1; it is also called the *conservative law*.

The next step is to try to identify the variables with important characteristics of the data. We find from lines 21, 22, 27, and 33 that

$$B + C = \text{number of words of input} = 16X - 1, \tag{9}$$

where X is the number of input cards. From line 25,

$$B = \text{number of "(" in input} = \text{number of cycles in input.} \tag{10}$$

Similarly, from line 31,

$$D = \text{number of ")" in input} = \text{number of cycles in input.} \tag{11}$$

Now (10) and (11) give us a fact that could not be deduced by Kirchhoff's law:

$$B = D. \tag{12}$$

From line 62,

$$H = \text{number of cycles in output (including singletons).} \tag{13}$$

Line 81 says R is equal to this same quantity; the fact that $H = R$ *was* in this case deducible from Kirchhoff's law, since it already appears in (8).

Using the fact that each nonblank word is ultimately tagged, and lines 26, 32, and 65, we find that

$$J = Y - 2B, \tag{14}$$

where Y is the number of nonblank words appearing in the input permutations. From the fact that every *distinct* element appearing in the input permutation is written into the output just once, either at line 63 or line 77, we have (see Eqs. 8)

$$P = H + Q = \text{number of distinct elements in input.} \tag{15}$$

A moment's reflection makes this clear from line 75 as well. Finally, we see from line 84 that

$$S = \text{number of singleton cycles in output.} \tag{16}$$

Clearly the quantities B, C, H, J, P, and S that we have now interpreted are essentially independent parameters which may be expected to enter into the timing of Program A.

The results we have obtained so far leave us with only the unknowns G and L to be analyzed. For these we must use a little more ingenuity. The scans of the input which start at lines 39 and 79 always terminate either at line 45 (the last time) or at line 75. During each one of these $P + 1$ loops, the instruction "`INC3 1`" is performed $B + C$ times; this takes place only at lines 42, 66, and 72, so we get the nontrivial relation

$$G + J + L = (B + C)(P + 1) \tag{17}$$

connecting our unknowns G and L. Fortunately, the timing formula is a function of $G + L$ (it involves $\cdots + 3F + 4G \cdots + 3K + 4L + \cdots = \cdots + 7G + \cdots + 7L + \cdots$) so we need not try to analyze the individual quantities G and L any further.

Summing up all the above results, we find that the total time, excluding input-output, comes to

$$(112NX + 304X + N - 2M - Y + 10U + 2V - 11)u; \tag{18}$$

in this formula, new names for the data characteristics have been used as follows:

$$\begin{aligned}
X &= \text{number of cards of input,}\\
Y &= \text{number of nonblank fields in input (excluding final "="),}\\
M &= \text{number of cycles in input,}\\
N &= \text{number of distinct element names in input,}\\
U &= \text{number of cycles in output (including singletons),}\\
V &= \text{number of singleton cycles in output.}
\end{aligned} \tag{19}$$

In this way we have found that analysis of a program like Program A is in many respects like solving an amusing puzzle.

We will show below that, if the output permutation is assumed to be random, the quantities U and V will be H_N and 1, respectively, on the average.

Another approach. Algorithm A multiplies permutations together much as people ordinarily do the same job. Quite often we find that problems to be solved by computer are very similar to problems that have confronted humans for many years; therefore time-honored methods of solution which have evolved for use by mortals such as we are also appropriate procedures for computer algorithms.

Just as often, however, we find that some methods which are quite unsuitable for human use are really superior for computers. The central reason is that the computer "thinks" differently; it has a different kind of memory for facts. An instance of this difference may be seen in our permutation-multiplication problem—using the algorithm below, a computer can do the multiplication in one sweep over the formula, simultaneously remembering the current state of the permutations being multiplied. While Algorithm A scans once through the formula for each element of the output, the new algorithm does all in one scan; this is a feat which could not be reliably done by man.

Let us now look into this computer-oriented method for multiplying permutations. It is convenient to go from right to left; consider the following table:

	(	a	c	f	g	)	(	b	c	d	)	(	a	e	d	)	(	f	a	d	e	)	(	b	g	f	a	e	)
$a \rightarrow$	d	d	a	a	a	a	a	a	a	a	a	a	a	a	d	d	d	d	d	d	e	e	e	e	e	e	e	e	a	a
$b \rightarrow$	c	c	c	c	c	c	c	c	g	g	g	g	g	g	g	g	g	g	g	g	g	g	g	g	b	b	b	b	b	
$c \rightarrow$	e	e	e	d	d	d	d	d	d	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	
$d \rightarrow$	g	g	g	g	g	g	g	)	)	)	d	d	)	)	)	b	b	b	b	b	d	d	d	d	d	d	d	d	d	
$e \rightarrow$	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	a	a	a	)	)	)	)	b	b	)	)	)	)	)	e
$f \rightarrow$	f	f	f	f	e	e	e	e	e	e	e	e	e	e	e	e	e	e	a	a	a	a	a	a	a	a	f	f	f	
$g \rightarrow$	a	)	)	)	)	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	g	g	g	g

The column below each character of the cycle form represents what permutation is represented by the partial cycles *to the right;* for example, the fragmentary formula "... d e)(b g f a e)" represents the permutation

$$\begin{pmatrix} a & b & c & d & e & f & g \\ e & g & c & b & ? & a & f \end{pmatrix},$$

which appears under the rightmost d of the table.

Inspection of this table shows how it was constructed, going from right to left. The column below letter x differs from that on its right only in row x; the new value in that row and column is the one which disappeared in the preceding change. More precisely, we have the following algorithm:

Algorithm B (*Multiply permutations in cycle form*). This algorithm accomplishes essentially the same result as Algorithm A. Assume that the elements permuted are named $x_1, x_2, \ldots, x_n$. We use an auxiliary table $T[1], T[2], \ldots, T[n]$; upon termination of this algorithm, x_i goes to x_j under the input permutation if and only if $T[i] = j$.

B1. [Initialize.] Set $T[k] \leftarrow k$ for $1 \le k \le n$. Also, prepare to scan the input from right to left.

B2. [Next element.] Examine the next element of the input (right to left). If the input has been exhausted, the algorithm terminates. If the element is a ")", set $Z \leftarrow 0$ and repeat step B2; if it is a "(", go to B4; otherwise the element is x_i for some i, go on to B3.

B3. [Change $T[i]$.] Exchange $Z \leftrightarrow T[i]$. If this makes $T[i] = 0$, set $j \leftarrow i$. Return to step B2.

B4. [Change $T[j]$.] Set $T[j] \leftarrow Z$. (At this point, j is the row which shows a ")" entry in (21), corresponding to the right parenthesis which matches this left parenthesis.) Return to step B2. ∎

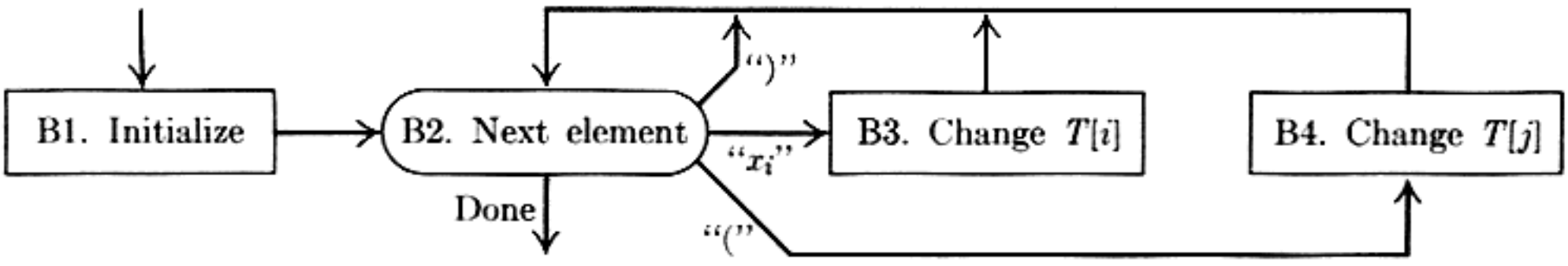


Fig. 21. Algorithm B for multiplying permutations.

Of course, after this algorithm has been performed, we still must output the contents of Table T in cycle form; this is easily done by a "tagging" method, as we shall see below.

Let us now write a MIX program based on the new algorithm. We wish to use the same ground rules as those in Program A, i.e., the form of the input and output should be essentially the same. A slight problem presents itself; namely, how can we implement Algorithm B without knowing in advance what the elements $x_1, x_2, \ldots, x_n$ are? We don't know n, and we don't know whether the element named b is to be x_1, or x_2, etc. A simple way to solve this problem is to keep a table of the element names encountered so far, and to search for the current name each time (see lines 31–36 in the program below).

Program B (*Same effect as Program A*). rX ≡ Z; rI4 ≡ i; rI1 ≡ j; rI3 ≡ 2 × (size of names table) + 1. The table of names consists of two-word entries:

Word 1	+	Name of x_i				(in character code)
Word 2	+	0	0	0	'$T[i]$'	(address of entry for x_j, if x_i goes to x_j).

```
01    NAMES   ORIG  *+1000          Table of names
02    CARDS   EQU   16       }
 . . .                       } Same as lines 01–19 of Program A
20    *                     }      At this point, (rI2) words of
21            DEC2  1          1       input are in PERM, PERM+1 ...
22            ENT3  1          1    Size of NAMES table
23    RIGHT   ENTX  0          A    Set Z ← 0.
24    SCAN    DEC2  1          B    B2. Next element.
25            LDA   PERM,2     B
26            JAZ   CYCLE      B    Skip over blanks.
27            CMPA  RPREN      C
28            JE    RIGHT      C    Is the next element ")"?
29            CMPA  LPREN      D
30            JE    LEFT       D    Is it "("?
31            ENT4  -2,3       E
32            J4N   1F         E    Is the names table empty?
33    2H      CMPA  NAMES,4     F    Search through names table.
34            JE    FOUND      F    Has the name appeared before?
35            DEC4  2          G
36            J4NN  2B         G
37    1H      STA   NAMES,3     H    Put new entry into table.
38            ST3   NAMES+1,3   H    Set T[i] ← i.
39            ENT4  0,3        H
40            INC3  2          H    Increase size of table.
41    FOUND   LDA   NAMES+1,4   J    B3. Change T[i].
42            STX   NAMES+1,4   J    Store Z.
43            SRC   5          J    Set Z.
44            JANZ  SCAN       J
45            ENT1  0,4        K    If Z was zero, set j ← i.
46            JMP   SCAN       K
47    LEFT    STX   NAMES+1,1   L    B4. Change T[j].
48    CYCLE   J2P   SCAN       P    Return to B2, unless finished.
49    *
50    OUTPUT  ENT1  ANS        1    All input has been scanned.
51            DEC3  2          1    Names table contains the answer.
52    1H      LDAN  NAMES,3     Q    Now we construct cycle notation.
53            JAP   SKIP       Q    Has name been tagged?
```

```
54              CMP3    NAMES+1,3       R       Is there a singleton cycle?
55              JE      SKIP            R
56              ENT2    0,3             S
57              MOVE    LPREN           S       Open a cycle.
58      2H      MOVE    NAMES,2         T
59              STA     NAMES,2         T       Tag the name.
60              LD2     NAMES+1,2       T       Find successor of element.
61              LDAN    NAMES,2         T
62              JAN     2B              T       Is it already tagged?
63              MOVE    RPREN           W       Yes, cycle closes.
64      SKIP    DEC3    2               Z       Move to next name.
65              J3P     1B              Z
66      *
67      DONE    CMP1    =ANS=   }
 . . .                          } Same as lines 46–60 of Program A
81      EQUALS  ALF       =     }
82              END     BEGIN   ▮
```

Lines 50–65, which construct the cycle notation from the T table (i.e., the names table), make a rather pretty little algorithm which merits some study. The quantities $A, B, \ldots, R, S, T, W. Z$ which enter into the timing of this program are, of course, different from the quantities of the same name in the analysis of Program A. The reader will find it an interesting exercise to analyze these times (see exercise 10).

Experience shows that the main portion of the execution time of Program B will be spent in searching the NAMES table—this is quantity G in the timing. Actually much better algorithms for searching and building such a NAMES table are available; these are called *symbol table algorithms*, and they are of great importance in computer applications. Section 6.5 contains a thorough discussion of efficient symbol table algorithms.

Inverses. The inverse π^{-1} of a permutation π is the rearrangement which undoes the effect of π; if i goes to j under π, then j goes to i under π^{-1}. Thus the product $\pi\pi^{-1}$ equals the identity permutation.

Every permutation has an inverse; for example, the inverse of

$$\begin{pmatrix} a & b & c & d & e & f \\ c & d & f & b & e & a \end{pmatrix} \quad \text{is} \quad \begin{pmatrix} c & d & f & b & e & a \\ a & b & c & d & e & f \end{pmatrix} = \begin{pmatrix} a & b & c & d & e & f \\ f & d & a & b & e & c \end{pmatrix}.$$

We will now consider some simple algorithms for computing the inverse of a permutation.

For the rest of this section, let us assume we are dealing with permutations of the numbers $\{1, 2, \ldots, n\}$. If $X[1]\,X[2]\cdots X[n]$ is such a permutation, there is a simple method to compute its inverse: set $Y[X[k]] \leftarrow k$ for $1 \le k \le n$. Then $Y[1]\,Y[2]\cdots Y[n]$ is the desired inverse. This method uses $2n$ memory cells, n for X and n for Y.

Just for fun, suppose that n is very large and suppose also that we wish to compute the inverse of $X[1]\,X[2]\cdots X[n]$ without using much additional memory space; we want to compute the inverse "in place" so that after our algorithm is finished $X[1]\,X[2]\cdots X[n]$ is the inverse of the original permutation. Merely setting $X[X[k]] \leftarrow k$ for $1 \le k \le n$ will certainly fail, but by considering the cycle structure we can derive the following simple algorithm:

Algorithm I (*Inverse in place*). Replace $X[1]X[2]\cdots X[n]$, a permutation on $\{1, 2, \ldots, n\}$, by its inverse. *Reference: CACM* **8** (1965), 670.

I1. [Initialize.] Set $m \leftarrow n$.

I2. [Next element.] Set $i \leftarrow X[m]$. If $i < 0$, set $X[m] \leftarrow -i$ and go to I6 (this element has already been processed). If $i = m$, go to I6 (this element is fixed by the permutation).

I3. [Open.] Set $k \leftarrow m$.

I4. [Invert one.] Set $j \leftarrow X[i]$, $X[i] \leftarrow -k$.

I5. [End cycle.] If $j = m$, then set $X[m] \leftarrow i$; otherwise set $k \leftarrow i$, $i \leftarrow j$, and return to I4.

I6. [More?] Decrease m by 1; if $m > 0$, go to I2. Otherwise the algorithm terminates. ∎

For an example of this algorithm, see Table 2. The method is based on inversion of successive cycles of the permutation.

Table 2

COMPUTING THE INVERSE OF 6 2 1 5 4 3 BY ALGORITHM I.
(Read columns from left to right.) At point *, the cycle (163) has been inverted.

After step:	I1	I3	I4	I4	I6*	I3	I4	I6	I6	I6	I6	I6
$X[1]$	6	6	6	−3	−3	−3	−3	−3	−3	−3	−3	3
$X[2]$	2	2	2	2	2	2	2	2	2	2	2	2
$X[3]$	1	1	−6	−6	−6	−6	−6	−6	−6	6	6	6
$X[4]$	5	5	5	5	5	5	−5	−5	5	5	5	5
$X[5]$	4	4	4	4	4	4	4	4	4	4	4	4
$X[6]$	3	3	3	3	1	1	1	1	1	1	1	1
m	6	6	6	6	5	5	5	4	3	2	1	0
i		3	3	1	1	4	4	4	−5	−6	2	−3
j			1	6	6	6	5	5	5	5	5	5
k		6	6	3	3	5	5	5	5	5	5	5

Algorithm I resembles parts of Algorithm A, and it very strongly resembles the cycle-finding algorithm in Program B (lines 50–65). Thus it is typical of a number of algorithms involving rearrangements. A MIX program to implement it is quite simple; see Program I on the next page.

Program I. $m \equiv \text{rI1}$; $i \equiv \text{rI2}$; $(-k) \equiv \text{rI3}$; $j \equiv \text{rI4}$; $n \equiv$ N, a symbol to be defined when this program is assembled as part of a larger program.

```
01    INVERT  ENT1  N          1          I1. Initialize. m ← n.
02    2H      LD2   X,1        N          I2. Next element. i ← X[m].
03            J2NN  *+3        N
04            STZ   X,1(0:0)   N − C      Set X[m] positive.
05            JMP   6F         N − C
06            CMP1  X,1        C          i = m?
07            JE    6F         C
08            ENN3  0,1        C − S      I3. Open. k ← m.
09            JMP   4F         C − S
10    3H      INC4  0,1        N − 2C + S
11            ENN3  0,2        N − 2C + S k ← i.
12            ENT2  0,4        N − 2C + S i ← j.
13    4H      LD4   X,2        N − C      I4. Invert one. j ← X[i].
14            ST3   X,2        N − C          X[i] ← −k.
15            DEC4  0,1        N − C      I5. End cycle.
16            J4NZ  3B         N − C          j = m?
17            ST2   X,1        C − S          Yes, set X[m] ← i.
18    6H      DEC1  1          N          I6. More?
19            J1P   2B         N
20            HLT                         Inverse found. ∎
```

The timing for this program is easily worked out in the manner shown earlier: it is $(17N - 8C - S + 1)u$, where N is the order of the permutation, C is the total number of cycles, and S is the number of fixed elements (singleton cycles). The quantities C, S in a random permutation are analyzed below.

There is almost always more than one algorithm to do any given task, so we would expect there may be another way to invert a permutation. The following ingenious algorithm is due to J. Boothroyd:

Algorithm J (*Inverse in place*). This algorithm has the same effect as Algorithm I but uses a different method.

J1. [Negate all.] Set $X[k] \leftarrow -X[k]$, for $1 \le k \le n$. Also set $m \leftarrow n$.

J2. [Initialize j.] Set $j \leftarrow m$.

J3. [Find negative entry.] Set $i \leftarrow X[j]$. If $i > 0$, set $j \leftarrow i$ and repeat this step.

J4. [Invert.] Set $X[j] \leftarrow X[-i]$, $X[-i] \leftarrow m$.

J5. [Loop on m.] Decrease m by 1; if $m > 0$, go back to J2. Otherwise the algorithm terminates. ∎

See Table 3 for an example of this algorithm. Again the method is essentially based on the cycle structure, but this time it is less obvious that the algorithm really works! Verification is left to the reader (see exercise 13).

Table 3

COMPUTING THE INVERSE OF 6 2 1 5 4 3 BY ALGORITHM J.

After step:	J2	J3	J5	J3	J5	J3	J5	J3	J5	J3	J5	J3	J5
$X[1]$	−6	−6	−6	−6	−6	−6	−6	−6	3	3	3	3	3
$X[2]$	−2	−2	−2	−2	−2	−2	−2	−2	−2	−2	2	2	2
$X[3]$	−1	−1	6	6	6	6	6	6	6	6	6	6	6
$X[4]$	−5	−5	−5	−5	5	5	5	5	5	5	5	5	5
$X[5]$	−4	−4	−4	−4	−5	−5	4	4	4	4	4	4	4
$X[6]$	−3	−3	−1	−1	−1	−1	−1	−1	−6	−6	−6	−6	1
m	6	6	5	5	4	4	3	3	2	2	1	1	0
i		−3	−3	−4	−4	−5	−5	−1	−1	−2	−2	−6	−6
j	6	6	6	5	5	5	5	6	6	2	2	6	6

Program J (*Analogous to Program I*). $m \equiv$ rI1; $j \equiv$ rI2; $(-i) \equiv$ rI3.

```
01    INVERT  ENN1  N                 1    J1. Negate all.
02            ST1   X+N+1,1(0:0)      N        Set sign negative.
03            INC1  1                 N
04            J1N   *-2               N        More?
05            ENT1  N                 1        m ← n.
06    2H      ENN3  0,1               N    J2. Initialize j.
07            ENN2  0,3               A
08            LD3N  X,2               A    J3. Find negative entry.
09            J3N   *-2               A        i > 0?
10            LDA   X,3               N    J4  Invert.
11            STA   X,2               N        X[j] ← X[−i].
12            ST1   X,3               N        X[−i] ← m.
13            DEC1  1                 N    J5. Loop on m.
14            J1P   2B                N
15            HLT                     1    Inverse found. ∎
```

This program is a little shorter than the preceding one. To decide how fast it runs, we need to know the quantity A; this quantity is so interesting and instructive, it has been left as an exercise (see exercise 14).

In spite of the elegance of Algorithm J, we must reluctantly report that the results of an analysis of these two algorithms show that Algorithm I is definitely superior. In fact, it turns out that the running time for Algorithm I is essentially proportional to n, while that of Algorithm J is essentially proportional to $n \ln n$. As n approaches infinity (and the algorithms were intended for large n), the ratio of execution times goes to zero. It is perhaps a shame that the more subtle algorithm loses out in this case; but the analysis of algorithms is expressly intended to tell us the true facts, however greatly they might run contrary to personal taste. Maybe some day someone will find a use for Algorithm J (or some related modification); it is a bit too pretty to be forgotten altogether.

An unusual correspondence. We have already remarked that the cycle notation for a permutation is not unique; the permutation on six elements (1 6 3)(4 5) may be written (5 4)(3 1 6), etc. It will be useful to consider a *canonical form* for the cyclic notation; the canonical form *is* unique. To get the canonical form, proceed as follows:

a) Write all singleton cycles explicitly.
b) Within each cycle, put the smallest number first.
c) Order the cycles in *decreasing* order of the first number in the cycle.

For example, starting with (3 1 6)(5 4) we would get

$$\text{(a): } (3\ 1\ 6)(5\ 4)(2); \quad \text{(b): } (1\ 6\ 3)(4\ 5)(2); \quad \text{(c): } (4\ 5)(2)(1\ 6\ 3). \tag{20}$$

The important property of this canonical form is that the parentheses may be dropped and uniquely reconstructed again. Thus there is only one way to insert parentheses in "4 5 2 1 6 3" to get a canonical cycle form; one must insert a left parenthesis just before each *left-to-right minimum* (i.e., just before each element which is preceded by no smaller elements).

This insertion and removal of parentheses gives us an unusual one-to-one correspondence between the set of all permutations expressed in cycle form and the set of all permutations expressed in linear form. [*Example:* the permutation 6 2 1 5 4 3 in cycle form is (4 5)(2)(163); remove parentheses to get 4 5 2 1 6 3 which in cycle form is (2 5 6 3)(1 4); remove parentheses to get 2 5 6 3 1 4 which in cycle form is (3 6 4)(1 2 5); etc.]

This correspondence has numerous applications to the study of permutations of different types. For example, let us ask "How many cycles does a permutation on n elements have, on the average?" To answer this question we consider the set of all $n!$ permutations expressed in canonical form, and drop the parentheses; we are left with the set of all $n!$ permutations in some order. Our original question is therefore equivalent to, "How many left-to-right minima does a permutation on n elements have, on the average?" We have already answered this question in Section 1.2.10 (actually, we discussed the average number of right-to-left maxima, which is essentially the same by symmetry); this was the quantity $(A + 1)$ in the analysis of algorithm 1.2.10M, for which we found the statistics

$$\min 1, \qquad \text{ave } H_n, \qquad \max n, \qquad \text{dev } \sqrt{(H_n - H_n^{(2)})}. \tag{21}$$

Furthermore, we found that *a permutation of n objects has k cycles* (i.e., k left-to-right minima) *with probability* $\left[{n \atop k}\right]/n!$.

We can also ask about the average distance *between* left-to-right minima, which becomes equivalent to the average length of a cycle. By (21), the *total* number of cycles among all the $n!$ permutations is $n!H_n$ (since it is $n!$ times the *average* number of cycles). If we pick a cycle at random, what is its average length?

Imagine all $n!$ permutations of $\{1, 2, \ldots, n\}$ written down in cycle notation; how many three-cycles are present? To answer this question, let us consider how many times a particular three-cycle $(x\ y\ z)$ appears: clearly, the cycle $(x\ y\ z)$ appears in exactly $(n-3)!$ of the permutations, since this is the number of ways the remaining $n-3$ elements may be permuted. Now the number of different possible three-cycles $(x\ y\ z)$ is $n(n-1)(n-2)/3$, since there are n choices for x, $(n-1)$ for y, $(n-2)$ for z, and among these $n(n-1)(n-2)$ choices each different three-cycle has appeared in three forms $(x\ y\ z)$, $(y\ z\ x)$, $(z\ x\ y)$. Therefore the total number of three-cycles among all $n!$ permutations is $n(n-1)(n-2)/3$ times $(n-3)!$, namely $n!/3$. Similarly, the total number of m-cycles is $n!/m$, $1 \le m \le n$. (This provides another simple proof of the fact that the total number of cycles is $n!H_n$; hence the average number of cycles in a permutation is H_n, as we already know.) If we consider the $n!H_n$ *cycles* equally probable, the average length of a randomly chosen cycle is n/H_n; if an *element* is chosen at random in a random permutation, the average length of the cycle containing it is somewhat longer than this (see exercise 17).

To complete our analyses of Algorithms A, B, and I, we would like to know the average number of *singleton cycles* in a random permutation. This is an interesting problem. Suppose we write down the $n!$ permutations, listing first those with no singleton cycles, then those with just one, etc.; for example, if $n = 4$,

no fixed elements:	2143	2341	2413	3142	3412	3421	4123	4312	4321
one fixed element:	$\underline{1}342$	$\underline{1}423$	$3\underline{2}41$	$4\underline{2}13$	$24\underline{3}1$	$41\underline{3}2$	$231\underline{4}$	$312\underline{4}$	
two fixed elements:	$\underline{12}43$	$\underline{1}4\underline{3}2$	$\underline{1}32\underline{4}$	$4\underline{23}1$	$3\underline{2}1\underline{4}$	$21\underline{34}$			
three fixed elements:									
four fixed elements:	$\underline{1234}$								

(Singleton cycles, i.e. fixed elements, have been specially designated in this list.)

Let P_{nk} be the number of permutations of order n having exactly k fixed elements, so that for example,

$$P_{40} = 9, \qquad P_{41} = 8, \qquad P_{42} = 6, \qquad P_{43} = 0, \qquad P_{44} = 1.$$

Study of the list above shows us the principal relationship between these numbers: we can get all permutations with k fixed elements by first choosing the k that are to be fixed [this can be done in $\binom{n}{k}$ ways] and then permuting the remaining $n-k$ elements in all $P_{(n-k)0}$ ways that leave no further elements fixed. Hence

$$P_{nk} = \binom{n}{k} P_{(n-k)0}. \tag{22}$$

We also have the rule that "the whole is the sum of its parts":

$$n! = P_{nn} + P_{n(n-1)} + P_{n(n-2)} + P_{n(n-3)} + \cdots. \tag{23}$$

Combining Eqs. (22) and (23) and rewriting the result slightly, we find that

$$n! = P_{00} + \frac{1}{1!} nP_{10} + \frac{1}{2!} n(n-1)P_{20} + \frac{1}{3!} n(n-1)(n-2)P_{30} + \cdots, \quad (24)$$

an equation that must be true for all positive integers n. This equation already has confronted us before—it appears in Section 1.2.5 in connection with Stirling's attempt to generalize the factorial function—and a simple derivation of the coefficients was given in Section 1.2.6 (Eq. 32 and following). We conclude that

$$\frac{1}{m!} P_{m0} = 1 - \frac{1}{1!} + \frac{1}{2!} - \cdots + (-1)^m \frac{1}{m!}. \quad (25)$$

Now let p_{nk} be the probability that a permutation of order n has exactly k singleton cycles; since $p_{nk} = P_{nk}/n!$, we have from Eqs. (22) and (25)

$$p_{nk} = \frac{1}{k!}\left(1 - \frac{1}{1!} + \frac{1}{2!} - \cdots + (-1)^{n-k} \frac{1}{(n-k)!}\right). \quad (26)$$

The generating function $G_n(z) = p_{n0} + p_{n1}z + p_{n2}z^2 + \cdots$ is therefore

$$G_n(z) = 1 + \frac{1}{1!}(z-1) + \cdots + \frac{1}{n!}(z-1)^n = \sum_{0 \le j \le n} \frac{1}{j!}(z-1)^j. \quad (27)$$

From this formula it follows that $G_n'(z) = G_{n-1}(z)$, and from the methods of Section 1.2.10 we obtain the following statistics on the number of singleton cycles:

$$(\min 0,\ \text{ave } 1,\ \max n,\ \text{dev } 1), \qquad \text{if} \qquad n \ge 2. \quad (28)$$

A somewhat more direct way to count the number of permutations having no singleton cycles follows from the "principle of inclusion and exclusion," which is an important method for many enumeration problems. The general principle of inclusion and exclusion may be formulated as follows: We are given N elements, and M subsets, $S_1, S_2, \ldots, S_M$, of these elements; and our goal is to count how many of the elements lie in none of these subsets. Let $||S||$ denote the number of elements in a set S; then the desired number of objects in none of the sets S_j is

$$N - \sum_{1 \le j \le M} ||S_j|| + \sum_{1 \le j < k \le M} ||S_j \cap S_k|| - \sum_{1 \le i < j < k \le M} ||S_i \cap S_j \cap S_k|| + \cdots + (-1)^M ||S_1 \cap \cdots \cap S_M||. \quad (29)$$

(Thus we first subtract the number of elements in $S_1, \ldots, S_M$ from the total number, N, but this underestimates the desired total; so we add back the number of elements which are common to pairs of sets, $S_j \cap S_k$, for each pair S_j and S_k, then subtract the elements common to triples of sets, etc.) There are several ways to prove this formula, and the reader is invited to discover one of these for himself.

To count the number of permutations on n elements having no singleton cycles, we consider the $N = n!$ permutations and let S_j be the set of permutations in which element j forms a singleton cycle. If $1 \le j_1 < j_2 < \cdots < j_k \le n$, the number of elements in $S_{j_1} \cap S_{j_2} \cap \cdots \cap S_{j_k}$ is the number of permutations in which $j_1, \ldots, j_k$ are singleton cycles, and this is clearly $(n - k)!$. Thus formula (29) becomes

$$n! - \binom{n}{1}(n-1)! + \binom{n}{2}(n-2)! - \binom{n}{3}(n-3)! + \cdots + (-1)^n\binom{n}{n}0!$$

and this agrees with (25).

EXERCISES

1. [*10*] Show that the transformation of the numbers $\{0, 1, 2, 3, 4, 5, 6\}$, defined by the rule that x goes to $(2x)$ mod 7, is a permutation, and write it in cycle form.

2. [*10*] The text shows how we might set $(a, b, c, d, e, f) \leftarrow (c, d, f, b, e, a)$ by using a series of replacement operations and one auxiliary variable t. Show how to do this by using a series of *exchange* operations (i.e., $x \leftrightarrow y$) and no auxiliary variables.

3. [*10*] Compute the product

$$\begin{pmatrix} a & b & c & d & e & f \\ b & d & c & a & f & e \end{pmatrix} \times \begin{pmatrix} a & b & c & d & e & f \\ c & d & f & b & e & a \end{pmatrix},$$

and express the answer in two-line notation (cf. Eq. 4).

4. [*10*] Express $(a\ b\ d)(e\ f)(a\ c\ f)(b\ d)$ in terms of disjoint cycles.

▶ **5.** [*M10*] Equation (3) shows several equivalent ways to express the same permutation in cycle form. How many different ways of writing that permutation are possible, if all singleton cycles are suppressed?

6. [*M23*] What changes are made to the timing of Program A if we remove the assumption that all blank words occur at the extreme right?

7. [*10*] If Program A is presented with the input (6), what are the quantities X, Y, M, N, U, and V of (19)? What is the time required by Program A, exclusive of input-output?

▶ **8.** [*23*] Would it be feasible to modify Algorithm B to go from left to right instead of from right to left through the input?

9. [*10*] Both Programs A and B accept the same input and give the answer in essentially the same form. Is the output *exactly* the same under both programs?

▶ **10.** [*M28*] Examine the timing characteristics of Program B, viz. the quantities $A, B, \ldots, Z$ shown there; express the total time in terms of X, Y, M, N, U, V [cf. (19)] and of the quantity G. Compare the total time for Program B with the total time for Program A when given the input (6), using the fact that $G = 55$ in this case (cf. exercise 7).

11. [*15*] Find a simple rule for writing π^{-1} in cycle form, if the permutation π is given in cycle form.

12. [*M27*] (Transposing a rectangular matrix.) Suppose an $m \times n$ matrix (a_{ij}), $m \neq n$, is stored in memory in a fashion like that of exercise 1.3.2–10, so that the value of a_{ij} appears in location $L + n(i-1) + (j-1)$, where L is the location of a_{11}. The problem is to find a way to *transpose* this matrix, obtaining an $n \times m$ matrix (b_{ij}), where $b_{ij} = a_{ji}$ and b_{ij} is stored in location $L + m(i-1) + (j-1)$. Thus the matrix is to be transposed "on itself." (a) Show that this transposition transformation moves the value which appears in cell $L + x$ to cell $L + (mx) \bmod N$, where $0 \leq x < N = mn - 1$. (b) Discuss methods for doing this transposition by computer.

▶ **13.** [*M24*] Prove that Algorithm J is valid.

▶ **14.** [*M34*] Find the average value of the quantity A which enters into the timing of Algorithm J.

15. [*M12*] Is there a permutation which represents exactly the same transformation both in the canonical cycle form without parentheses and in the linear form?

16. [*M15*] Start with the permutation 1324 in linear notation; convert it to canonical cycle form and then remove the parentheses; repeat this process until arriving at the original permutation. What permutations occur during this process?

17. [*M24*] (a) The text demonstrates that there are $n!H_n$ cycles in all among the permutations on n elements. If these cycles (including singleton cycles) are individually written on $n!H_n$ slips of paper, and if one of these slips of paper is chosen at random, what is the average length of the cycle that is thereby picked? (b) If we write the $n!$ permutations on $n!$ slips of paper, and if we choose a number k at random and also choose one of these slips of paper, what is the probability that the cycle containing the element k is an m-cycle? What is the average length of the cycle containing k?

▶ **18.** [*M27*] What is p_{nkm}, the probability that a permutation of n objects has exactly k m-cycles? What is the corresponding generating function $G_{nm}(z)$? What is the average number of m-cycles and what is the standard deviation? (The text considers only the case $m = 1$.)

19. [*HM21*] Show that, in the notation of Eq. (25),

$$P_{n0} = \lfloor n!/e + \tfrac{1}{2} \rfloor, \qquad n \geq 1.$$

20. [*M20*] Given that all singleton cycles are written out explicitly, how many different ways are there to write the cycle notation of a permutation which has α_1 one-cycles, α_2 two-cycles, . . . ? (Cf. exercise 5.)

21. [*M22*] What is the probability $P(n; \alpha_1, \alpha_2, \ldots)$ that a permutation of order n has exactly α_1 one-cycles, α_2 two-cycles, etc.?

▶ **22.** [*HM34*] (The following approach, due to L. Shepp and S. P. Lloyd, gives a convenient and powerful method for solving problems related to the cycle structure of random permutations.) Instead of regarding the number, n, of objects as fixed, and the permutation variable, let us instead suppose that we independently choose the quantities $\alpha_1, \alpha_2, \alpha_3, \ldots$ appearing in exercises 20 and 21 according to some probability distribution. Let w be any real number between 0 and 1. (a) Suppose that we choose the random variables $\alpha_1, \alpha_2, \alpha_3, \ldots$ according to the rule that "the probability

$\alpha_m = k$ is $f(w, m, k)$," for some function $f(w, m, k)$. Determine the value of $f(w, m, k)$ so that the following two conditions hold:

i) $\sum_{k\geq 0} f(w, m, k) = 1$, for $0 < w < 1$ and $m \geq 1$.

ii) The probability that $\alpha_1 + 2\alpha_2 + 3\alpha_3 + \cdots = n$ *and* that $\alpha_1 = k_1$, $\alpha_2 = k_2$, $\alpha_3 = k_3, \ldots,$ is $(1 - w)w^n P(n; k_1, k_2, k_3, \ldots)$ as in exercise 21.

b) A permutation whose cycle structure is $\alpha_1, \alpha_2, \alpha_3, \ldots$ clearly permutes exactly $\alpha_1 + 2\alpha_2 + 3\alpha_3 + \cdots$ objects. Show that if the α's are randomly chosen according to the probability distribution in part (a), the probability that $\alpha_1 + 2\alpha_2 + 3\alpha_3 + \cdots = n$ is $(1 - w)w^n$; the probability that $\alpha_1 + 2\alpha_2 + 3\alpha_3 + \cdots$ is *infinite* is zero.

c) Let $\phi(\alpha_1, \alpha_2, \ldots)$ be any function of the infinitely many numbers $\alpha_1, \alpha_2, \ldots$; show that if the α's are chosen according to the probability distribution in (a), the average value of ϕ is $(1 - w) \sum_{n\geq 0} w^n \phi_n$; here ϕ_n denotes the average value of ϕ taken over all permutations of n objects, where $\alpha_1, \alpha_2, \ldots$ represent the number of cycles of the permutation. [For example, if $\phi(\alpha_1, \alpha_2, \ldots) = \alpha_1$, the text showed that $\phi_n = 1$, the average number of singleton cycles, regardless of n.]

d) Use this method to find the average number of cycles of *even* length in a random permutation of n objects.

e) Use this method to solve exercise 17. (Cf. exercise 1.2.10–15.)

23. [*HM45*] (Golomb, Shepp, Lloyd.) If l_n denotes the average length of the *longest* cycle in a permutation of n objects, show that $l_n \approx \lambda n + \frac{1}{2}\lambda$, where $\lambda \approx 0.62433$ is a constant. Show in fact that $\lim_{n\to\infty} (l_n - \lambda n - \frac{1}{2}\lambda) = 0$.

24. [*M41*] Find the generating function for the quantity A which enters into the timing of Algorithm J. (Cf. exercise 14.)

25. [*M22*] Prove Eq. (29).

▶ **26.** [*M24*] Extend the principle of inclusion and exclusion to obtain a formula for the number of elements which are in exactly r of the subsets $S_1, S_2, \ldots, S_M$. (The text considers only the case $r = 0$.)

27. [*M20*] Use the principle of inclusion and exclusion to count the number of integers n in the range $0 \leq n < am_1m_2 \cdots m_t$, where $a, m_1, m_2, \ldots, m_t$ are positive integers, with $\gcd(m_j, m_k) = 1$ when $j \neq k$.

1.4. SOME FUNDAMENTAL PROGRAMMING TECHNIQUES

1.4.1. Subroutines

When a certain task is to be performed at several different places in a program, it is usually undesirable to repeat the coding in each place. To avoid this situation, the coding (called a "subroutine") can be put into one place only, and a few extra instructions can be added to restart the outer program properly after the subroutine is finished. Transfer of control between subroutines and main programs is called "subroutine linkage."

Each machine has its own peculiar manner for achieving efficient subroutine linkage, usually involving special instructions. In MIX, the J-register is used for this purpose; our discussion will be based on MIX machine language, but similar remarks will apply to subroutine linkage on other computers.

Subroutines are used to save space in a program; they do not save any time, other than the time implicitly saved by having less space (e.g., less time to load the program, or fewer passes necessary in the program, or better use of high-speed memory on machines with several grades of memory). The extra time taken to enter and leave a subroutine is usually negligible.

Subroutines have several other advantages. They make it easier to visualize the structure of a large and complex program; they form a logical segmentation of the entire problem, and this usually makes debugging of the program easier. Many subroutines have additional value because they can be used by people other than the programmer of the subroutine.

Most computer installations have built up a large "library" of useful subroutines, and such a library greatly facilitates the programming of standard computer applications which arise. A programmer should not think of this as the *only* purpose of subroutines, however; subroutines should not always be regarded as "general purpose" programs to be used by the community. Even the use of very special purpose subroutines, which are intended to appear in only one program, is an important technique.

The simplest subroutines are those which have only one entrance and one exit, such as the MAXIMUM subroutine we have already considered (see Section 1.3.2, Program M). For reference, we will recopy that program here, changing it so that a fixed number of cells, 100, is searched for the maximum:

```
MAX100  STJ   EXIT    Subroutine linkage
        ENT3  100     M1. Initialize.
        JMP   2F
1H      CMPA  X,3     M3. Compare.
        JGE   *+3                                   (1)
2H      ENT2  0,3     M4. Change m.
        LDA   X,3     (New maximum found)
        DEC3  1       M5. Decrease k.
        J3P   1B      M2. All tested?
EXIT    JMP   *       Return to main program. ∎
```

In a larger program containing this coding as a subroutine, the single instruction "JMP MAX100" would cause register A to be set to the current maximum value of locations X + 1 through X + 100, and the position of the maximum would appear in rI2. Subroutine linkage in this case is achieved by the instructions "MAX100 STJ EXIT" and, later, "EXIT JMP *". Because of the way the J-register operates, the exit instruction will then jump to the location following the place where the original reference to MAX100 was made.

It is not hard to obtain *quantitative* statements about the amount of code saved and the amount of time lost when subroutines are used. Suppose that a piece of coding requires k locations and that it appears in m places in the program. Rewriting this as a subroutine, we need an extra instruction STJ and an exit line for the subroutine, plus a single JMP instruction in each of the m places where the subroutine is called. This gives a total of $m + k + 2$ locations, rather than mk, so the amount saved is

$$(m - 1)(k - 1) - 3. \tag{2}$$

If k is 1 or m is 1 we cannot possibly save any space by using subroutines; this, of course, is obvious. If k is 2, m must be greater than 4 in order to gain, etc.

The amount of time lost is the time taken for the extra JMP, STJ, and JMP instructions, which are not present if the subroutine is not used; therefore if the subroutine is used t times during a run of the program, $4t$ extra cycles of time are required.

These estimates must be taken with a grain of salt, because they were given for an idealized situation. Many subroutines cannot be called simply with a single JMP instruction. Furthermore, if the coding is repeated in many parts of a program, without using a subroutine approach, the coding for each part can take advantage of special characteristics of the particular part of the program in which it lies. With a subroutine, on the other hand, the coding must be written for the most general case, not a specific case, and this will often add several additional instructions.

When a subroutine is written to handle a general case, it is often written in terms of *parameters*, values which govern the subroutine's action, but which are subject to change from one call of the subroutine to another.

The coding in the outside program which transfers control to the subroutine and gets it properly started is known as the "calling sequence." Particular values of parameters, supplied when the subroutine is called, are known as *arguments*. With our MAX100 subroutine, the calling sequence is simply "JMP MAX100", but when arguments must be supplied, a longer calling sequence is generally necessary. For example, Program 1.3.2M is a generalization of MAX100 which finds the maximum of the first n elements of the table. The parameter n appears in index register 1, and we may regard the calling sequence as

```
LD1   =n=
JMP   MAXIMUM.
```

If the calling sequence takes c memory locations, formula (2) for the amount of space saved changes to

$$(m - 1)(k - c) - \text{const} \tag{3}$$

and the time lost for subroutine linkage is slightly increased.

A further correction to the above formulas can be necessary because certain registers might need to be saved and restored. For example, in the `MAX100` subroutine, the programmer must remember that by writing "`JMP MAX100`" he is not only getting the maximum value in register A and its position in register I2; he is also setting register I3 to zero. A subroutine may destroy register contents, and this must be kept in mind. In order to prevent `MAX100` from changing the setting of rI3, it would be necessary to insert "`ST3 TEMP`" just after `MAX100` and also "`LD3 TEMP`" just before `EXIT`; it is an extra two lines of code, plus perhaps another memory cell for `TEMP`, and an extra four machine cycles on every call of the subroutine. (Of course, the subroutine now destroys the contents of memory cell `TEMP`, and the programmer must keep this in mind; it seems we have solved one problem only to have created another. There are two common solutions to this possible dilemma: (a) essentially make `TEMP` part of the subroutine, by changing the instructions to "`ST3 3F`" and "`LD3 3F`" and adding the line "`3H CON 0`" just after `EXIT`; or (b) use a systematic method for the names of temporary storage locations in subroutines, as explained later in this section or as explained in Chapter 8.)

A subroutine may be regarded as an *extension* of the computer's machine language. With the `MAX100` subroutine in memory, we now have a single instruction (namely, "`JMP MAX100`") which is a maximum-finder. It is important to define the effect of each subroutine just as carefully as the machine language operators themselves have been defined, and so the programmer should be sure to write down the characteristics of each subroutine, even though he himself will be the only one to make use of it. In the case of `MAXIMUM` as given in Section 1.3.2, the characteristics are as follows:

$$\begin{aligned}
&\text{Calling sequence:} && \texttt{JMP MAXIMUM.}\\
&\text{Entry conditions:} && \text{rI1} = n;\quad \text{assume}\quad n \geq 1.\\
&\text{Exit conditions:} && \text{rA} = \max_{1\leq k\leq n} \texttt{CONTENTS}\,(\texttt{X} + k)\\
& && \quad = \texttt{CONTENTS}\,(\texttt{X} + (\text{rI2}));\\
& && \text{rI3} = 0;\ \text{rJ and CI are also affected.}
\end{aligned} \tag{4}$$

(We will customarily omit mention of the fact that register J and the comparison indicator are affected by a subroutine; it has been mentioned here only for completeness.) Note that rX and rI1 are unaffected by the action of the subroutine, for otherwise these registers would have been mentioned in the exit conditions.

Now let us consider *multiple entrances* to subroutines. Suppose that we have a program which requires the general subroutine MAXIMUM, but which most frequently wants to use the special case MAX100, with $n = 100$. The two can be combined as follows:

```
MAX100  ENT3   100      First entrance
MAXN    STJ    EXIT     Second entrance
        JMP    2F       Continue as in (1).          (5)
...
EXIT    JMP    *
```

Subroutine (5) is essentially the same as (1), with the first two instructions interchanged; we have used the fact that "ENT3" does not change the setting of the J-register. If we were to add a *third* entrance, MAX50, to this subroutine, we could insert the code

```
MAX50   ENT3   50
        JSJ    MAXN                                 (6)
```

at the beginning. (Recall that "JSJ" means jump without changing register J.)

When the number of parameters is small, it is often desirable to transmit them to a subroutine either by having them in convenient registers (as we have used rI3 to hold the parameter n in MAXN and as we used rI1 to hold the parameter n in MAXIMUM), or by storing them in fixed memory cells. Another way to supply arguments which is often convenient is to simply list them *after* the JMP instruction; the subroutine may refer to its parameters because it knows the J-register setting.

For example, if we wanted to make the calling sequence for MAXN be

```
        JMP    MAXN
        CON    n                                    (7)
```

then the subroutine could be written

```
MAXN    STJ    *+1
        ENT1   *        rI1 ← rJ.
        LD3    0,1      rI3 ← n.
        JMP    2F       Continue as in (1).          (8)
...
        J3P    1B
        JMP    1,1      Return. ∎
```

On machines such as the 709 for which linkage is ordinarily done by putting the exit location in an index register, the above procedure is particularly convenient. It is also useful when a fairly large number of arguments is to be passed to a subroutine, as well as in conjunction with programs written by compilers

(see Section 12.2.7). The technique of multiple entrances which we used above often fails in this case, however; we could "fake it" by writing

```
MAX100  STJ   1F
        JMP   MAXN
        CON   100
1H      JMP   *
```

but this is not as attractive as (5).

A technique similar to that of listing arguments after the jump is normally used for subroutines with *multiple exits*. Multiple exit means that we want the subroutine to return to one of several different locations, depending on conditions detected by the subroutine. In the strictest sense, the location to which a subroutine exits is a parameter; so if there are several places to which it should exit, depending on the circumstances, these should be supplied as arguments. Our final example of the "maximum" subroutine will have two entrances and two exits. The calling sequence is:

For general n	For $n = 100$
`ENT3  n`	
`JMP   MAXN`	`JMP  MAX100`
Exit here if max ≤ 0 or max $\geq$ rX.	Exit here if max ≤ 0 or max $\geq$ rX.
Exit here if $0 <$ max $<$ rX.	Exit here if $0 <$ max $<$ rX.

(In other words, exit is made to the location *two* past the jump when the maximum value is positive and less than the contents of register X.) The subroutine for these conditions is easily written:

```
MAX100  ENT3  100      Entrance for n = 100
MAXN    STJ   EXIT     Entrance for general n
        JMP   2F       Continue as in (1).
...
        J3P   1B
        JANP  EXIT     Is max positive?                    (9)
        STX   TEMP
        CMPA  TEMP
        JGE   EXIT     Is it less than rX?
        INC3  1        Set rI3 ← 1.
EXIT    JMP   *,3      Return to proper place. ∎
```

In summary, subroutines are often desirable for saving space in a program and reducing its complexity. As always, we don't get something for nothing, and some expense in running time occurs. If the extra time for calling a subroutine is small compared to the total execution time for that subroutine, and if it is fairly long or is referred to quite often, then writing it as a subroutine is far superior to rewriting the code over and over in the program. For typical

uses of subroutines in a larger program, see the examples in Section 1.4.2.1 and throughout the assembler in Chapter 9.

Subroutines may call on other subroutines; in complicated programs it is not unusual to have subroutine calls nested more than 10 deep. The restriction that must be followed when using linkage as described here, however, is that no subroutine may call on any other subroutine which is (directly or indirectly) calling on it. For example,

[Main program]

```
        ⋮
      JMP A
        ⋮
```

[Subroutine A]

```
A       STJ EXITA
        ⋮
        JMP B
        ⋮
EXITA   JMP *
```

[Subroutine B]

```
B       STJ EXITB
        ⋮
        JMP C
        ⋮
EXITB   JMP *
```

[Subroutine C]

```
C       STJ EXITC
        ⋮
        JMP A
        ⋮
EXITC   JMP *
```

(10)

If the main program calls on A, which calls B, which calls C, and then C calls on A, the address in EXITA referring to the main program is destroyed, and there is no way to return to the main program. A similar remark applies to all temporary storage cells and registers used by each subroutine. It is possible to make subroutine linkages which will handle the above "recursive" situation properly, and these will be discussed in Chapter 8.

We conclude this section by discussing briefly how we might go about writing a complex and lengthy program. How can we decide what kind of subroutines we will need, and what calling sequences should be used? One successful way to determine this is to use an iterative procedure:

Step 0 (Initial idea). First we vaguely decide upon the general plan of attack in the program.

Step 1 (A rough sketch of the program). We start now by writing the middle portions of the program, assuming that initialization in some form has taken place. (Lacking clairvoyance, we cannot figure out what initialization will be necessary until the rest of the program is written.) With this assumption, we write down the steps the program is to carry out, either in English flow outline form as used in the algorithms of this book, or in flow-chart form, or immediately in some computer language, according to personal preference. Whenever something occurs which seems likely to occur elsewhere or which has already occurred elsewhere, we define a subroutine to do that job. We do not write the subroutine at this point; we continue writing the main program, assuming the subroutine has performed its task. Finally, when the main program has been sketched, we tackle the subroutines in turn, trying to take the most complex subroutines first and then their sub-subroutines, etc. In this manner we will come up with a list of subroutines. The actual function of each subroutine has probably already changed several times, so that the first parts of our sketch will

by now be incorrect; but that is all right, for it is merely a sketch. For each subroutine we now have a reasonably good idea as to how it will be called and how general-purpose it should be. It usually pays to extend the generality of each subroutine a little.

Step 2 (First working program). This step goes in the opposite direction from step 1. We now write in computer language, say `MIXAL`; we start this time with the lowest level subroutines, and do the main program last. As far as possible, we try never to write any instructions which call a subroutine before the subroutine itself has been coded. (In step 1, we tried the opposite, never considering a subroutine until all of its calls had been written.)

As more and more subroutines are written during this process, our confidence gradually grows, since we are continually extending the power of the machine we are programming. After an individual subroutine is coded, we should immediately prepare a complete description of what it does, and what its calling sequences are, as in (4). It is also important not to overlay temporary storage cells; it may very well be disastrous if every subroutine refers to location `TEMP`, although when preparing the sketch in step 1, it is convenient not to worry about this problem. An obvious way to overcome overlay worries is to have each subroutine use only its own temp storage, but if this is too wasteful of space, another scheme which does fairly well is to name the cells `TEMP1`, `TEMP2`, etc.; the numbering within a subroutine starts with `TEMP`j, where j is one higher than the greatest number used by any of the sub-subroutines of this subroutine.

Step 3 (Reexamination). The result of step 2 should be very nearly a working program, but it may be possible to improve on it. A good way is to reverse direction again, studying for each subroutine *all* of the calls made on it. It may well be that the subroutine should be enlarged to do some of the more common things which are always done by the outside routine just before or after it uses the subroutine. Perhaps several subroutines should be merged into one; or perhaps a subroutine is called only once (or, if we are fortunate, perhaps one is never called!) and should not be a subroutine at all.

At this point, it is often a good idea to scrap everything and start over again at step 1! This is not intended to be a facetious remark; the time spent in getting this far has not been wasted, for we have learned a great deal about our problem. We will probably know of several improvements that can be made to the organization of the program; there is no reason to be afraid to go back to step 1—it will be much easier to go through steps 1 and 2 again after a program has been done already. Moreover, we will quite probably save as much debugging time later on as it will take to rewrite the program. Some of the best computer programs ever written owe much of their success to the fact that at about this stage all the work was unintentionally lost and the authors had to begin again.

On the other hand, there is probably never a point when a complex computer program cannot be improved somehow, so steps 1 and 2 should not be repeated indefinitely; see the further discussion in Section 9.3.10. When significant improvements can clearly be made, it is well worth the additional time required to start over.

Step 4 (Initialization). Go through the program, allocate storage for it, and figure out exactly what needs to be initialized; then write the initialization part of the program last. (This step is something like writing a textbook: Chapter 1 is usually written last.)

Step 5 (Debug the program). Here we may look at our program in still another direction from the three that were used in steps 1, 2, and 3—we study the program in the order in which the computer will *perform* it.

Debugging is an art that needs much further study, and the way to approach it is highly dependent on the facilities that are available at each computer installation. A good start towards effective debugging is often the preparation of an appropriate "test deck," as explained in Section 9.3.10. The most effective debugging techniques seem to be those which are designed and built into the program itself—many of today's best programmers will devote nearly half of their programs to facilitating the debugging process on the other half; the first half, which usually consists of fairly straightforward routines, will eventually be thrown away, but the net result is a surprising gain in productivity.

Another good debugging practice is to keep a record of every mistake that is made. Even though this will probably be quite embarrassing, such information is invaluable to anyone doing research on the debugging problem, and it will also help you learn how to reduce the number of future errors.

EXERCISES

1. [*10*] State the characteristics of subroutine (5), just as (4) gives the characteristics of Subroutine 1.3.2M.

2. [*10*] Suggest code to substitute for (6) without using the JSJ instruction.

3. [*M15*] Complete the information in (4) by stating exactly what happens to register J and the comparison indicators as a result of the subroutine; state also what happens if register I1 is not positive.

▶ **4.** [*21*] Write a subroutine that generalizes MAXN by finding the maximum value of X[1], X[1 + r], X[1 + 2r], . . . , X[n], where r and n are parameters. Give a special entrance for the case $r = 1$.

5. [*21*] Suppose that MIX did not have a J-register. Invent a means for subroutine linkage which does not use register J, and give an example of your invention by writing a MAX100 subroutine effectively equivalent to (1). State the characteristics of this subroutine in a fashion similar to (4).

▶ **6.** [*26*] Suppose MIX did not have a MOVE operator; write a subroutine entitled MOVE such that the calling sequence

```
JMP   MOVE
NOP   A,I(F)
```

has an effect just the same as "MOVE A,I(F)" if the latter were admissible. The only differences should be the effect on register J and the fact that the time to execute the subroutine will be somewhat longer.

1.4.2. Coroutines

Subroutines are special cases of more general program components, called "coroutines." In contrast to the unsymmetric relationship between a main routine and a subroutine, there is complete symmetry between coroutines, which *call on each other*.

To understand the coroutine concept, let us consider another way of thinking about subroutines. The viewpoint adopted in the previous section was that a subroutine merely was an extension of the computer hardware, introduced to save lines of coding. This may be true, but another point of view is possible: We may consider the main program and the subroutine as a *team* of programs, with each member of the team having a certain job to do. The main program, in the course of doing its job, will activate the subprogram; the subprogram performs its own function and then activates the main program. We might stretch our imagination to believe that, from the subroutine's point of view, when it exits *it* is calling the *main* routine; the main routine continues to perform its duty, then "exits" to the subroutine. The subroutine acts, then calls the main routine again.

This somewhat far-fetched philosophy actually takes place with coroutines, when it is impossible to distinguish which is a subroutine of the other. Suppose we have coroutines `A` and `B`; when programming `A`, we may think of `B` as our subroutine, but when programming `B`, we may think of `A` as our subroutine. That is, in coroutine `A`, the instruction "`JMP B`" is used to activate coroutine `B`. In coroutine `B` the instruction "`JMP A`" is used to activate coroutine `A` again. It represents teamwork as in a relay race. Whenever a coroutine is activated, it resumes execution of its program at the point where the action was last suspended.

With `MIX`, such linkage between coroutines `A` and `B` is done by including the following four instructions in the program:

```
A    STJ  BX          B    STJ  AX
AX   JMP  A1          BX   JMP  B1
```
(1)

This requires four machine cycles for transfer of control each way. Initially `AX` and `BX` are set to jump to the starting places of each coroutine, `A1` and `B1`. Suppose we start up coroutine `A` first, at location `A1`. When it executes "`JMP B`" from location `A2`, say, the instruction in location `B` stores rJ in `AX`, which then says "`JMP A2+1`". The instruction in `BX` gets us to location `B1`, and after coroutine `B` begins its execution, it will eventually get to an instruction "`JMP A`" in location `B2`, say. We store rJ in `BX` and jump to location `A2+1`, continuing the execution of coroutine `A` until it again jumps to `B`, which stores J in `AX` and jumps to `B2+1`, etc.

The essential difference between routine-subroutine and coroutine-coroutine linkage, as can be seen by studying the example above, is that a subroutine is always initiated *at its beginning*, i.e., at a fixed place, while the main routine or a coroutine is always initiated *at the place following* where it last terminated.

Coroutines connected with algorithms for input and output arise most naturally in practice. For example, suppose it is the duty of coroutine A to read cards and to perform some transformation on the input, reducing it to a sequence of items. Another coroutine, which we will call B, does further processing of these items, and prints the answers; B will periodically call for the successive input items found by A. Thus, coroutine B jumps to A whenever it wants the next input item, and coroutine A jumps to B whenever an input item has been found. (The reader may say, "Well, B is the main program and A is merely a *subroutine* for doing the input." This, however, becomes less true when the process A is very complicated; indeed, we can imagine A as the main routine and B as a subroutine for doing the output, and the above description remains valid. The usefulness of the coroutine idea emerges midway between these two extremes, when both A and B are complicated and each one calls the other in numerous places. It is rather difficult to find short, simple examples of coroutines which illustrate the importance of the idea; the most useful coroutine applications are generally quite lengthy.

In order to study coroutines in action, let us consider a "contrived" example. Suppose we want to write a program that translates one code into another. The input code to be translated is a sequence of alphameric characters terminated by a period, e.g.,

```
A2B5E3426FG0ZYW3210PQ89R.                                    (2)
```

which has been punched onto cards; blank columns appearing on these cards are to be ignored. This input is to be understood as follows, from left to right: If the next character is a digit (i.e., 0, 1, . . . , 9), say n, it indicates $(n + 1)$ repetitions of the following character, whether the following character is a digit or not. A nondigit simply denotes itself. The output of our program is to consist of the sequence indicated in this manner and separated into groups of three characters each (where the last group may have less than three characters). For example, (2) should be translated by our program into

```
ABB  BEE  EEE  E44  446  66F  GZY  W22  220  0PQ  999  999  999  R.   (3)
```

Note that 3426F does not mean 3427 repetitions of the letter F; it means 4 fours and 3 sixes followed by F. Our program is to punch the output onto cards, with sixteen groups of three on each card.

To accomplish this translation, we will write two coroutines and a subroutine. The subroutine, called NEXTCHAR, is designed to successively find nonblank characters of input, and to put the next character into register A:

```
01    * SUBROUTINE FOR CHARACTER INPUT
02    READER    EQU   16              Unit number of card reader
03    INPUT     ORIG  *+16            Place for input cards
04    NEXTCHAR  STJ   9F              Entrance to subroutine
05              JXNZ  3F              Initially rX = 0
06    1H        J6N   2F              Initially rI6 = 0
07              IN    INPUT(READER)   Read next card.
```

```
08          JBUS    *(READER)       Wait for completion.
09          ENN6    16              Let rI6 point to first word.
10    2H    LDX     INPUT+16,6      Get next word of input.
11          INC6    1               Advance pointer.
12    3H    ENTA    0
13          SLAX    1               Next character → rA.
14    9H    JANZ    *               Skip blanks.
15          JMP     NEXTCHAR+1      ∎
```

This subroutine has the following characteristics:

Calling sequence: `JMP NEXTCHAR`.

Entry conditions: rI6 points to next word, or rI6 = 0 indicating that a new card must be read; rX = characters yet to be used.

Exit conditions: rA = next nonblank character of input; rX, rI6 set for next entry to `NEXTCHAR`.

Our first coroutine, called `IN`, finds the characters of the input code with the proper replication:

```
16    * FIRST COROUTINE
17    2H    INCA    30              Nondigit found
18          JMP     OUT             Send it to OUT coroutine.
19    IN1   JMP     NEXTCHAR        Get character.
20          DECA    30
21          JAN     2B              Is it a letter?
22          CMPA    =10=
23          JGE     2B              Is it a special character?
24          STA     *+1(0:2)        Digit n found
25          ENT5    *               rI5 ← n.
26          JMP     NEXTCHAR        Get next character.
27          JMP     OUT             Send it to OUT coroutine.
28          DEC5    1               Decrease n by 1.
29          J5NN    *-2             Repeat if necessary.
30          JMP     IN1             Begin new cycle. ∎
```

(Recall that in `MIX`'s character code, the digits 0–9 have codes 30–39.) This coroutine has the following characteristics:

Calling sequence: `JMP IN`.

Exit conditions (when jumping to `OUT`): rA = next character of input with proper replication; rI4 unchanged from its value at entry.

Entry conditions (upon return): rA, rX, rI5, rI6 should be unchanged from their values at the last exit.

The other coroutine, called OUT, puts the code into three-digit groups and punches the cards:

```
31      * SECOND COROUTINE
32              ALF                     Constant used for blanking
33      OUTPUT  ORIG    *+16            Buffer area for answers
34      PUNCH   EQU     17              Unit number for card punch
35      OUT1    ENT4    -16             Start new output card.
36              ENT1    OUTPUT
37              MOVE    -1,1(16)        Set output area to blanks.
38      1H      JMP     IN              Get next translated character.
39              STA     OUTPUT+16,4(1:1) Store in output.
40              CMPA    PERIOD          Is it "."?
41              JE      9F
42              JMP     IN              If not, get another character.
43              STA     OUTPUT+16,4(2:2) Store it.
44              CMPA    PERIOD          Is it "."?
45              JE      9F
46              JMP     IN              If not, get another character.
47              STA     OUTPUT+16,4(3:3) Store it.
48              CMPA    PERIOD          Is it "."?
49              JE      9F
50              INC4    1               Move to next word in output.
51              J4N     1B              End of card?
52      9H      OUT     OUTPUT(PUNCH)   If so, punch.
53              JBUS    *(PUNCH)        Wait for completion.
54              JNE     OUT1            Return for more, unless
55              HLT                         "." was sensed.
56      PERIOD  ALF                     ▮
```

This coroutine has the following characteristics:

Calling sequence:	JMP OUT.
Exit conditions (when jumping to IN):	rA, rX, rI5, rI6 unchanged from their value at entry.
Entry conditions (upon return):	rA = next character of input with proper replication; rI4 unchanged from its value at the last exit.

To complete the program, we need to write the coroutine linkage [cf. (1)] and to provide the proper initialization. Initialization of coroutines tends to be a little tricky, although not really difficult.

```
57      * INITIALIZATION AND LINKAGE
58      START   ENT6    0               Initialize rI6 for NEXTCHAR.
59              ENTX    0               Initialize rX for NEXTCHAR.
60              JMP     OUT1            Start with OUT (cf. exercise 2).
```

```
61    OUT     STJ   INX          Coroutine linkage
62    OUTX    JMP   OUT1
63    IN      STJ   OUTX
64    INX     JMP   IN1
65            END   START
```

This completes the program. The reader should study it carefully, noting in particular how each coroutine can be written independently as though the other coroutine were its subroutine.

The entry and exit conditions for the IN and OUT coroutines mesh perfectly in the above program. In general, we would not be so fortunate, and the coroutine linkage would also include loading and storing appropriate registers. For example, if OUT would destroy the contents of register A, the coroutine linkage would become

```
OUT    STJ   INX
       STA   HOLDA     Store A when leaving IN.
OUTX   JMP   OUT1                                  (4)
IN     STJ   OUTX
       LDA   HOLDA     Restore A when leaving OUT.
INX    JMP   IN1
```

There is an important relation between coroutines and *multiple-pass algorithms*. For example, the translation process we have just described could have been done in two distinct passes: We could first have done just the IN coroutine, applying it to the entire input and writing each character with the proper amount of replication onto magnetic tape. After this was finished, we could rewind the tape and then do just the OUT coroutine, taking the characters from tape in groups of three. This would be called a "two-pass" process. (Intuitively, a "pass" denotes a complete scan of the input. This definition is not precise, and in many algorithms the number of passes taken is not at all clear; but the intuitive concept of "pass" is useful in spite of its vagueness.)

Figure 22(a) illustrates a four-pass process. Quite often we will find that the same process can be done in just one pass, as shown in part (b) of the figure, if we substitute four coroutines A, B, C, D for the respective passes A, B, C, D. Coroutine A will jump to B when pass A would have written an item of output on tape 1; coroutine B will jump to A when pass B would have read an item of input from tape 1, and B will jump to C when pass B would have written an item of output on tape 2; etc.

Conversely, a process done by n coroutines can often be transformed into an n-pass process. Due to this correspondence it is worth while comparing multipass algorithms to one-pass algorithms:

a) *Psychological difference.* A multipass algorithm is generally easier to create and to understand than a one-pass algorithm for the same problem.

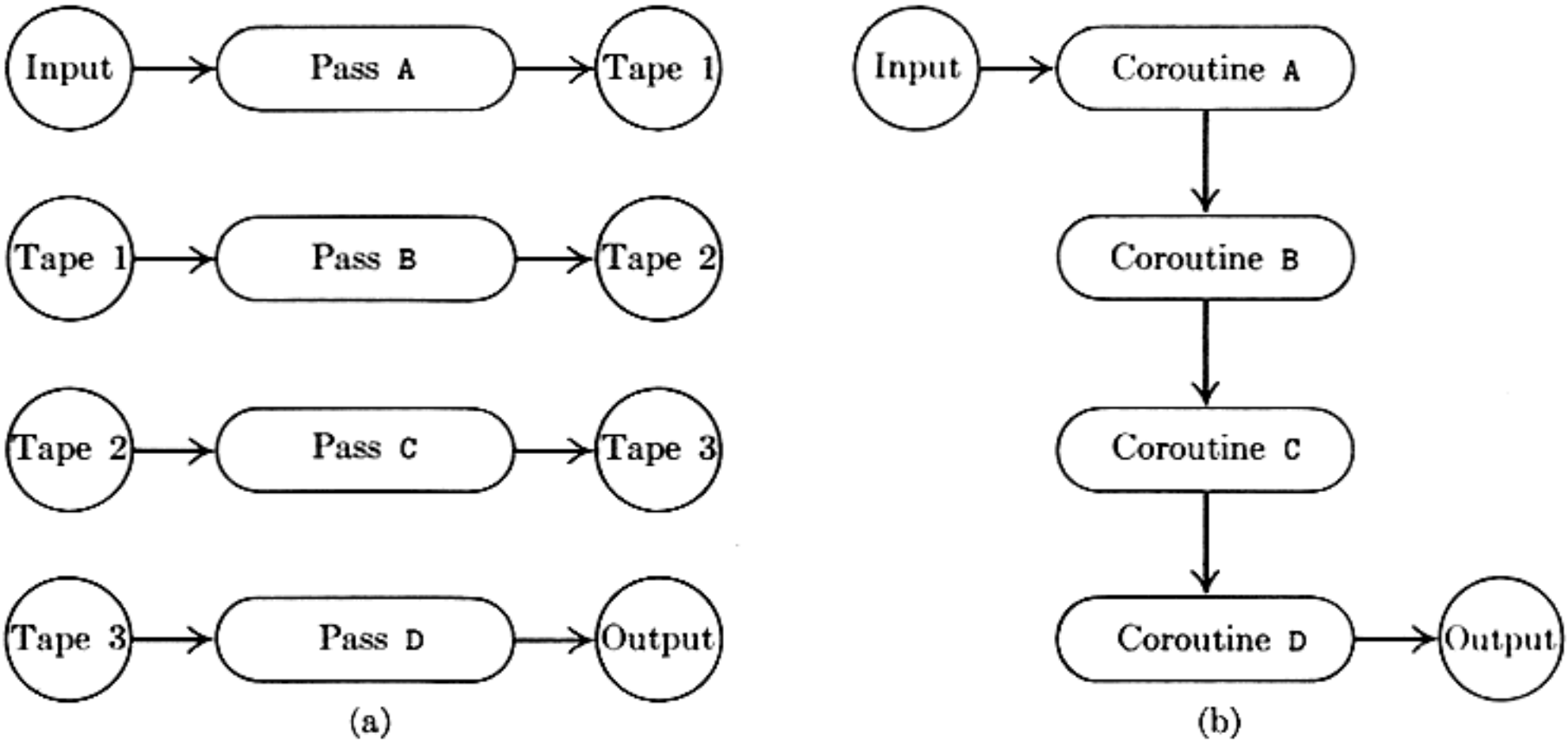


Fig. 22. Passes: (a) a four-pass algorithm, and (b) a one-pass algorithm.

Breaking a process down into a sequence of small steps which happen one after the other is easier to comprehend than considering an involved process in which all of these things go on simultaneously.

Also, if a very large problem is being done and if many people are to cooperate in producing the computer program, a multipass algorithm provides a natural way to divide up the job.

These advantages of a multipass algorithm are present in coroutines as well, since each coroutine can be written essentially separate from the others, and the linkage makes an apparently multipass algorithm into a single-pass process.

b) *Time difference.* The time required to pack, write, read, and unpack intermediate data between passes (e.g., the information in Fig. 22 on tapes) is avoided in a one-pass algorithm. For this reason, a one-pass algorithm will be faster.

c) *Space difference.* The one-pass algorithm requires space to hold all the programs in memory simultaneously, while a multipass algorithm requires space for only one at a time. This may affect the speed, even to a greater extent than indicated in statement (b). For example, many computers have a limited amount of "fast memory" and a larger amount of slower memory; if each pass can fit into the fast memory, the result will be considerably faster than if we use coroutines in a single pass (since the use of coroutines would presumably force most of the program to appear in the slower memory).

Occasionally, there is a need to design algorithms for several computer configurations at once, some of which have larger memory capacity than others. In this case it would be possible to write the program in terms of coroutines, and to let the memory size govern the number of passes: load together as many coroutines as feasible, and supply input or output subroutines for the missing links.

Although this relationship between coroutines and passes is important, we should keep in mind that not all coroutine applications can be split into multipass algorithms. For example, if coroutine B gets input from A and also sends back crucial information to A, it cannot be converted into pass A followed by pass B.

Conversely, it is clear that not all multipass algorithms can be converted to coroutines. Some algorithms are inherently multipass; for example, the second pass may require cumulative information from the first pass (like the total number of occurrences of a certain word in the input). There is an old joke worth noting in this regard:

> *Little old lady, riding a bus. "Little boy, can you tell me how I can get off at Pasadena Street?"*
> *Little boy. "Just watch me, and get off two stops before I do."*

(The joke is that the little boy gives a two-pass algorithm.)

So much for multipass algorithms. We will see further examples of coroutines in numerous places throughout the book, for example, as part of the buffering schemes in Section 1.4.4. Coroutines also play an important role in discrete system simulation; see Section 2.2.5. The important idea of *recursive coroutines* is introduced in Chapter 8, and some interesting applications of this idea may be found in Section 10.6.

EXERCISES

1. [*10*] Explain why short, simple examples of coroutines are hard for the author of a textbook to find.

▶ **2.** [*20*] The program in the text starts up the `OUT` coroutine first. What would happen if `IN` were the first to be executed, i.e., if line 60 were changed from "`JMP OUT1`" to "`JMP IN1`"?

3. [*20*] True or false: The three "`CMPA PERIOD`" instructions within `OUT` may all be omitted, and the program would still work.

4. [*20*] Show how coroutine linkage analogous to (1) can be given for several real-life computers you are familiar with.

5. [*15*] Suppose that both coroutines `IN` and `OUT` would want the contents of register A untouched between exit and entry; thus, assume that wherever the instruction "`JMP IN`" occurs within `OUT`, the contents of register A are to be unchanged when control returns to the next line, and make a similar assumption about "`JMP OUT`" within `IN`. What coroutine linkage is needed? [Cf. (4).]

▶ **6.** [*22*] Give coroutine linkage analogous to (1) for the case of *three* coroutines, `A`, `B`, `C`, each of which can jump to either of the other two. (Whenever a coroutine is activated, it begins where it last left off.)

▶ **7.** [*30*] Write a program which *reverses* the translation done by the program in the text, i.e., it would convert cards punched like (3) into cards punched like (2). The output should be as short a string of characters as possible, so that the zero before the Z in (2) would not really be produced from (3).

1.4.3. Interpretive routines.

In this section we will investigate a common type of computer program, the *interpretive routine* (which will be called *interpreter* for short). An interpretive routine is a computer program that performs the instructions of another program, where the other program is written in some machine-like language. By a machine-like language, we mean some way of representing instructions having, say, operation codes, addresses, etc. (This definition, like most definitions of today's computer terms, is not precise, nor should it be; it is impossible to draw the line exactly and to say just which programs are interpreters and which are not.)

Historically, the first interpreters were built around machine-like languages designed especially for simple programming; it was to be a language easier to use than machine language. The rise of programming languages has gradually made this function of interpretive routines obsolete, but interpreters are by no means dying out. On the contrary, their use has been growing, to the extent that effective use of interpretive routines may be regarded as one of the essential characteristics of modern programming. The new applications of interpreters are made chiefly for the following reasons:

a) to represent a fairly complicated sequence of decisions and actions in a compact, efficient manner, and
b) to communicate between passes of a multipass program.

In these cases, special purpose machine-like languages are developed for use in a particular program, and often the machine is the only individual who ever writes "programs" in this language. (Today's expert programmers are also good machine designers, as they not only create the interpretive routine, but also invent the language to be interpreted.)

The interpretive technique has the further advantage of being relatively machine-independent—only the interpreter must be rewritten when changing machines. Furthermore, helpful debugging aids can readily be built in to an interpretive system.

Examples of interpreters of type (a) appear in several places later in this book, e.g., the recursive interpreter in Chapter 8, the "Parsing Machine" interpreter in Section 10.6.4, the `XMIX` interpreter in Section 12.2, and the `TROLL` interpreter in Section 12.3.4.

Another example would be a program in which a great many special cases arise, all similar, but having no really simple pattern. Thus, consider writing a compiler (cf. Chapter 12) in which we would like to generate efficient object programs for adding two quantities together. There might be ten classes of quantities (e.g., constants, simple variables, temp storages, subscripted variables, a quantity in an accumulator or index register, fixed and floating point, etc.) and the combination of all pairs yields 100 different cases. A long program would be required to do the proper thing in each case; the interpretive solution to this problem is to make up a language whose "instructions" fit in one byte. Then

keep a table of 100 "programs" in this language, where each program consists of one to five instructions so it fits in a single word. The idea is merely to pick out the appropriate table entry and to perform the program found there. This technique is simple and efficient.

An example of an interpreter of type (b) appears in the article "Computer-Drawn Flowcharts" by D. E. Knuth, *CACM* **6** (1963), 555–563. In a multipass program, the earlier passes must transmit information to the later passes. This information is often transmitted most efficiently in a somewhat machine-like language, as a set of instructions for the later pass; the later pass is then nothing but a special purpose interpretive routine, and the earlier pass is a special purpose "compiler." This philosophy of multipass operation may be characterized as *telling* the later pass what to do, whenever possible, rather than simply presenting it with a lot of facts and asking it to *figure out* what to do.

Another example of an interpreter of type (b) occurs in connection with compilers for special languages. If the language includes many features which are not easily done on the machine except by subroutine, the resulting object programs will be very long sequences of subroutine calls. This would happen, for example, if the language were concerned primarily with multiple-precision arithmetic. In such a case the object program would be considerably shorter if it were expressed in an interpretive language. An illustration of this approach may be found in Section 12.3, where the `TROL` language and its interpreter are discussed. See also the book *ALGOL 60 Implementation*, by B. Randell and L. J. Russell (New York: Academic Press, 1964), which describes a compiler to translate from ALGOL 60 into an interpretive language, and which also describes the interpreter for that language; and see "An ALGOL 60 Compiler," by Arthur Evans, Jr., *Ann. Rev. Auto. Programming* **4** (1964), 87–124, for examples of interpretive routines used *within* a compiler.

There is another way to look at a program written in interpretive language—it may be regarded as a series of subroutine calls, one after another. Such a program may in fact be expanded into a long sequence of calls on subroutines, and, conversely, such a sequence can usually be packed into a coded form which is readily interpreted. The advantages of interpretive techniques are the compactness of representation, the machine independence, and the increased diagnostic capability. An interpreter can usually be written so that the amount of time spent in interpretation of the code itself and branching to the appropriate routine is negligible.

1.4.3.1. A MIX simulator. When the language presented to an interpretive routine is the machine language of another computer, the interpreter is often called a simulator.

In the author's opinion, entirely too much programmers' time has been spent in writing such simulators and entirely too much computer time has been wasted in using them. The motivation for simulators is simple: A computer installation buys a new machine and still wants to run programs written for the old machine (rather than rewriting the programs). However, this usually costs

more and gives poorer results than if a special task force of programmers were given temporary employment to do the re-programming. For example, the author once participated in such a re-programming project, and a serious error was discovered in the original program which had been in use for several years; the new program worked at five times the speed of the old, besides giving the right answers for a change! (Not all simulators are bad; for example, it is usually advantageous for a computer manufacturer to simulate a new machine before it has been built, so that software for this machine may be developed as soon as possible. But this is a very specialized application.) An extreme example of the inefficient use of computer simulators is the true story of machine A simulating machine B running a program which simulates machine C! This is the way to make a large, expensive computer give poorer results than its cheaper cousin.

In view of all this, why should a simulator rear its head in this book? There are two reasons:

a) The simulator we will describe below is a good example of a typical interpretive routine; the basic techniques employed in interpreters are illustrated here. It also illustrates the use of subroutines in a moderately long program.

b) We will describe a simulator of the MIX computer, written in (of all things) the MIX language. This will facilitate the writing of MIX simulators for most computers, which are similar; the coding of our program intentionally avoids making heavy use of MIX-oriented features. A MIX simulator will be of advantage as a teaching aid in conjunction with this book and possibly others.

Computer simulators as described in this section should be distinguished from *discrete system simulators*, which are important programs studied in Section 2.2.5.

Now let us turn to the task of writing a MIX simulator. The numbering of MIX's instructions LDA, LD1, . . . , LDX and other similar ones suggests that we keep the simulated contents of these registers in sequential locations, as follows:

```
AREG, I1REG, I2REG, I3REG, I4REG, I5REG, I6REG, XREG, JREG, ZERO.
```

Here ZERO is a "register" filled with zeros at all times. The position of JREG and ZERO is suggested by the operation code numbers of the instructions STJ and STZ.

In keeping with our philosophy of writing the simulator as though it were not done with MIX, we will treat the signs as independent parts of a word. For example, many computers cannot represent the number "minus zero", while MIX definitely can; therefore we will always treat signs specially in this program. The locations AREG, I1REG, . . . , ZERO will always contain the absolute values of the corresponding register contents; another set of locations in our program, called SIGNA, SIGN1, . . . , SIGNZ will contain $+1$ or -1, depending on whether the sign of the corresponding register is plus or minus.

An interpretive routine generally has a central control section which is called into action between interpreted instructions. In our case, the program transfers to location CYCLE at the end of each simulated instruction.

The control routine does the things common to all instructions, unpacks the instruction into its various parts, and puts the parts into convenient places for later use. The program below sets

rI6 = location of the next instruction;
rI5 = M (address of present instruction, plus indexing);
rI4 = operation code of present instruction;
rI3 = F-field of present instruction;
INST = present instruction.

Program M.

```
01     * MIX SIMULATOR
02               ORIG   3500          (Simulated memory is in locations 0000 up.)
03     BEGIN     STZ    TIME(0:2)
04               STZ    OVTOG         OVTOG is the simulated overflow toggle.
05               STZ    COMPI         COMPI, ±1 or 0, is comparison indicator.
06               ENT6   0             Take first instruction from location zero.
07     CYCLE     LDA    CLOCK         Beginning of control routine
08     TIME      INCA   0             This address is set to execution time
09               STA    CLOCK               of previous instruction, see line 33.
10               LDA    0,6           Instruction to simulate → rA.
11               STA    INST
12               INC6   1             Advance location counter.
13               LDX    INST(1:2)     Get absolute value of address.
14               SLAX   5             Attach sign to address.
15               STA    M
16               LD2    INST(3:3)     Examine index field.
17               J2Z    1F            Is it zero?
18               DEC2   6
19               J2P    INDEXERROR    Illegal index specified?
20               LDA    SIGN6,2       Get sign of index register.
21               LDX    I6REG,2       Get magnitude of index register.
22               SLAX   5             Attach sign.
23               ADD    M             Signed addition for indexing.
24               CMPA   ZERO(1:3)     Is result too large?
25               JNE    ADDRERROR
26               STA    M             Address has been found.
27     1H        LD3    INST(4:4)     F-field → rI3.
28               LD5    M             M → rI5.
29               LD4    INST(5:5)     C-field → rI4.
30               DEC4   63
31               J4P    OPERROR       Is op code ≥ 64?
32               LDA    OPTABLE,4(4:4)  Get execution time from table.
33               STA    TIME(0:2)
34               LD2    OPTABLE,4(0:2)  Get address of proper routine.
35               JNOV   0,2           Jump to operator.
36               JMP    0,2           (Protect against overflows.) ▮
```

The reader's attention is called particularly to lines 34–36: a "switching table" of the 64 operators is part of the simulator, allowing it to jump rapidly

to the correct routine for the current instruction. This is an important time-saving technique (cf. exercise 1.3.2–9).

The 64-word switching table, called OPTABLE, gives also the execution time for the various operators; the following lines indicate the contents of that table:

```
37          NOP     CYCLE(1)
38          ADD     ADD(2)
39          SUB     SUB(2)
40          MUL     MUL(10)
41          DIV     DIV(12)
42          HLT     SPEC(1)
43          SLA     SHIFT(2)
44          MOVE    MOVE(1)
45          LDA     LOAD(2)
46          LD1     LOAD,1(2)
                . . .
51          LD6     LOAD,1(2)
52          LDX     LOAD(2)
53          LDAN    LOADN(2)
54          LD1N    LOADN,1(2)
                . . .
60          LDXN    LOADN(2)
61          STA     STORE(2)
                . . .
69          STJ     STORE(2)
70          STZ     STORE(2)
71          JBUS    JBUS(1)
72          IOC     IOC(1)
73          IN      IN(1)
74          OUT     OUT(1)
75          JRED    JRED(1)
76          JMP     JUMP(1)
77          JAP     REGJUMP(1)
                . . .
84          JXP     REGJUMP(1)
85          INCA    ADDROP(1)
86          INC1    ADDROP,1(1)
                . . .
92          INCX    ADDROP(1)
93          CMPA    COMPARE(2)
                . . .
100   OPTABLE   CMPX    COMPARE(2)
```

Operation code table
typical entry is
"OP address (time)"

(The entries for operators LD*i*, LD*i*N, and INC*i* have an additional ",1" to set the (3:3) field nonzero; this is used below in lines 289–290 to indicate the fact that the size of the quantity within the corresponding index register must be checked after simulating these operations.)

The next part of our simulator program merely lists the locations used to contain the contents of the simulated registers:

```
101     AREG    CON   0     Magnitude of A-register
102     I1REG   CON   0     Magnitude of index registers
103     I2REG   CON   0
104     I3REG   CON   0
105     I4REG   CON   0
106     I5REG   CON   0
107     I6REG   CON   0
108     XREG    CON   0     Magnitude of X-register
109     JREG    CON   0     Magnitude of J-register
110     ZERO    CON   0     Constant zero, for "STZ"
111     SIGNA   CON   1     Sign of A-register
112     SIGN1   CON   1     Sign of index registers
113     SIGN2   CON   1
114     SIGN3   CON   1
115     SIGN4   CON   1
116     SIGN5   CON   1
117     SIGN6   CON   1
118     SIGNX   CON   1     Sign of X-register
119     SIGNJ   CON   1     Sign of J-register
120     SIGNZ   CON   1     Sign stored in "STZ"
121     INST    CON   0     Instruction being simulated
122     COMPI   CON   0     Comparison indicator
123     OVTOG   CON   0     Overflow toggle
124     CLOCK   CON   0     Simulated execution time ▮
```

Now we will consider the various subroutines used by the simulator. First comes the MEMORY subroutine:

Calling sequence: `JMP MEMORY`.

Entry conditions: rI5 = valid memory address (otherwise subroutine will jump to MEMERROR).

Exit conditions: rX = sign of word in memory location (rI5); rA = magnitude of word in memory location (rI5).

```
125     * SUBROUTINES
126     MEMORY  STJ   9F          Memory fetch subroutine
127             J5N   MEMERROR
128             CMP5  =BEGIN=     Simulated memory is in
129             JGE   MEMERROR        locations 0000 to BEGIN.
130             LDX   0,5
131             ENTA  1
132             SRAX  5           Sign of word → rX.
133             LDA   0,5(1:5)    Magnitude of word → rA.
134     9H      JMP   *           Exit. ▮
```

The FCHECK subroutine processes a partial field specification, making sure it has the form 8L+R with $L \leq R \leq 5$.

Calling sequence: JMP FCHECK.

Entry conditions: rI3 is valid field specification (otherwise subroutine will jump to FERROR).

Exit conditions: rA = rI1 = L, rX = R.

```
135     FCHECK  STJ     9F      Field check subroutine
136             ENTA    0
137             ENTX    0,3     rAX ← field specification.
138             DIV     =8=     Separate into L and R.
139             CMPX    =5=     Is R > 5?
140             JG      FERROR
141             STX     R
142             STA     L
143             LD1     L       rI1 ← L.
144             CMPA    R
145     9H      JLE     *       Exit unless L > R.
146             JMP     FERROR      ∎
```

The last subroutine, GETV, finds the quantity V (i.e., the appropriate field of location M) used in various MIX operators, as defined in Section 1.3.1.

Calling sequence: JMP GETV.

Entry conditions: rI5 = valid memory address; rI3 = valid field.

Exit conditions: rA = magnitude of V; rX = sign of V; rI1 = L; rI2 = −R.

Second entrance: JMP GETAV, used only in comparison operators to extract a field from a register.

```
147     GETAV   STJ     9F      Special entrance, see line 300.
148             JMP     1F
149     GETV    STJ     9F      Subroutine to find V
150             JMP     FCHECK  Process field; L → rI1.
151             JMP     MEMORY  rA = memory magnitude,
                                    rX = sign.
152     1H      J1Z     2F      Is sign part of the field?
153             ENTX    1       If not, set sign positive.
154             SLA     -1,1    Extract off bytes to left
155             SRA     -1,1        of the field.
156     2H      LD2N    R       Shift right into
157             SRA     5,2         proper position.
158     9H      JMP     *       Exit. ∎
```

Now we come to the routines for the individual operators. These routines are given here for completeness, but the reader should study only a few of them

unless he is exceptionally ambitious; those for SUB and JUMP are recommended as typical examples for study. Note how routines for similar operations are neatly combined, and note how the JUMP routine uses another switching table to govern the type of jump.

```
159     * INDIVIDUAL OPERATORS
160     ADD         JMP   GETV       Get value of V in rA, rX.
161                 ENT1  0          Let rI1 indicate the A register.
162                 JMP   INC        Go to "increase" routine.
163     SUB         JMP   GETV       Get value of V in rA, rX.
164                 ENT1  0          Let rI1 indicate the A register.
165                 JMP   DEC        Go to "decrease" routine.
166     *
167     MUL         JMP   GETV       Get value of V in rA, rX.
168                 CMPX  SIGNA      Are signs the same?
169                 ENTX  1
170                 JE    *+2        Set rX to result sign.
171                 ENNX  1
172                 STX   SIGNA      Put it in both simulated registers.
173                 STX   SIGNX
174                 MUL   AREG       Multiply the operands.
175                 JMP   STOREAX    Store the magnitudes.
176     *
177     DIV         LDA   SIGNA      Set sign of remainder.
178                 STA   SIGNX
179                 JMP   GETV       Get value of V in rA, rX.
180                 CMPX  SIGNA      Are signs the same?
181                 ENTX  1
182                 JE    *+2        Set rX to result sign.
183                 ENNX  1
184                 STX   SIGNA      Put it in simulated rA.
185                 STA   TEMP
186                 LDA   AREG       Divide the operands.
187                 LDX   XREG
188                 DIV   TEMP
189     STOREAX     STA   AREG       Store the magnitudes.
190                 STX   XREG
191     OVCHECK     JNOV  CYCLE      Did overflow just occur?
192                 ENTX  1          If so, set simulated
193                 STX   OVTOG            overflow toggle on.
194                 JMP   CYCLE      Return to control routine.
195     *
196     LOADN       JMP   GETV       Get value of V in rA, rX.
197                 ENT1  47,4       rI1 ← C-16; indicates register.
198     LOADN1      STX   TEMP       Negate sign.
199                 LDXN  TEMP
200                 JMP   LOAD1      Change LOADN to LOAD.
201     LOAD        JMP   GETV       Get value of V in rA, rX.
```

```
202                ENT1   55,4          rI1 ← C-8, indicates register.
203   LOAD1        STA    AREG,1        Store magnitude.
204                STX    SIGNA,1       Store sign.
205                JMP    SIZECHECK     Check if magnitude too large.
206   *
207   STORE        JMP    FCHECK        rI1 ← L.
208                JMP    MEMORY        Get contents of memory location.
209                J1P    1F            Is the sign part of the field?
210                ENT1   1             If so, change L to 1
211                LDX    SIGNA+39,4          and "store" sign of register.
212   1H           LD2N   R             rI2 ← —R.
213                SRAX   5,2           Save area to right of field.
214                LDA    AREG+39,4     Insert register in field.
215                SLAX   5,2
216                ENN2   0,1           rI2 ← —L.
217                SRAX   6,2
218                LDA    0,5           Restore area to left of field.
219                SRA    6,2
220                SRAX   -1,1          Attach the sign.
221                STX    0,5           Store in memory.
222                JMP    CYCLE         Return to control routine.
223   *
224   JUMP         DEC3   9             Jump operators
225                J3P    FERROR        Is F too large?
226                LDA    COMPI         Comparison indicator → rA.
227                JMP    JTABLE,3      Jump to appropriate routine.
228   JMP          ST6    JREG          Set simulated J-register.
229                JMP    JSJ
230                JMP    JOV
231                JMP    JNOV
232                JMP    LS
233                JMP    EQ
234                JMP    GR
235                JMP    GE
236                JMP    NE
237   JTABLE       JMP    LE            Jump table
238   JOV          LDX    OVTOG         Check whether to jump on
239                JMP    *+3                 overflow.
240   JNOV         LDX    OVTOG
241                DECX   1             Get complement of overflow toggle.
242                STZ    OVTOG         Shut off overflow toggle.
243                JXNZ   JMP           Jump.
244                JMP    CYCLE         Don't jump.
245   LE           JAZ    JMP           Jump if rA zero or negative.
246   LS           JAN    JMP           Jump if rA negative.
247                JMP    CYCLE         No jump
248   NE           JAN    JMP           Jump if rA negative or positive.
249   GR           JAP    JMP           Jump if rA positive.
```

```
250                 JMP     CYCLE           No jump
251     GE          JAP     JMP             Jump if rA positive or zero.
252     EQ          JAZ     JMP             Jump if rA zero.
253                 JMP     CYCLE           No jump
254     JSJ         JMP     MEMORY          Check for valid memory address.
255                 ENT6    0,5             Simulate a jump.
256                 JMP     CYCLE           Return to main control routine.
257     *
258     REGJUMP     LDA     AREG+23,4       Register jumps
259                 JAZ     *+2             Is register zero?
260                 LDA     SIGNA+23,4      If not, put sign into rA.
261                 DEC3    5
262                 J3NP    JTABLE,3        Change to a conditional JMP unless
263                 JMP     FERROR              F-specification too large.
264     *
265     ADDROP      DEC3    3               Address transfer operators
266                 J3P     FERROR          Is F too large?
267                 ENTX    0,5
268                 JXNZ    *+2             Find sign of M.
269                 LDX     INST
270                 ENTA    1
271                 SRAX    5               rX = sign of M.
272                 LDA     M(1:5)          rA = magnitude of M.
273                 ENT1    15,4            rI1 indicates the register.
274                 JMP     1F,3            Four-way jump.
275                 JMP     INC             Increase.
276                 JMP     DEC             Decrease.
277                 JMP     LOAD1           Enter.
278     1H          JMP     LOADN1          Enter negative.
279     DEC         STX     TEMP            Reverse sign.
280                 LDXN    TEMP            Change to "increase."
281     INC         CMPX    SIGNA,1         Addition routine
282                 JE      1F              Are signs the same?
283                 SUB     AREG,1          No; subtract magnitudes.
284                 JANP    2F              Sign change in register?
285                 STX     SIGNA,1         Change register sign.
286                 JMP     2F
287     1H          ADD     AREG,1          Add magnitudes.
288     2H          STA     AREG,1(1:5)     Store magnitude of result.
289     SIZECHECK   LD1     OPTABLE,4(3:3)  Have we just loaded an
290                 J1Z     OVCHECK             index register?
291                 CMPA    ZERO(1:3)       If so, make sure result
292                 JE      CYCLE               fits in two bytes.
293                 JMP     SIZEERROR
294     *
295     COMPARE     JMP     GETV            Get value of V in rA, rX.
296                 SRAX    5               Attach sign.
297                 STX     V
```

```
298                 LDA   XREG,4          Get field of appropriate register.
299                 LDX   SIGNX,4
300                 JMP   GETAV
301                 SRAX  5               Attach sign.
302                 CMPX  V               Compare (note that -0 = +0).
303                 STZ   COMPI           Set comparison indicator to
304                 JE    CYCLE                 either zero, plus one,
305                 ENTA  1                     or minus one.
306                 JG    *+2
307                 ENNA  1
308                 STA   COMPI
309                 JMP   CYCLE           Return to control routine
310   *
311                 END   BEGIN           ▮
```

The above code adheres to a rather subtle rule that was stated in Section 1.3.1: the instruction "`ENTA -0`" loads minus zero into register A, as does "`ENTA -5,1`" when index register 1 contains +5. In general, when `M` is zero, `ENTA` loads the sign of the instruction and `ENNA` loads the opposite sign. The need to specify this condition was overlooked in the presentation in Section 1.3.1. Questions like this usually come to light only when a computer program is being written to follow the rules.

In spite of its length, the above program is incomplete in several respects:

a) It does not recognize floating-point operations.
b) The coding for operation codes 5, 6, and 7 has been left as an exercise.
c) The coding for input-output operators has been left as an exercise.
d) No provision has been made for loading simulated programs (see exercise 4).
e) The error routines

`INDEXERROR, ADDRERROR, OPERROR, MEMERROR, FERROR, SIZEERROR`

have not been included; these are for error conditions which are detected in the simulated program.

f) No provision for diagnostic facilities (e.g., printouts of registers as the program is being executed) has been included.

EXERCISES

1. [*14*] Study all the uses of the `FCHECK` subroutine in the simulator program. Can you suggest a better way to organize the subroutines in this program? (Cf. step 3 in the discussion at the end of Section 1.4.1.)

2. [*20*] Write the `SHIFT` routine, which is missing from the program in the text (operation code 6).

▶ **3.** [*22*] Write the `MOVE` routine, which is missing from the program in the text (operation code 7).

4. [*14*] Change the program in the text so that it begins as though MIX's "GO-button" had been pushed (cf. exercise 1.3.1–26).

▸ **5.** [*24*] Determine the time required to simulate the LDA and ENTA operators, compared with the actual time for MIX to execute these operators directly.

6. [*28*] Write programs for the input-output operators JBUS, IOC, IN, OUT, and JRED, which are missing from the program in the text, allowing only units 16 and 18. Assume that a card read or a skip to new page takes $10000u$ and a print takes $7500u$. (*Note:* Experience shows that the JBUS instruction should be simulated by treating "JBUS *" as a special case; otherwise the simulator seems to stop!)

▸ **7.** [*32*] Modify the solutions of the previous exercise in such a way that execution of IN or OUT does not cause I/O transmission immediately; the transmission should take place after approximately half of the time required by the simulated devices has elapsed. (This will prevent a frequent student error of improperly using the IN and OUT operators.)

***1.4.3.2. Trace routines.** When a machine is being simulated on itself (as MIX was simulated on MIX in the previous section) we have the special case of a simulator called a *trace* or *monitor* routine. Such programs are occasionally used to help in debugging, since they print out a step-by-step account of how the simulated program behaves.

The program in the preceding section was written as though another computer were simulating MIX. A quite different approach is used for trace programs; we generally let registers represent themselves and let the operators perform themselves.

In a trace program we usually contrive to let the machine execute most of the instructions; the exception is a jump or conditional jump instruction which must not be executed without modification (for the trace program would lose control). Each machine also has its own idiosyncrasies which make tracing more of a challenge; in MIX's case, this is the J-register.

The trace routine given below is initiated when the main program jumps to location ENTER with register J set to the address for *starting* to trace, register X set to the address where tracing should *stop*. The program is interesting and merits careful study.

```
01      * TRACE ROUTINE
02  ENTER   STX   TEST(0:2)      Set exit location.
03          STX   LEAVEX(0:2)
04          STA   AREG            Save contents of rA.
05          STJ   JREG(0:5)       Save contents of rJ.
06          LDA   JREG            Get start location for trace.
07  CYCLE   STA   PREG(0:2)       Store location of next instruction.
08  TEST    DECA  *               Is it the exit location?
09          JAZ   LEAVE
10  PREG    LDA   *               Get next instruction.
11          STA   INST            Make two copies
12          STA   INST1
```

```
13          LDA   INST(5:5)      Get operation code, C.
14          DECA  38
15          JANN  1F              Is C ≥ 38?
16          INCA  6
17          JANZ  2F              Is C ≠ 32?
18          LDA   INST(0:4)       C = 32 (STJ).
19          STA   *+2(0:4)        Changed to STA.
20          LDA   JREG
21          STA   *
22          JMP   INCP
23    2H    DECA  2
24          JANZ  2F              C ≠ 34?
25          JMP   3F              C = 34 (JBUS).
26    1H    DECA  9               Test for jump instructions.
27          JAP   2F              C ≥ 48?
28    3H    LDA   8F(0:3)         Jump instruction detected;
29          STA   INST(0:3)       its address is changed to "JUMP".
30    2H    LDA   AREG            Restore register A.
31    *                           All registers except J now have proper
32    *                           values with respect to the external program.
33    INST  NOP   *               The instruction is executed.
34          STA   AREG            Store register A again.
35    INCP  LDA   PREG(0:2)       Move to next instruction.
36          INCA  1
37          JMP   CYCLE
38    8H    JSJ   JUMP            Constant for lines 28, 39
39    JUMP  LDA   8B(4:5)         A jump has occurred.
40          SUB   INST(4:5)       Was it JSJ?
41          JAZ   *+4
42          LDA   PREG(0:2)       If not, update simulated
43          INCA  1                     J-register.
44          STA   JREG
45          LDA   INST1(0:3)      Compute indexed address.
46          STA   *+1(0:3)
47          ENTA  *
48          JMP   CYCLE           Move to this instruction.
49    LEAVE LDA   AREG            Restore A-register.
50    LEAVEX JMP  *               Stop tracing.
51    AREG  CON   0               External rA contents
52    JREG  CON   0               External rJ contents
53    INST1 CON   0               Copy of instruction ▮
```

The following things should be noted about trace routines in general and this one in particular:

1) We have presented only the most interesting part of a trace program, the part that retains control while executing another program. For a trace to be useful, there must also be a routine for writing out the contents of registers,

and this has not been included. Such a routine distracts from the more subtle features of a trace program, although it certainly is important; the necessary modifications are left as an exercise (see exercise 2).

2) Space is generally more important than time, i.e., the program should be written to be as short as possible. This is done so the trace routine can coexist with large programs, and the running time is consumed by output anyway.

3) Care was taken to avoid destroying the contents of most registers; in fact, the program uses only MIX's A-register. Neither the comparison indicator nor the overflow toggle are affected by the trace routine. (The less things we use, the less things we need to restore.)

4) When a jump to location JUMP occurs, it is not necessary to "STA AREG", since rA cannot have changed.

5) After leaving the trace routine, the J-register is not reset properly. Exercise 1 shows how to remedy this.

6) The program being traced is subject to only three restrictions: (a) It must not store anything into the locations used by the trace program. (b) It must not use the output device on which tracing information is being recorded (for example, JBUS would give an improper indication). (c) The program is executed at a different rate of speed when tracing.

EXERCISES

1. [*22*] Modify the trace routine of the text so it restores register J when leaving. (You may assume register J is not zero.)

2. [*26*] Modify the trace routine of the text so that before executing each program step it writes the following information on tape unit 0. Use a "buffer swapping" method (see Section 1.4.4) for speed.

Word 1: location.
Word 2: instruction.
Word 3: register A (before execution).
Word 4: register X (before execution).
Words 5–10: registers I1–I6 (before execution).
Word 11: register J.
Word 12: 1 if overflow is on, 0 otherwise.
Word 13: +1 if comparison is greater, 0 if equal, −1 if less.
Words 14–100: not used.

3. [*10*] The previous exercise suggests having the trace program write its output onto tape. Discuss why this would be preferable to printing directly.

▸ **4.** [*25*] What would happen if the trace routine were tracing *itself*? Specifically, consider the behavior if the two instructions

```
ENTX  0
JMP   *+1
```

were placed just before ENTER.

5. [*28*] In a manner similar to that used to solve the previous exercise, consider the situation in which two copies of the trace routine are placed in different places in memory, and each is set up to trace the other. What would happen?

▸ **6.** [*40*] Design a trace routine which is capable of tracing itself, in the sense of exercise 4; i.e., it should print out the steps of its own program at slower speed, and that program will be tracing itself at still slower speed, ad infinitum until memory capacity is exceeded.

1.4.4. Input and Output

Perhaps the most outstanding differences between one computer and the next are the facilities available for doing input and output, and the computer instructions which govern these peripheral devices. We cannot hope to discuss in a single book all of the problems and techniques that arise in this area, so we will confine ourselves to a study of typical input-output methods which apply to most computers. The input-output operators of MIX represent a compromise between the widely varying facilities available in actual machines; to give an example of how to think about input-output, let us discuss in this section the problem of getting the best MIX input-output.

Many computer users feel that input and output are not actually part of "real programming," they are merely things that (unfortunately) must be done in order to get information in and out of the machine. For this reason, the input and output facilities of a computer are usually not learned until after all other features have been examined, and it frequently happens that only a small fraction of the programmers of a particular machine ever know much about the details of input and output. This attitude is present to a certain extent in this book, where we generally give only a light treatment to input-output topics. Hopefully some day another book, covering the principles of executive systems as this book covers other aspects of software systems, will be written, giving a really adequate treatment of input-output control routines.

A brief digression about terminology is perhaps appropriate here. Although contemporary dictionaries seem to regard the words "input" and "output" only as nouns (e.g., "What kind of input are we getting?"), it is now customary to use them grammatically as adjectives (e.g., "Don't drop the input tape.") and as transitive verbs (e.g., "Why did the program output this garbage?"). The combined term "input-output" is most frequently referred to by the abbreviation "I/O". Inputting is often called *reading*, and outputting is, similarly, called *writing*. The stuff that is input or output is generally known as "data"—this word is, strictly speaking, a plural form of the word "datum," but it is used collectively as if it were singular (e.g., "The data has not been read."). This completes today's English lesson.

Suppose now that we wish to read from magnetic tape. The IN operator of MIX, as defined in Section 1.3.1, merely *initiates* the input process, and the computer continues to execute further instructions while the input is taking place. Thus the instruction "IN 1000(5)" will begin to read 100 words from

tape unit number 5 into memory cells 1000–1099, but the ensuing program must not refer to these memory cells until later. The input will be complete only after (a) another I/O operation (IN, OUT, or IOC) referring to unit 5 has been initiated, or (b) the conditional jump instructions JBUS(5) or JRED(5) indicate that unit 5 is no longer "busy."

The simplest way to read a tape record into locations 1000–1099 and to have the information present is therefore the sequence of two instructions

```
IN    1000(5)
JBUS  *(5)                                   (1)
```

We have used this rudimentary method in the program of Section 1.4.2 (see lines 07–08 and 52–53). The method is generally wasteful of computer time, however, because a very large amount of potentially useful calculating time, say $1000u$ or even $10000u$, is consumed by the repeated execution of the "JBUS" instruction. The program's running speed can be as much as doubled if this additional time is utilized for calculation. (See exercises 4 and 5.)

One way to avoid wasting this computation time is to have two areas of memory used for the input; we would read into one area, and while this is going on, the program would compute from the data in the other area. For example, suppose the program begins with the instruction:

```
IN    2000(5)      Begin reading first record.   (2)
```

Subsequently, whenever a tape record is desired we may now give the following five commands:

```
ENT1  1000         Prepare for MOVE operator.
JBUS  *(5)         Wait until unit 5 is ready.
MOVE  2000(50)     (2000–2049) → (1000–1049).    (3)
MOVE  2050(50)     (2050–2099) → (1050–1099).
IN    2000(5)      Begin reading next record.
```

These have the same overall effect as (1).

This program begins to read a tape record into locations 2000–2099 before the preceding record has been examined. This is called "reading ahead" or *anticipated input*—it is done on faith that the record will eventually be needed. In fact, however, we might learn (by examining the contents of the record moved to 1000–1099) that no more input is really required. For example, consider the analogous situation in the coroutine program of Section 1.4.2, where the input was coming from punched cards instead of tape: a "." appearing anywhere in the card meant that it was the final card of the deck. Such a situation would make anticipated input impossible, unless we would assume that either (a) a blank card or special trailer card of some other sort must follow the input deck, or (b) an identifying mark (e.g. ".") must appear in column 80

of the final card of the deck. Some means for properly terminating the input at the end of the program must always be provided whenever input is anticipated.

The technique of overlapping computation time and I/O time is known as *buffering*. The rudimentary method (1) is called "unbuffered" input. The area of memory 2000–2099 used to hold the anticipated input in (3), as well as the area 1000–1099 to which the input was moved, is called a "buffer." Webster's New World Dictionary defines "buffer" as "any person or thing that serves to lessen shock," and the term is appropriate because buffering tends to keep I/O devices running smoothly. (Computer engineers often use the word "buffer" in another sense, to denote a part of the I/O device which stores information during the transmission, but in this book "buffer" will signify an area of *memory* used by a programmer to hold I/O data.)

The sequence (3) is not always superior to (1), although the exceptions are rare. Let us compare the execution times; suppose T is the time required to input 100 words, and suppose C is the computation time which intervenes between input requests. Method (1) requires a time of essentially $T + C$ per tape record, while method (3) takes essentially $\max(C, T) + 202u$. (The quantity $202u$ is the time required by the two `MOVE` instructions.) One way to look at this running time is to consider so-called "critical path time," in this case, the amount of time the I/O unit is idle between uses. Method (1) keeps the unit idle for C units of time, while method (3) keeps it idle for 202 units (assuming $C < T$).

The relatively slow `MOVE` commands of (3) are undesirable, particularly because they take up critical path time when the tape unit must be inactive. An almost obvious improvement of the method allows us to avoid these `MOVE` instructions: the outside program can be revised so that it refers alternately to locations 1000–1099 and 2000–2099; while we are reading into one buffer area, we can be computing with the information in the other. This is the important technique known as *buffer swapping*. The location of the current buffer of interest will be kept in an index register (or, if no index registers are available, in a memory location). We have already seen an example of buffer swapping applied to output in Algorithm 1.3.2P (see steps P9–P11) and the accompanying program.

As an example of buffer swapping on input, suppose that we have a computer application in which each tape record consists of 100 separate one-word items. The following program is a subroutine which gets the next word of input, and which reads in a new record if the current one is exhausted.

```
01    WORDIN  STJ     1F              Store exit location.
02            INC6    1               Advance to next word.
03    2H      LDA     0,6             Is it the end of the
04            CMPA    =SENTINEL=          buffer?
05    1H      JNE     *               If not, exit.                (4)
06            IN      -100,6(U)       Refill this buffer.
07            LD6     1,6             Get address of other
08            JMP     2B                  buffer and return. ∎
```

In this program, index register 6 is used to address the last word of input; we assume that the outside program does not affect this register. The symbol U refers to a tape unit, and the symbol SENTINEL refers to a value which is known (from characteristics of the program) to be *absent* from all tape records. The subroutine is accompanied by the following layout of buffers:

```
09      INBUF1  ORIG    *+100           First buffer
10              CON     SENTINEL        'Sentinel' at end of buffer
11              CON     *+1             Address of other buffer
12      INBUF2  ORIG    *+100           Second buffer
13              CON     SENTINEL        'Sentinel' at end of buffer
14              CON     INBUF1          Address of other buffer ▮
```

Several things about this program should be noted:

1) The "sentinel" constant appears as the 101st word of each buffer, and it makes a convenient test for the end of the buffer. In many applications, however, this technique will not be reliable, since any word may appear on tape. If we were doing card input, a similar technique (with the 17th word of the buffer equal to a sentinel) can always be used; in this case, any negative word can serve as a sentinel, since input from cards always gives nonnegative words.

2) Each buffer contains the address of the other buffer (see lines 07, 11, and 14). This "linking together" facilitates the swapping process.

3) No "JBUS" instruction was necessary, since the next input was initiated before any word of the previous record was accessed. If the quantities C and T refer as before to computation time and tape time, the execution time per tape record is now $\max(C, T)$; it is therefore possible to keep the tape going at full speed if $C < T$. (*Note:* MIX is an idealized computer in this regard, however, since no I/O errors must be treated by the program. On most computers some instructions to test the successful completion of the previous operation would be necessary just before the "IN" instruction here.)

4) To make this subroutine work properly, it will be necessary to get things started out right when the program begins. Details are left to the reader (see exercise 6).

5) The WORDIN subroutine makes the tape unit appear to have a record length of 1 rather than 100 as far as the rest of the program is concerned. The idea of having several program-oriented records filling a single actual tape record is called "blocking of records."

The techniques which we have illustrated for input apply, with minor changes, to output as well (see exercises 2 and 3).

Multiple buffers. Buffer swapping is just the special case $N = 2$ of a general method involving N buffers. In some applications it is desirable to have more than two buffers; for example, consider the following type of algorithm:

Step 1. Read in 5 records in rapid succession.

Step 2. Perform a fairly long calculation based on this data.
Step 3. Return to step 1.

Here five or six buffers would be desirable, so that the next batch of five records could be read during step 2. This tendency for I/O activity to be "bunched" makes multiple buffering an improvement over buffer swapping. Multiple buffering is also desirable for those devices (such as certain UNIVAC card readers) on which the IN instruction initiates the input process but several additional INs (initiating the input process for further records) may be given before the first record is available; then it is necessary to have multiple buffers in order to have any chance of running the input device at a reasonable speed.

Suppose we have N buffers for some input or output process using a single I/O device; we will think of them as if they were arranged in a circle as shown in Fig. 23. So far as this I/O unit is concerned, the program external to the buffering process will be assumed to have the following form:

```
    ⋮
ASSIGN
    ⋮
RELEASE
    ⋮
ASSIGN
    ⋮
RELEASE
    ⋮
```

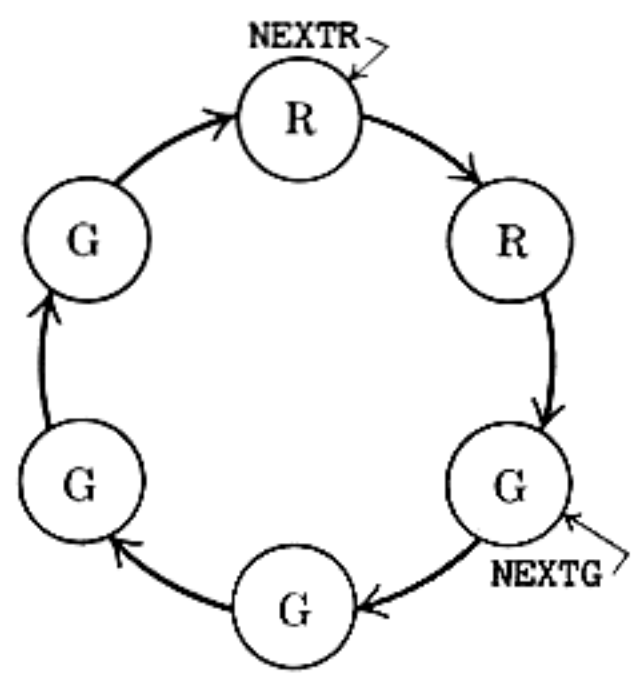


Fig. 23. A circle of buffers ($N = 6$).

i.e., alternate actions of "ASSIGN" and "RELEASE", separated by other computation which does not affect the buffer manipulations for this device.

ASSIGN means that the program acquires the address of the next buffer area, i.e., this address is assigned as the value of some program variable.

RELEASE means the program is done with the current buffer area.

Between ASSIGN and RELEASE the program is communicating with one of the buffers, called the *current* buffer area; between RELEASE and ASSIGN, the program makes no reference to any buffer area.

Conceivably, ASSIGN could immediately follow RELEASE, and discussions of buffering have often been based on this assumption. However, if RELEASE is done as soon as possible, the buffering process has more freedom and will be more effective; by separating the two essentially different functions of ASSIGN and RELEASE we will find the buffering technique is simpler to understand, and our discussion will be meaningful even if $N = 1$.

To be more explicit, let us consider the cases of input and output separately. For input, suppose we are dealing with a card reader. The action ASSIGN means the program needs the information from a new card; we would like to set an index register to the memory address at which the next card image is located. The action RELEASE occurs when the information in the current card image is no longer needed—it has somewhere been digested by the program, perhaps copied to another part of memory, etc. The current buffer area may therefore be filled with further anticipated input.

For output, consider the case of a printer. The action ASSIGN occurs when a free buffer area is needed, into which a line image is to be placed for printing. We wish to set an index register equal to the memory address of such an area. The action RELEASE occurs when this line image has been fully set up in the buffer area, in a form ready to be printed.

Example. To print the contents of locations 0800–0823, we might write

```
JMP    ASSIGNP      (Sets rI5 to buffer location)
ENT1   0,5                                          (5)
MOVE   800(24)      Move 24 words into buffer.
JMP    RELEASEP
```

where ASSIGNP and RELEASEP represent subroutines to do the two buffering functions for the printer.

In an optimal situation (from the standpoint of the computer), the ASSIGN operation will require virtually no execution time. This would mean, on input, that each card image has been anticipated, so it is available when the program is ready for it; and on output, it would mean that there always is a free place in memory to record the line image. No time will be spent waiting for the I/O device.

To help describe the buffering algorithm, and to make it more colorful, we will say buffer areas are either "green," "yellow," or "red" (shown as *G*, *Y*, and *R* in Fig. 24).

Green means that the area is ready to be ASSIGNed; this means it has been filled with anticipated information [in an input situation], or that it is a free area [in an output situation].

Yellow means that the area has been ASSIGNed, not RELEASEd; this means it is the current buffer, and the program is communicating with it.

Red means that the area has been RELEASEd; thus it is a free area (in an input situation) or it has been filled with information (in an output situation).

Figure 23 shows two "pointers" associated with the circle of buffers. These are, conceptually, index registers in the program. NEXTG and NEXTR point to the "next green" and "next red" buffer, respectively. A third pointer, CURRENT (shown in Fig. 24), indicates the yellow buffer when one is present.

Although the algorithm applies equally well to output, we will first consider the case of input from a card reader for definiteness. Suppose a program has

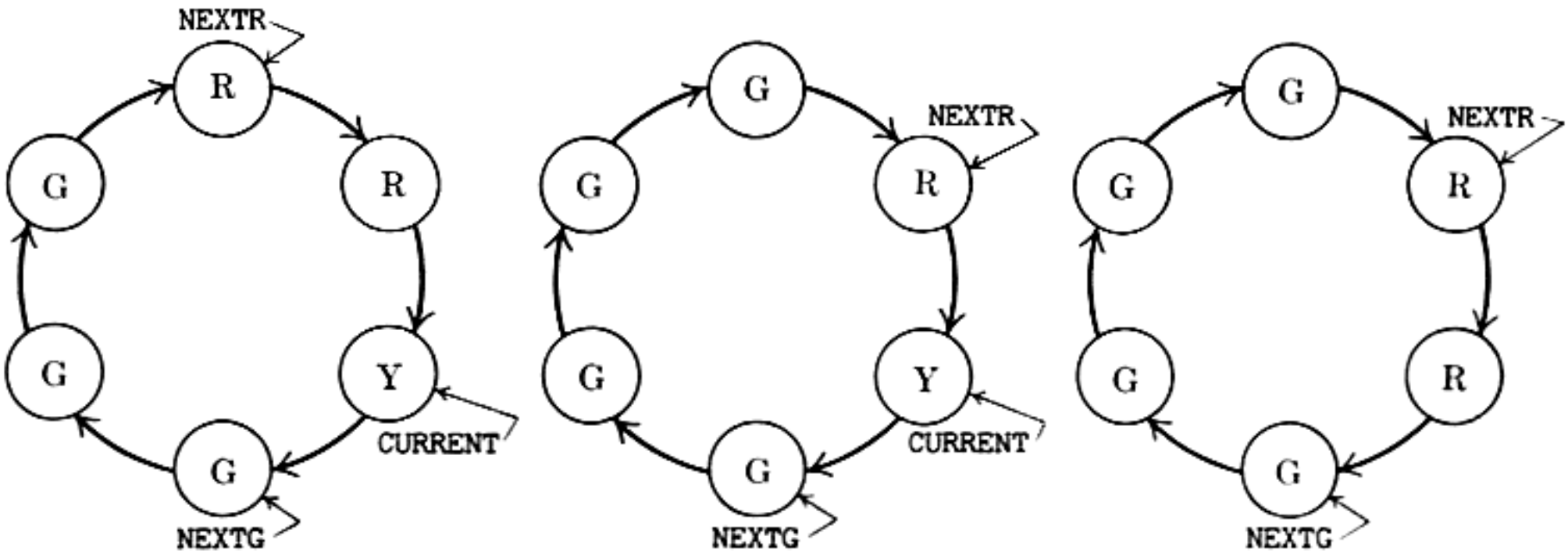


Fig. 24. Buffer transitions, (a) after **ASSIGN**, (b) after I/O complete, and (c) after **RELEASE**.

reached the state shown in Fig. 23. This means that four card images have been anticipated by the buffering process, and they reside in the green buffers. At this moment, two things are happening *simultaneously:* (a) The program is computing, following a **RELEASE** operation; (b) a card is being read into the buffer indicated by **NEXTR**. This state of affairs will continue until the input cycle is completed (the unit will then go from "busy" to "ready"), or until the program does an **ASSIGN** operation. Suppose the latter occurs: then the buffer indicated by **NEXTG** changes to yellow (it is assigned as the current buffer), **NEXTG** moves clockwise, and we arrive at the position shown in Fig. 24(a). If now the input is completed, another anticipated record is present so the buffer changes from red to green and **NEXTR** moves over as shown in Fig. 24(b). If the **RELEASE** operation follows next, we obtain Fig. 24(c).

For an example concerning output, see Fig. 27 on page 223. That illustration shows the "colors" of buffer areas as a function of time, in a program that opens with four quick outputs, then produces four at a slow pace, and finally issues two in rapid succession as the program ends. Three buffers appear in that example.

The pointers **NEXTR** and **NEXTG** proceed merrily around the circle, each at an independent rate of speed, moving clockwise. It is a race between the program (which turns buffers from green to red) and the I/O buffering process (which turns them from red to green). Two situations of conflict can occur:

a) if **NEXTG** tries to pass **NEXTR**, the program has gotten ahead of the I/O device and it must wait until the device is ready.
b) if **NEXTR** tries to pass **NEXTG**, the I/O device has gotten ahead of the program and we must shut it down until the next **RELEASE** is given.

Both of these situations are depicted in Fig. 27. (See exercise 9.)

Fortunately, in spite of the rather lengthy explanation just given of the ideas behind a circle of buffers, the actual algorithms for handling the situation are

very simple. In the following description,

$$\begin{aligned} N &= \text{total number of buffers;} \\ n &= \text{current number of red buffers.} \end{aligned} \tag{6}$$

The variable n is used in the algorithm below to avoid interference between NEXTG and NEXTR.

Algorithm A (ASSIGN *action*). This algorithm includes the steps implied by ASSIGN within a computational program, as described above.

A1. [Wait for $n < N$.] If $n = N$, stall the program until $n < N$. (If $n = N$, no buffers are ready to be assigned; but Algorithm B below, which runs in parallel to this one, will eventually succeed in producing a green buffer.)

A2. [CURRENT ← NEXTG.] Set CURRENT ← NEXTG (thereby assigning the current buffer).

A3. [Advance NEXTG.] Advance NEXTG to the next clockwise buffer. ∎

Algorithm R (RELEASE *action*). This algorithm includes the steps implied by RELEASE within a computational program, as described above.

R1. [Increase n.] Increase n by one. ∎

Algorithm B (*Buffer control*). This algorithm performs the actual initiation of I/O operators in the machine; it is to be executed "simultaneously" with the main program, in the sense described below.

B1. [Compute.] Let the main program compute for a short period of time; step B2 will be executed after a certain time delay, at a time when the I/O device is ready for another operation.

B2. [$n = 0$?] If $n = 0$, go to B1. (Thus, if no buffers are red, no I/O action can be performed.)

B3. [Initiate I/O.] Initiate transmission between the buffer area designated by NEXTR and the I/O device.

B4. [Compute.] Let the main program run for a period of time; then go to step B5 when the I/O operation is completed.

B5. [Advance NEXTR.] Advance NEXTR to the next clockwise buffer.

B6. [Decrease n.] Decrease n by one, and go to B2. ∎

In these algorithms, we have two independent processes which are going on "simultaneously": the buffering control program and the computation program. These are, in fact, *coroutines*, which we will call CONTROL and COMPUTE. Coroutine CONTROL jumps to COMPUTE in steps B1 and B4; coroutine COMPUTE jumps to CONTROL by interspersing "jump ready" instructions at sporadic intervals in its program.

Coding this algorithm for MIX is extremely simple. For convenience, assume that the buffers are linked so that the word *preceding* each one is the address of the next; i.e., if $N = 3$,

$$\texttt{CONTENTS(BUF1-1)} = \texttt{BUF2},$$
$$\texttt{CONTENTS(BUF2-1)} = \texttt{BUF3}, \quad \text{and} \quad \texttt{CONTENTS(BUF3-1)} = \texttt{BUF1}.$$

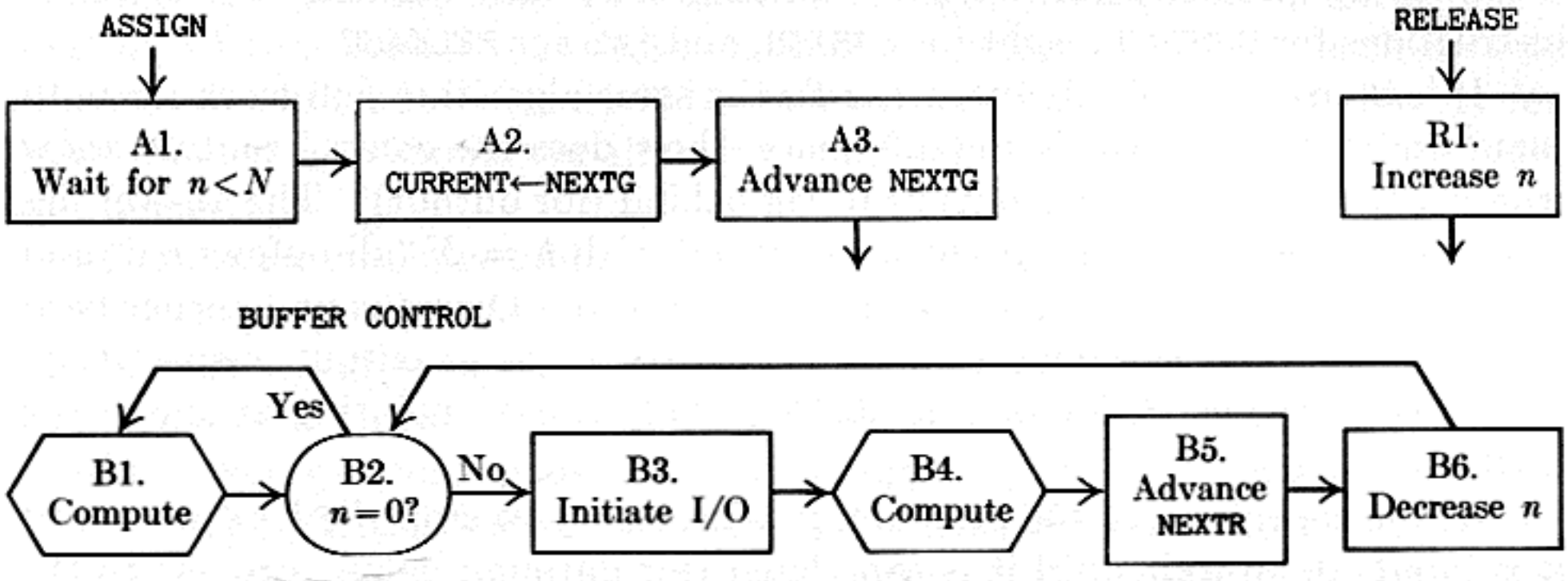


Fig. 25. Algorithms for multiple buffering.

Program A (ASSIGN, *a subroutine within the* COMPUTE *coroutine*). rI4 ≡ CURRENT; rI6 ≡ n; calling sequence is JMP ASSIGN; on exit, rX contains NEXTG.

```
ASSIGN  STJ   9F              Subroutine linkage
1H      JRED  CONTROL(U)      A1. Wait for n < N.
        CMP6  =N=
        JE    1B
        LD4   NEXTG           A2. CURRENT ← NEXTG.
        LDX   -1,4            A3. Advance NEXTG.
        STX   NEXTG
9H      JMP   *               Exit. ∎
```

Program R (RELEASE, *code used within the* COMPUTE *coroutine*). rI6 ≡ n. This short code is to be inserted wherever RELEASE is desired.

```
        INC6  1               R1. Increase n.
        JRED  CONTROL(U)      Possible jump to CONTROL coroutine ∎
```

Program B (*The* CONTROL *coroutine*). rI6 ≡ n, rI5 ≡ NEXTR.

```
CONT1   JMP   COMPUTE         B1. Compute.
1H      J6Z   *-1             B2. n = 0?
        IN    0,5(U)          B3. Initiate I/O.
        JMP   COMPUTE         B4. Compute.
        LD5   -1,5            B5. Advance NEXTR.
        DEC6  1               B6. Decrease n.
        JMP   1B                    ∎
```

Besides the above code, we also have the usual coroutine linkage

```
CONTROL   STJ  COMPUTEX        COMPUTE   STJ  CONTROLX
CONTROLX  JMP  CONT1          COMPUTEX  JMP  COMP1
```

and the instruction "`JRED CONTROL(U)`" is to be placed within `COMPUTE` about once in every fifty instructions.

Thus the programs for multiple buffering essentially amount to only seven instructions for `CONTROL`, eight for `ASSIGN`, and two for `RELEASE`.

It is perhaps remarkable that *exactly* the same algorithm will work for both input and output. What is the difference—how does the control routine know whether to anticipate (for input) or to lag behind (for output)? The answer lies in the initial conditions: for input we start out with $n = N$ (all buffers red) and for output we start out with $n = 0$ (all buffers green). Once the process has been started properly, it continues to behave as either input or output, respectively. The other initial condition is that `NEXTR` = `NEXTG`, both pointing at one of the buffers.

At the conclusion of the program, it is necessary to stop the I/O process (if it is input) or to wait until it is completed (for output); details are left to the reader (see exercises 12 and 13).

It is important to ask what is the best value of N to use. Certainly as N gets larger, the speed of the program will not decrease, but it will not increase indefinitely either and so we come to a point of diminishing returns. Let us refer again to the quantities C and T, representing computation time between I/O operators and the I/O time itself. More precisely, let C be the amount of time between successive `ASSIGN`s, and let T be the amount of time needed to transmit one record. If C is always *greater* than T, then $N = 2$ is adequate, for it is not hard to see that with two buffers we keep the computer busy at all times. If C is always *less* than T, then again $N = 2$ is adequate, for we keep the I/O device busy at all times. Larger values of N are therefore useful only when C varies between small values and large values; one plus the average number of consecutive small values may be right for N, if the large values of C are significantly longer than T. (However, the advantage of buffering is virtually nullified if all input occurs at the beginning of the program and if all output occurs at the end.) If the time between `ASSIGN` and `RELEASE` is always quite small, the value of N may be decreased by 1 throughout the above discussion, with little effect on running time.

The above approach to buffering can be adapted in many ways, and we will mention a few of these briefly. So far we have assumed only one I/O device was being used; in practice, of course, several will be in use at the same time.

There are several ways to approach the subject of multiple units. In the simplest case, we can have a separate circle of buffers for each device. There will be values of n, N, `NEXTR`, `NEXTG`, and `CURRENT`, and a different `CONTROL` coroutine for each unit. This will give efficient buffering action simultaneously on each I/O device.

It is also possible to "pool" buffer areas which are of the same size, i.e., to have two or more devices sharing buffers from a common list. This would be handled by using the linked memory techniques of Chapter 2: all red input buffers and green output buffers would be linked together. It becomes necessary to distinguish between input and output in this case, and to rewrite the algorithms without using n amd N. The algorithm may get irrevocably stuck if all buffers in the pool are filled with anticipated input, so a check should be made that at all times there is at least one buffer (preferably one for each device) which is not input-green; only if the `COMPUTE` routine is stalled at step A1 for some input device should we allow input into the final buffer of the pool from this device.

Some machines have additional constraints on the use of input-output units, so that it is impossible to be transmitting data from certain pairs of devices at the same time. (For example, several units might be attached to the computer by means of a single "channel.") This constraint also affects our buffering routine; when we must choose which I/O unit to initiate first, how is the choice to be made? This is called "forecasting." The best forecasting rule for the general case would seem to give preference to the unit whose buffer circle has the largest value of n/N, assuming the number of buffers in the circles has been wisely chosen.

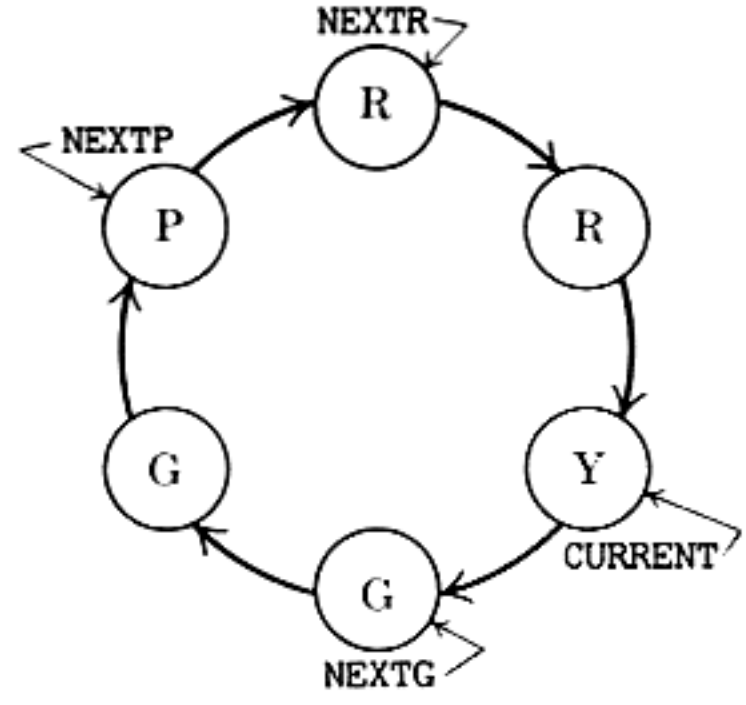


Fig. 26. Input and output from the same circle.

To conclude this discussion, we will mention a useful method for doing input and output from the same buffer circle, under certain conditions. In Fig. 26 we have added another color of buffer (purple). In this situation, green buffers represent anticipated *input;* the program `ASSIGN`s and a green buffer becomes yellow, then upon `RELEASE` it turns red and represents a record to be *output.* The input and output processes follow around the circle independently as before, except now we turn red buffers to purple after the output is done, and convert purple to green on input. It is necessary to ensure that none of the pointers `NEXTG`, `NEXTR`, `NEXTP` passes another. At the instant shown in Fig. 26, the program is computing between `ASSIGN` and `RELEASE`, using the yellow buffer; simultaneously, input is going into the buffer indicated by `NEXTP`; and output is coming from the buffer indicated by `NEXTR`.

EXERCISES

1. [*05*] Would sequence (3) still be correct if the MOVE instructions were placed before the JBUS instruction instead of after it? What if the MOVE instructions were placed after the IN command?

2. [*10*] The instructions

```
OUT    1000(6)
JBUS   *(6)
```

may be used to output a tape record in an unbuffered fashion, just as the instructions (1) did this for input. Give a method analogous to (2) and (3) which buffers this output, by using MOVE instructions and an auxiliary buffer in locations 2000–2099.

▸ **3.** [*22*] Write a buffer-swapping output subroutine analogous to (4). The subroutine, called WORDOUT, should store the word in rA as the next word of output, and if a buffer is full it should write 100 words onto tape unit V. Index register 5 should be used to refer to the current buffer position. Before storing any words into a buffer, it should be cleared to zeros. Show the layout of buffer areas and explain what instructions (if any) are necessary at the beginning and end of the program to ensure that the first and last records are properly written.

4. [*M20*] Show that if a program refers to a single I/O device, it is possible to double the running speed by buffering the I/O, in favorable circumstances, but it is not possible to improve the running speed over the amount of time taken by unbuffered I/O by more than a factor of two.

▸ **5.** [*M21*] Generalize the situation of the preceding exercise to the case when the program refers to n I/O devices instead of just one.

6. [*12*] What instructions should be placed at the beginning of a program so that the WORDIN subroutine (4) gets off to the right start? (For example, index register 6 must be set to *something*.)

7. [*22*] Write a subroutine called WORDIN which is essentially like (4) except that it does not make use of a "sentinel."

8. [*11*] The text describes a hypothetical input situation which leads from Fig. 23 through parts (a), (b), and (c) of Fig. 24. Interpret the same situation given that output to the printer is being done, instead of input from cards. (For example, what things are happening at the time shown in Fig. 23?)

▸ **9.** [*21*] A program which leads to the buffer contents shown in Fig. 27 may be characterized by the following list of times:

A, 1000, R, 1000, A, 1000, R, 1000, A, 1000, R, 1000, A, 1000, R, 1000,
A, 7000, R, 5000, A, 7000, R, 5000, A, 7000, R, 5000, A, 7000, R, 5000,
A, 1000, R, 1000, A, 2000, R, 1000.

This list means "assign, compute for 1000u, release, compute for 1000u, assign, . . . , compute for 2000u, release, compute for 1000u." The computation times given do not include any intervals during which the computer might have to wait for the output device to catch up (as at the fourth "assign" in Fig. 27). The output device operates at a speed of 7500u per record.

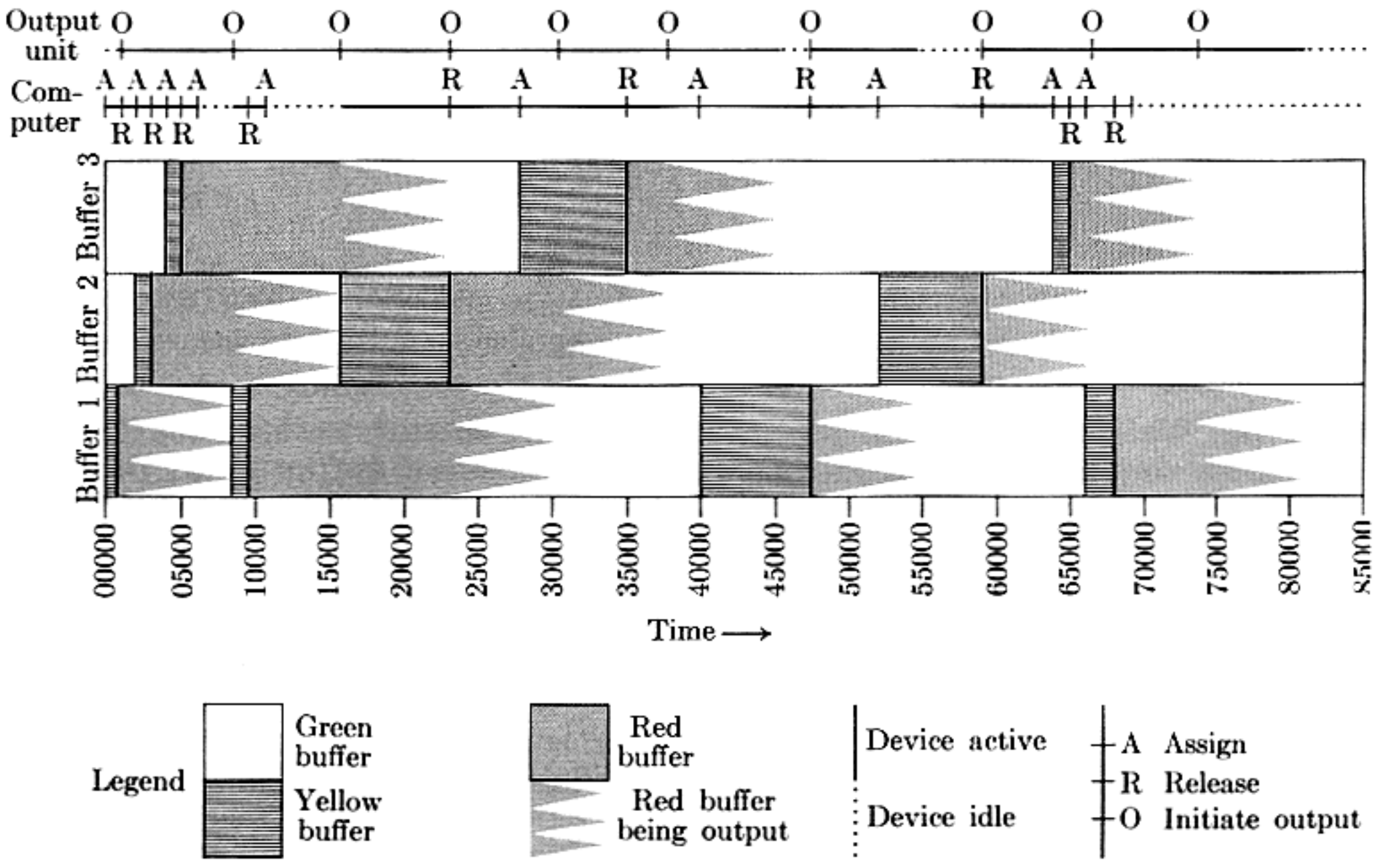


Fig. 27. Output with three buffers (cf. exercise 9).

The following chart specifies the actions shown in Fig. 27 as time passes:

Time	Action	Time	Action
0	`ASSIGN(BUF1)`	38500	`OUT BUF3`
1000	`RELEASE, OUT BUF1`	40000	`ASSIGN(BUF1)`
2000	`ASSIGN(BUF2)`	46000	Output stops.
3000	`RELEASE`	47000	`RELEASE, OUT BUF1`
4000	`ASSIGN(BUF3)`	52000	`ASSIGN(BUF2)`
5000	`RELEASE`	54500	Output stops.
6000	`ASSIGN` (wait)	59000	`RELEASE, OUT BUF2`
8500	`BUF1` assigned, `OUT BUF2`	64000	`ASSIGN(BUF3)`
9500	`RELEASE`	65000	`RELEASE`
10500	`ASSIGN` (wait)	66000	`ASSIGN(BUF1)`
16000	`BUF2` assigned, `OUT BUF3`	66500	`OUT BUF3`
23000	`RELEASE`	68000	`RELEASE`
23500	`OUT BUF1`	69000	Computation stops.
28000	`ASSIGN(BUF3)`	74000	`OUT BUF1`
31000	`OUT BUF2`	81500	Output stops.
35000	`RELEASE`		

The total time required was therefore 81500u; the computer was idle from 6000–8500, 10500–16000, and 69000–81500, or 20500u altogether; the output unit was idle from 0–1000, 46000–47000, and 54500–59000, or 6500u.

Make a "time-action" chart like the above for the same program, assuming there are only *two* buffers.

10. [*21*] Repeat exercise 9, except with *four* buffers.

11. [*21*] Repeat exercise 9, except with just *one* buffer.

12. [*24*] Suppose that the multiple buffering algorithm in the text is being used for card input, and suppose the input is to terminate as soon as a card with "." in column 80 has been read. Show how the CONTROL coroutine (i.e., Algorithm B and Program B) should be changed so that input is shut off in this way.

13. [*20*] What instructions should be included at the end of the COMPUTE coroutine in the text, if the buffering algorithms are being applied to output, to ensure that all information has been output from the buffers?

▸ **14.** [*20*] What if the computational program does not alternate between ASSIGN and RELEASE, but instead gives the sequence of actions . . . ASSIGN . . . ASSIGN . . . RELEASE . . . RELEASE. What effect does this have on the algorithms described in the text? Is it possibly useful?

▸ **15.** [*22*] Write a program that copies 100 records from tape unit 0 to tape unit 1, using just three buffers. The program should be as fast as possible.

16. [*29*] Formulate the "green-yellow-red-purple" algorithm suggested by Fig. 26, in the manner of the algorithms for multiple buffering given in the text, using three coroutines (one to control the input device, one for the output device, and the computation coroutine).

17. [*40*] Adapt the multiple-buffer algorithm to pooled buffers; build in methods which keep the process from slowing down, due to too much anticipated input. Try to make the algorithm as elegant as possible. Compare your method to nonpooling methods, applied to real-life problems.

▸ **18.** [*30*] A modification of MIX is planned which introduces "interrupt capability." This would be done as explained below; the exercise is to modify Algorithms and Programs A, R, and B of the text so that they use these interrupt facilities instead of the "JRED" instructions.

The new MIX features include an additional 3999 memory cells, locations −3999 through −0001. The machine has two internal "states," *normal state* and *control state*. In normal state, locations −3999 through −0001 are not admissible memory locations and the MIX computer behaves as usual. When an "interrupt" occurs, due to conditions explained later, locations −0009 through −0001 are set equal to the contents of MIX's registers: rA in −0009; rX in −0008; rI1 through rI6 in −0007 through −0002; and rJ, the overflow toggle, the comparison indicator, and the location of the next instruction all are stored in −0001 as

+	next inst.	OV, CI	rJ

;

control state is entered, and the machine jumps to a location depending on the type of interrupt.

Location −0010 acts as a "clock": every 1000*u* of time, the number appearing in this location is decreased by one, and if the result is zero an interrupt to location −0011 occurs.

The new MIX instruction "INT" (C = 5, F = 7) works as follows: (a) In normal state, an interrupt occurs to location —0012. (Thus a programmer may force an interrupt, to communicate with a control routine; the address of INT has no effect, although the control routine may use it for information to distinguish between types of interrupt.) (b) In control state, all MIX registers are loaded from locations —0009 to —0001, the computer goes into normal state, and it resumes execution. The execution time for INT is $2u$ in each case.

An IN, OUT, or IOC instruction given in *control* state will cause an interrupt to occur as soon as the I/O operation is completed. The interrupt goes to location —(0020 + unit number).

No interrupts occur while in control state; any interrupt conditions are "saved" until after the next INT operation, and interrupt will occur after one instruction of the normal state program has been performed.

▸ **19.** [*37*] Some computers do not have the ability to perform input-output simultaneously with computation; the I/O operators cause the computer to wait until transmission is complete. These computers have no equivalent of MIX's JBUS or JRED operators, since the units are always "ready"; and there is no interrupt capability as given in the previous exercise. However, there is sometimes the ability to do I/O operations on two different units at once, by giving an "IN–IN" or "IN–OUT" or "OUT–OUT" instruction that causes *two* operations to occur (and the computer waits until *both* are finished).

Multiple-buffering techniques can be used to advantage in this situation, in order to double up I/O operations as frequently as possible. For example, if an output is to be done, anticipated input could simultaneously be read into a buffer.

Develop algorithms for this situation, assuming that a program uses two input tapes and one output tape. There should be three circles of buffers, with N_1 buffers in the first circle, N_2 in the second, and N_3 in the third. Give algorithms for assigning and releasing on each unit, which are as similar to Algorithms A, R, and B of this section as possible. Test your algorithms using computer simulation.

1.4.5. History and Bibliography

Most of the fundamental techniques described in Section 1.4 have been independently developed by a number of different people, and the exact history of the ideas will probably never be known. An attempt has been made to record here the most important contributions to the history.

Subroutines were the first labor-saving devices invented for programmers. Perhaps they were envisioned already in the 19th century by Charles Babbage in connection with his Analytical Engine (but only to the extent that the idea would apply to the punched-card control of the Jacquard loom, since that served as the control of his machine). A subroutine for computing $\sin x$ was written in August, 1944, for the Mark I calculator under the direction of Howard Aiken. This machine was controlled by programs on external paper tapes, instead of programs stored in the memory. H. H. Goldstine and J. von Neumann gave detailed methods for relocating machine instructions and doing subroutine linkage in 1946; see von Neumann, *Collected Works*, Vol. 5, pp. 215–235. In

England, A. M. Turing independently designed computer hardware to facilitate subroutine linkage. [See *Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery* (Cambridge, Mass.: Harvard University, 1949), 83–90.] The use and construction of a very versatile subroutine library is the principal topic of the first computer programming text, *The Preparation of Programs for an Electronic Digital Computer*, by M. V. Wilkes, D. J. Wheeler, and S. Gill, 1st ed. (Reading, Mass.: Addison-Wesley, 1951).

The word "coroutine" was coined by M. E. Conway in 1958, after he had developed the concept, and he first applied it to the construction of an assembly program. Coroutines were independently studied by J. Erdwinn and J. Merner, at about the same time; they wrote a paper entitled "Bilateral linkage," which was not then considered sufficiently interesting to merit publication, and unfortunately no copies of this paper seem to exist today. The first published explanation of the coroutine concept appeared much later in Conway's article "Design of a Separable Transition-Diagram Compiler," *CACM* **6** (1963), 396–408.

The first interpretive routine may be said to be the "Universal Turing Machine," a Turing machine capable of simulating any other Turing machines (see Chapter 11). These are not actual machines, they are theoretical tools used in proving some problems "unsolvable." Interpretive routines in the conventional sense were spontaneously developed slightly before 1950 at several places, chiefly to provide a convenient means of doing floating-point arithmetic. Turing took a part in this development also; interpretive systems for the Pilot ACE computer were written under his direction. The most notable early interpreters were certain routines for the Whirlwind I (by C. W. Adams and others) and for the Illiac (by D. J. Wheeler and others). For references to the state of interpreters in the early fifties, see the article "Interpretative Sub-routines," by J. M. Bennett, D. G. Prinz, and M. L. Woods, *Proc. ACM Nat. Conf.* (1952), 81–87; see also various papers in the *Proceedings of the Symposium on Automatic Programming for Digital Computers* (1954), published by the Office of Naval Research, Washington, D.C.

The most extensively used early interpretive system was probably John Backus's "IBM 701 Speedcoding system" [see *JACM* **1** (1954), 4–6]. This system was slightly modified and skillfully written for the IBM 650 by V. M. Wolontis and others of the Bell Telephone Laboratories; their routine, called the "Bell Interpretive System," was extremely popular. The IPL interpretive systems, designed in 1956 by A. Newell and H. Simon for applications to quite different problems (see Section 2.6), have also seen extensive use as a programming tool. Modern uses of interpreters, as mentioned in the introduction to Section 1.4.3, are often mentioned in passing in the computer literature; see the references listed in that section for articles which discuss interpretive routines in somewhat more detail.

The first tracing routine was developed by Stanley Gill in 1950; see his interesting article in *Proceedings of the Royal Society of London*, series A, **206** (May, 1951), 538–554. The text by Wilkes, Wheeler, and Gill mentioned above

includes listings of several trace routines. Perhaps the most interesting of these is subroutine C-10 by D. J. Wheeler, which includes a provision for suppressing the trace upon entry to a library subroutine, executing the subroutine at full speed, then continuing the trace. Published information about trace routines is quite rare in the general computer literature, primarily because the methods are inherently oriented to a particular machine. The only other reference known to the author is H. V. Meek, "An Experimental Monitoring Routine for the IBM 705," *Proc. Western Joint Computer Conf.* (1956), 68–70, which discusses a trace routine for a machine on which the problem is particularly difficult.

The development of buffering techniques waited until computers became capable of doing simultaneous operations and suitably controlling the interactions. Buffering was first done by computer hardware, in a manner analogous to the code 1.4.4(3), where an internal "buffer area" inaccessible to the programmer plays the role of locations 2000–2099, and where the sequence 1.4.4(3) was performed when an input command was given. For a good survey of the prevailing philosophy towards I/O in 1952, see the Proceedings of the Eastern Joint Computer Conference held in that year.

The real invention of buffering occurred in the minds of computer designers who first realized that they could let the I/O device communicate with core memory at the same time the program is running, and who built in suitable instructions to make efficient programmatic buffering possible. (The author attempted to locate patent numbers of these inventions, but found the search too formidable.) The first published reference to buffering techniques in the sense we have described gives a highly sophisticated approach; see O. Mock and C. J. Swift, "Programmed Input-Output Buffering," *Proc. ACM Nat. Conf.* (1958) paper 19, and *JACM* **6** (1959), 145–151. (The reader is cautioned that these articles contain a good deal of local jargon which may take some time to understand, but neighboring articles in *JACM* **6** will help.) An interrupt system which enabled buffering of input and output was independently developed by E. W. Dijkstra of the Netherlands in 1957 and 1958. His doctoral thesis, "Communication with an Automatic Computer" (1958, now out of print) mentions buffering techniques, which in this case involved very long circles of buffers since the routines were primarily concerned with paper tape and typewriter I/O; each buffer contained either a single character or a single number. The paper "Input-Output Buffering and FORTRAN," by David E. Ferguson, *JACM* **7** (1960), 1–9, describes buffer circles and gives a detailed description of simple buffering with many units at once.

CHAPTER TWO

INFORMATION STRUCTURES

I think that I shall never see
A poem lovely as a tree.

—JOYCE KILMER (1913)

Yea, from the table of my memory
I'll wipe away all trivial fond records.

— Hamlet (Act I, Sc. 5, Line 98)

2.1. INTRODUCTION

Computer programs usually operate on tables of information. In most cases these tables are not simply amorphous masses of numerical values; they involve important *structural relationships* between the data elements.

In its simplest form, a table might be a linear list of elements, when its relevant structural properties include the answers to such questions as: which element is first in the list? which is last? which elements precede and follow a given one? There is a lot to be said about structure even in this apparently simple case (see Section 2.2).

In more complicated situations, the table might be a two-dimensional array (i.e., a matrix or grid, having both a row and a column structure), or it might be an n-dimensional array for higher values of n; it might be a tree structure, representing hierarchical or branching relationships; or it might be a complex multilinked structure with a great many interconnections, such as we may find in a human brain.

In order to use a computer properly, it is important to acquire a good understanding of the structural relationships present within data, and of the techniques for representing and manipulating such structure within a computer.

The present chapter summarizes the most important facts about information structures: the static and dynamic properties of different kinds of structure; means for storage allocation and representation of structured data; and efficient algorithms for creating, altering, accessing, and destroying structural information. In the course of this study, we will also work out several important examples which illustrate the application of these methods to a wide variety of problems. The examples include topological sorting, polynomial arithmetic,

discrete system simulation, operations on sparse matrices, algebraic formula manipulation, and applications to the writing of compilers and operating systems. Our concern will be almost entirely with structure as represented *inside* a computer; the conversion from external to internal representations is the subject of Chapters 9 and 10.

Much of the material we will discuss is often called "List processing," since a number of programming systems (e.g., IPL-V, LISP, and SLIP) have been designed to facilitate working with certain general kinds of structures called *Lists*. (When the word "list" is capitalized in this chapter, it is being used in a technical sense to denote a particular type of structure that is studied in detail in Section 2.3.5.) Although List-processing systems are useful in a large number of situations, they impose constraints on the programmer that are often unnecessary; it is usually better to use the methods of this chapter directly in one's own programs, tailoring the data format and the processing algorithms to the particular application. Too many people unfortunately still feel that List-processing techniques are quite complicated (so that it is necessary to use someone else's carefully written interpretive system or set of subroutines), and that List-processing must be done only in a certain fixed way. We will see that there is nothing magic, mysterious, or difficult about the methods for dealing with complex structures; these techniques are an important part of every programmer's repertoire, and he can use them easily whether he is writing a program in assembly language or in a compiler language like FORTRAN or ALGOL.

We will illustrate methods of dealing with information structures in terms of the MIX computer. A reader who does not care to look through detailed MIX programs should at least study the ways in which structural information is represented in MIX's memory.

It is important to define at this point several terms and notations which we will be using frequently from now on. The information in a table consists of a set of *nodes* (called "records," "entities," or "beads" by some authors); we will occasionally say "item" or "element" instead of "node." Each node consists of one or more consecutive words of the computer memory, divided into named parts called *fields*. In the simplest case, a node is just one word of memory, and it has just one field comprising that whole word. As a more interesting example, suppose the elements of our table are intended to represent playing cards; we might have two-word nodes broken into five fields, TAG, SUIT, RANK, NEXT, and TITLE:

+	TAG	SUIT	RANK	NEXT
+	TITLE			

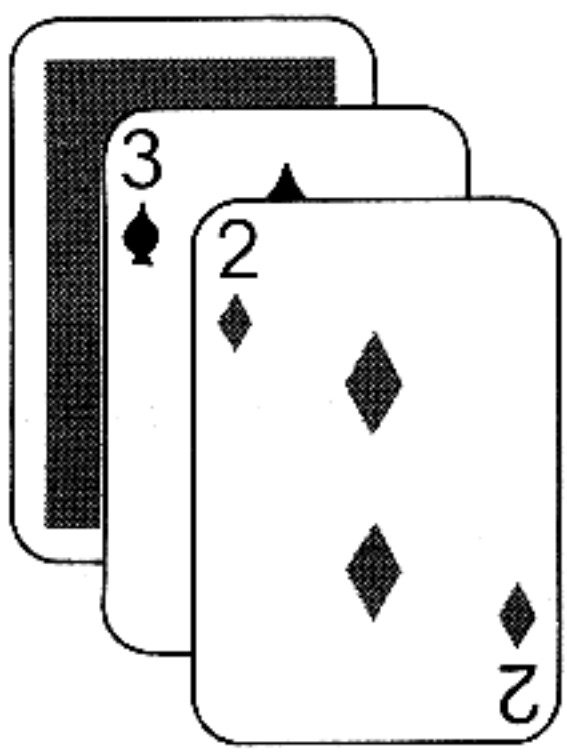
(1)

(This format reflects the contents of two MIX words. Recall that a MIX word consists of five bytes plus a sign; see Section 1.3.1. In this example we assume the signs are + in each word.) The *address* of a node, also called a *link*, *pointer*, or *reference* to that node, is the memory location of its first word. The address

is often taken relative to some "base" location, but in this chapter for simplicity we will take the address to be an absolute memory location.

The contents of any field within a node may represent numbers, alphabetic characters, links, or anything else the programmer may desire. In connection with the example above, let us suppose we wish to represent a pile of cards that might appear in a game of solitaire: TAG = 1 means the card is face down, TAG = 0 means it is face up; SUIT = 1, 2, 3, or 4 for clubs, diamonds, hearts, or spades, respectively; RANK = 1, 2, . . . , 13 for ace, deuce, . . . , king; NEXT is a *link* to the card *below* this one in the pile; and TITLE is the five-character alphabetic name of this card, for use in printouts. A typical pile might look like this:

Actual cards



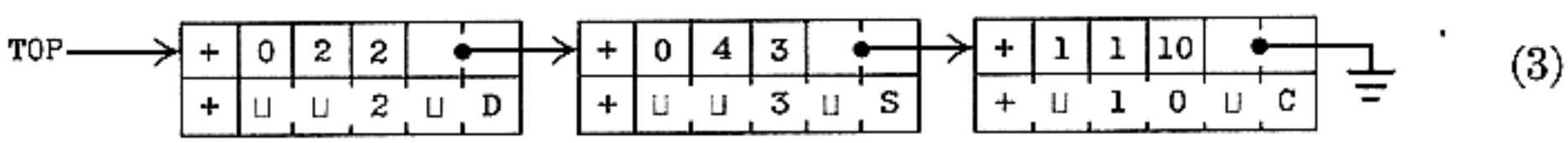
Computer representation

100:	+	1	1	10	Λ	
101:	+	␣	1	0	␣	C
386:	+	0	4	3	100	
387:	+	␣	␣	3	␣	S
242:	+	0	2	2	386	
243:	+	␣	␣	2	␣	D

(2)

The memory locations in the computer representation are shown here as 100, 386, and 242; these could have been any other numbers as far as this example is concerned, since each card links to the next. Note the special link "Λ" in the node 100; *we use the Greek letter lambda to denote the null link*, i.e., the link to no node. The null link Λ appears in node 100 since the 10 of clubs is the bottom card of the pile. Within the machine, Λ is represented by some easily recognizable value which cannot be the address of a node. We will generally assume no node appears in location 0, and, consequently, Λ will almost always be represented as the link value 0 in MIX programs.

The introduction of links to other elements of data is an extremely important idea in computer programming; this is the key to the representation of complex structures. When displaying computer representations of nodes it is usually convenient to represent links by arrows, so that our above example would appear thus:



(3)

The actual locations 242, 386, and 100 (which are irrelevant anyway) no longer

appear in the representation (3). A null link can be shown as "grounded" in electrical circuit notation. We have added "TOP" in (3); this stands for a *link variable*, often called a pointer variable, i.e., a variable within the computer program whose value is a link. All references to nodes in a program are made directly through link variables (or link constants), or indirectly through link fields in other nodes.

Now we come to the most important part of the notation, the means of referring to fields within nodes. This is done simply by giving the name of the field followed by a link to the desired node in parentheses; for example in (1), (2), (3) we have

$$\texttt{TITLE(TOP)} = \text{" }\sqcup\sqcup\texttt{2}\sqcup\texttt{D"};\qquad \texttt{SUIT(TOP)} = 2;\qquad \texttt{RANK(100)} = 10;$$
$$\texttt{RANK(NEXT(TOP))} = 3. \tag{4}$$

The reader should study these examples carefully, since such field notations will be used in many algorithms of this chapter and the following chapters. To make the ideas clearer, we will now state a simple algorithm for placing a new card on top of the pile, assuming NEWCARD is a link variable whose value is a link to the new card:

A1. Set NEXT(NEWCARD) ← TOP. (This sets the appropriate link in the new card node.)

A2. Set TOP ← NEWCARD. (This keeps TOP pointing to the top of the pile.)

A3. Set TAG(TOP) ← 0. (This marks the card as "face up.") ▮

Another example is the following algorithm, which counts the number of cards currently in the pile:

B1. Set N ← 0, X ← TOP. (N is an integer variable, X is a link variable.)

B2. If X = Λ, stop; N is the number of cards in the pile.

B3. Set N ← N + 1, X ← NEXT(X), and go back to step B2. ▮

Note that we use symbolic names for two quite different things in these algorithms: as names of *variables* (TOP, NEWCARD, N, X) and as names of *fields* (TAG, NEXT). These quantities must not be confused. If F is a field name and L ≠ Λ is a link, then F(L) is a variable; but F itself is not a variable—it does not possess a value unless it is qualified by a nonnull link.

Two further notations are used, to convert between addresses and the values stored there:

a) CONTENTS always denotes a full-word field of a one-word node; hence CONTENTS(1000) denotes the value stored in memory location 1000, i.e., it is a variable having this value. If V is a link variable, CONTENTS(V) is the value pointed to by V (not the value V itself).

b) If V is a variable, LOC(V) denotes the memory location of V. Consequently, if V is a variable whose value is stored in a full word of memory, we have CONTENTS(LOC(V)) = V.

It is easy to transform this notation into MIXAL assembly language code. The values of link variables are put into index registers, and the partial-field capability of MIX is used to refer to the desired field. For example, Algorithm A above could be written thus:

```
NEXT    EQU   4:5          Definition of the NEXT
TAG     EQU   1:1              and TAG fields for the assembler
        LD1   NEWCARD      A1. rI1←NEWCARD.
        LDA   TOP              rA←TOP.                          (5)
        STA   0,1(NEXT)        NEXT(rI1)←rA.
        ST1   TOP          A2. TOP←rI1.
        STZ   0,1(TAG)     A3. TAG(rI1)←0. ∎
```

The ease and efficiency with which these operations can be carried out in a computer is the primary reason for the importance of the "linked memory" concept.

Sometimes we have a single variable which denotes a whole node (i.e., a set of fields instead of just one field). Thus we might write

$$\texttt{CARD} \leftarrow \texttt{NODE(TOP)}, \tag{6}$$

where NODE is a field specification just like CONTENTS, except that it refers to an entire node, and where CARD is a variable which assumes values like those in (1). If there are c words in a node, the notation (6) is an abbreviation for the c assignments

$$\texttt{CONTENTS(LOC(CARD)} + j) \leftarrow \texttt{CONTENTS(TOP} + j), \qquad 0 \le j < c. \tag{7}$$

EXERCISES

1. [*04*] In (3), what is the value of
   a) `TAG(NEXT(TOP))`; b) `NEXT(NEXT(NEXT(TOP)))`?

▶ **2.** [*10*] The text points out that in many cases `CONTENTS(LOC(V)) = V`. Under what conditions do we have `LOC(CONTENTS(V)) = V`?

3. [*11*] Give an algorithm which essentially undoes the effect of Algorithm A, i.e., it removes the top card of the pile (if the pile is not empty) and sets `NEWCARD` to the address of this card.

4. [*18*] Give an algorithm analogous to Algorithm A, except that it puts the new card *face down* at the *bottom* of the pile. (The pile may be empty.)

▶ **5.** [*21*] Give an algorithm which essentially undoes the effect of exercise 4, i.e., assuming that the pile is not empty and its bottom card is face down, it removes this bottom card and makes `NEWCARD` link to it. (This algorithm is sometimes called "cheating" in solitaire games.)

6. [*06*] In the playing card example, suppose that `CARD` is the name of a variable whose value is an entire node. The operation `CARD ← NODE(TOP)` sets the fields of

CARD respectively equal to those of the top of the pile. After this operation, which of the following notations stands for the suit of the top card? (a) SUIT(CARD); (b) SUIT(LOC(CARD)); (c) SUIT(CONTENTS(CARD)); (d) SUIT(TOP)?

7. [*04*] In the text's example MIX program, (5), the link variable TOP is stored in the MIX computer word whose assembly language name is TOP. Assuming the field structure (1), which of the following sequences of code brings the quantity NEXT(TOP) into register A? Explain why the other sequence is incorrect.

a)
```
LDA  TOP(4:5)
```

b)
```
LD1  TOP
LDA  0,1(4:5)
```

▶ **8.** [*18*] Write a MIX program corresponding to Algorithm B.

9. [*23*] Write a MIX program which prints out the alphabetic names of the current contents of the card pile, starting at the top card, with one card per line, and with parentheses around cards that are face down.

2.2. LINEAR LISTS

Usually there is much more structural information present in the data than we actually want to represent directly in a computer. In each "playing card" node of the preceding section, for example, we have a NEXT field to specify what card is beneath it in the pile, but there is no direct way to find what card, if any, is *above* a given card, or to find which pile a given card is in. Of course, there is much information possessed by any *real* deck of playing cards which has been totally suppressed from the computer representation: the details of the design on the back of the cards, the relation of the cards to other objects in the room where the game is being played, the molecules which compose the cards, etc. It is conceivable that such structural information would be relevant in certain computer applications, but obviously we never want to store *all* of the structure present in every situation. Indeed, for most card-playing situations we would not need all of the facts retained in our earlier example; thus the TAG field, which tells whether a card is face up or face down, will often be unnecessary.

It is therefore clear that we must decide in each case how much structure to represent in our tables, and how accessible to make each piece of information. To make this decision, we need to know what operations are to be performed on the data. For each problem considered in this chapter, *we therefore consider not only the data structure but also the class of operations to be done on the data;* the design of computer representations depends on the desired function of the data as well as on its intrinsic properties. Such an emphasis on "function" as well as "form" is basic to design problems in general.

In order to illustrate this point further, let us consider a simple example which arises in computer hardware design: a computer memory is often classified as a "random access memory," i.e., MIX's main memory; or as a "read only memory," i.e., one which is to contain essentially constant information; or a "secondary bulk memory," like MIX's disk units, which cannot be accessed at high speed although large quantities of information can be stored; or an "associative memory," more properly called a "content-addressed memory," i.e., one for which information is addressed by values stored with it rather than by its location; and so on. Note that the intended function of each kind of memory is so important that it enters into the name of the particular memory type; all of these devices are "memory" units, but the purposes to which they are put profoundly influence their design and their cost.

A *linear list* is a set of $n \geq 0$ nodes X[1], X[2], ..., X[n] whose structural properties essentially involve only the linear (one-dimensional) relative positions of the nodes: the facts that, if $n > 0$, X[1] is the first node; when $1 < k < n$, the kth node X[k] is preceded by X[$k - 1$] and followed by X[$k + 1$]; and X[n] is the last node.

The operations we might want to perform on linear lists include, for example, the following.

i) Gain access to the kth node of the list to examine and/or change the contents of its fields.
ii) Insert a new node just before the kth node.
iii) Delete the kth node.
iv) Combine two or more linear lists into a single list.
v) Split a linear list into two or more lists.
vi) Make a copy of a linear list.
vii) Determine the number of nodes in a list.
viii) Sort the nodes of the list into ascending order based on certain fields of the nodes.
ix) Search the list for the occurrence of a node with a particular value in some field.

In operations (i), (ii), and (iii) the special cases $k = 1$ and $k = n$ are of principal importance since the first and last items of a linear list may be easier to get at than a general element is. We will not discuss operations (viii) and (ix) in this chapter, since these topics are the subjects of Chapters 5 and 6, respectively.

A computer application rarely calls for all nine of the above operations in their full generality, so we find there are many ways to represent linear lists depending on the class of operations which are to be done most frequently. It appears to be impossible to design a single representation method for linear lists in which all of these operations are efficient; for example, the ability to gain access to the kth node of a long list for random k is comparatively hard to do if at the same time we are inserting and deleting items in the middle of the list. Therefore we distinguish between types of linear lists depending on the principal operations to be performed, just as we have noted that computer memories are distinguished by their intended applications.

Linear lists in which insertions, deletions, and accesses to values occur almost always at the first or the last node are very frequently encountered, and we give them special names:

A *stack* is a linear list for which all insertions and deletions (and usually all accesses) are made at one end of the list.

A *queue* is a linear list for which all insertions are made at one end of the list; all deletions (and usually all accesses) are made at the other end.

A *deque* ("double-ended queue") is a linear list for which all insertions and deletions (and usually all accesses) are made at the ends of the list.

A deque is therefore more general than a stack or a queue; it has some properties in common with a deck of cards, and it is pronounced the same way. We also distinguish *output-restricted* or *input-restricted* deques, in which deletions or insertions, respectively, are allowed to take place at only one end.

In some disciplines the word "queue" has been used in a much broader sense to describe any kind of list that is subject to insertions and deletions; the special cases identified above are then called various "queuing disciplines." Only the

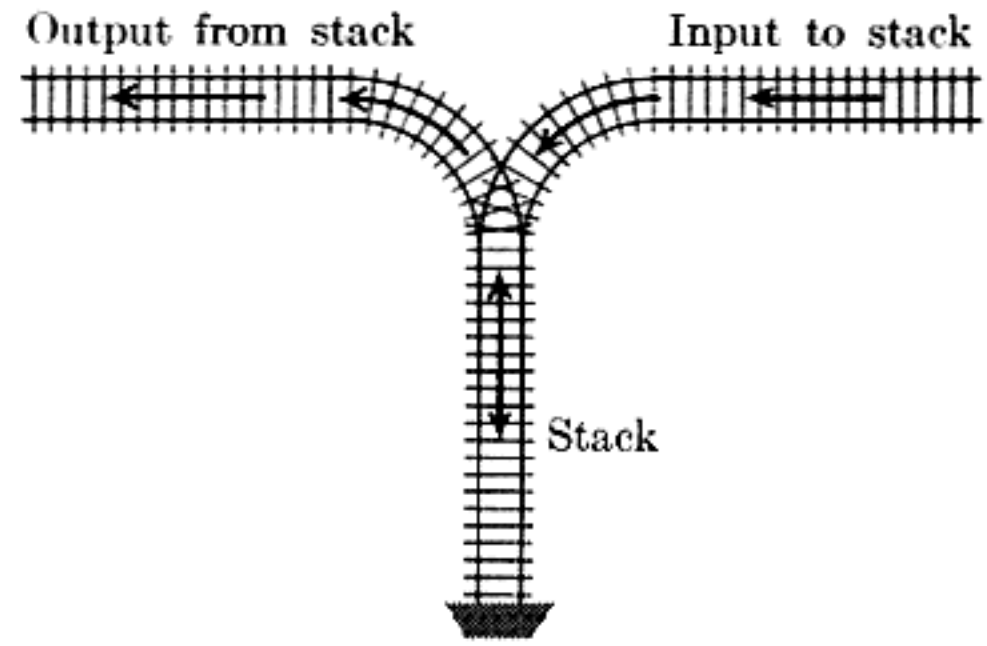


Fig. 1. A stack represented as a railway switching network.

restricted use of the term "queue" is intended in this book, however, by analogy with orderly queues of people waiting in line for service.

Sometimes it helps to understand the mechanism of a stack in terms of an analogy from the switching of railroad cars, as suggested by E. W. Dijkstra (see Fig. 1). A corresponding picture for deques is shown in Fig. 2.

With a stack we always remove the "youngest" item currently in the list, i.e., the one which has been inserted more recently than any other. With a queue just the opposite is true: the "oldest" item is always removed; the nodes leave the list in the same order as they entered it.

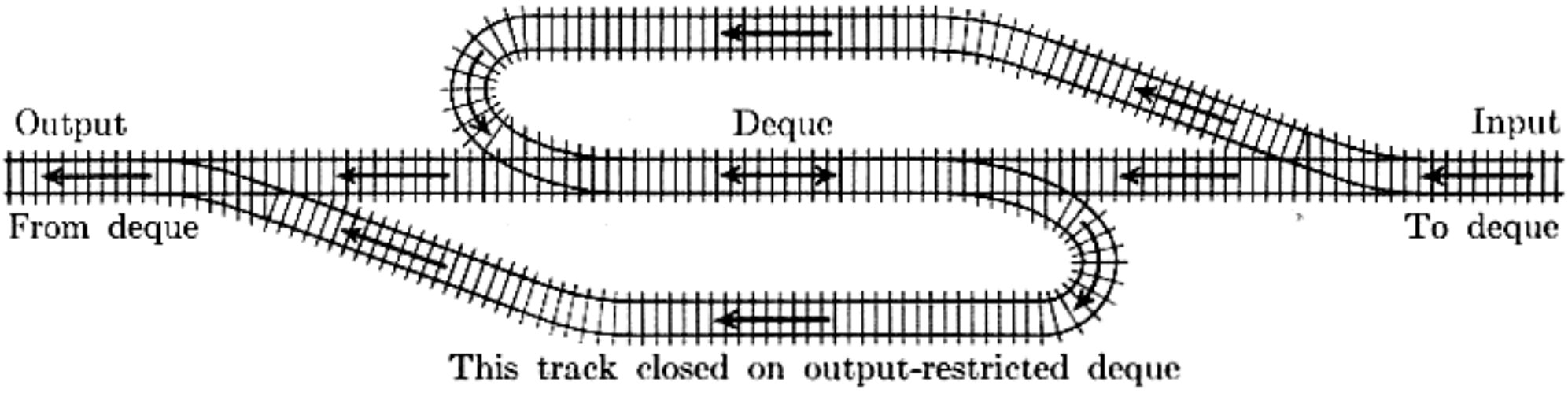


Fig. 2. A deque represented as a railway switching network.

Many people who realized the importance of stacks and queues independently have given other names to these structures: stacks have been called push-down lists, reversion storages, cellars, nesting stores, piles, last-in-first-out ("LIFO") lists, and even yo-yo lists! Queues are sometimes called circular stores or first-in-first-out ("FIFO") lists. The terms LIFO and FIFO have been used for many years by accountants, as names of methods for pricing inventories. Still another term, "shelf," has been applied to output-restricted deques, and input-restricted deques have been called "scrolls" or "rolls." This multiplicity of other names is interesting in itself since it is evidence for the importance of the concepts. The words stack and queue are gradually becoming standard terminology; and of all the other words listed above, only "push-down list" is still reasonably common, particularly in connection with automata theory.

Stacks arise quite frequently in practice. One simple example is the situation where we go through a set of data and keep a list of exceptional conditions or

things to do later; when the original set is processed, we come back to this list to do the subsequent processing, removing its entries until it becomes empty. (For example, see the "saddle point" problem, exercise 1.3.2–10.) Either a stack or a queue serves this purpose, and a stack is generally more convenient. We all have "stacks" in our minds when we are solving problems: One problem leads to another and this leads to another; we stack up the problems and subproblems and remove them as they are solved. Similarly, the process of entering and leaving subroutines during the execution of a computer program has a stack-like behavior. Stacks are particularly useful for the processing of languages with a nested structure, like programming languages, arithmetic expressions, and the literary German "Schachtelsätze." In general, stacks most frequently occur in connection with explicitly or implicitly recursive algorithms, and we will discuss this connection thoroughly in Chapter 8.

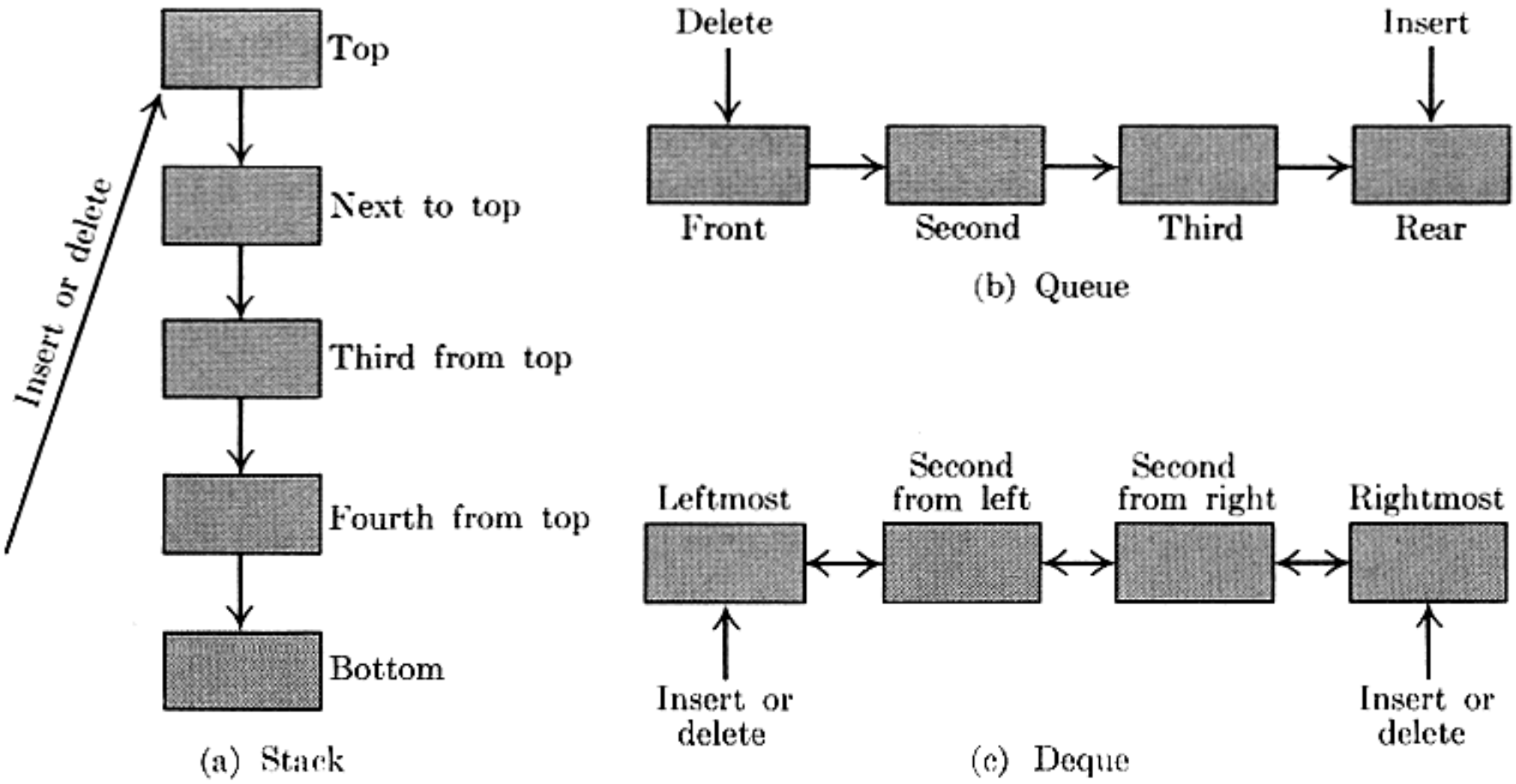


Fig. 3. Special linear lists.

Special terminology is generally used in algorithms referring to these structures: We put an item onto the *top* of a stack, or take off the top item (see Fig. 3a). The *bottom* of the stack is the least accessible item, and it will not be removed until all other items have been deleted. (People often say they *push down* an item onto a stack, and *pop up* the stack when the top item is deleted. This terminology comes from an analogy with the stack of plates often found in cafeterias, or with stacks of cards in some punched-card devices. The brevity of the words "push" and "pop" has its advantages, but these terms falsely imply a motion of the whole list within computer memory. Nothing is physically pushed down; items are added onto the top, as in haystacks or stacks of boxes.) With queues, we speak of the *front* and the *rear* of the queue; things enter at the rear and are removed when they ultimately reach the front position (see Fig. 3b). When referring to deques, we speak of the *left* and *right* ends (Fig. 3c). The concepts of top, bottom, front, and rear are sometimes applied to deques being

used as stacks or queues, with no standard conventions as to whether top, front, and rear are to appear at the left or the right.

Thus we find it easy to use a rich variety of descriptive words from English in our algorithms: "up-down" terminology for stacks, "left-right" terminology for deques, and "waiting in line" terminology for queues.

A little bit of additional notation has proved to be convenient for dealing with stacks and queues: we write

$$A \Leftarrow x \tag{1}$$

(when A is a stack) to mean that the value x is inserted on top of stack A, or (when A is a queue) to mean that x is inserted at the rear of the queue. Similarly, the notation

$$x \Leftarrow A \tag{2}$$

is used to mean that the variable x is set equal to the value at the top of stack A or at the front of queue A, and this value is deleted from A. Notation (2) is meaningless when A is empty, i.e., when A contains no values.

When A is a nonempty stack, we may write

$$\text{top } (A) \tag{3}$$

to denote its top element.

EXERCISES

1. [*06*] An input-restricted deque is a linear list in which items may be inserted at one end but removed from either end; clearly an input-restricted deque can operate either as a stack or as a queue, if we consistently remove all items from one of the two ends. Can an output-restricted deque also be operated either as a stack or a queue?

▶ **2.** [*15*] Imagine four railroad cars positioned on the input side of the track in Fig. 1, numbered 1, 2, 3, and 4, respectively. Suppose we perform the following sequence of operations (which is compatible with the direction of the arrows in the diagram and does not require cars to "jump over" other cars): (a) move car 1 into the stack; (b) move car 2 into the stack; (c) move car 2 into the output; (d) move car 3 into the stack; (e) move car 4 into the stack; (f) move car 4 into the output; (g) move car 3 into the output; (h) move car 1 into the output.

As a result of these operations the original order of the cars, 1234, has been changed into 2431. *It is the purpose of this exercise and the following exercises to examine what permutations are obtainable in such a manner from stacks, queues, or deques.*

If there are six railroad cars numbered 123456, can they be permuted into the order 325641? Can they be permuted into the order 154623? (In case it is possible, show how to do it.)

3. [*25*] The operations (a) through (h) in the previous exercise can be much more concisely described by the code SSXSSXXX, where S stands for "move a car from the input into the stack," and X stands for "move a car from the stack into the output."

Some sequences of S's and X's specify meaningless operations, since there may be no cars available on the specified track; for example, the sequence SXXSSXXS cannot be carried out.

Let us call a sequence of S's and X's *admissible* if it contains n S's and n X's, and if it specifies no operations that cannot be performed. Formulate a rule by which it is easy to distinguish between admissible and inadmissible sequences; show furthermore that no two different admissible sequences give the same output permutation.

4. [*M34*] Find a simple formula for a_n, the number of permutations on n elements that can be obtained with a stack like that in exercise 2.

▸ **5.** [*M28*] Show that it is possible to obtain the permutation $p_1\, p_2 \ldots p_n$ from $1\, 2 \ldots n$ using a stack if and only if there are no indices $i < j < k$ such that $p_j < p_k < p_i$.

6. [*00*] Consider the problem of exercise 2, with a queue substituted for a stack. What permutations of $1\, 2 \ldots n$ can be obtained with the use of a queue?

▸ **7.** [*25*] Consider the problem of exercise 2, with a deque substituted for a stack. (a) Find a permutation of 1 2 3 4 which can be obtained with an input-restricted deque, but which cannot be obtained with an output-restricted deque. (b) Find a permutation of 1 2 3 4 which can be obtained with an output-restricted deque but not with an input-restricted deque. [As a consequence of (a) and (b), there is a definite difference between input-restricted and output-restricted deques.] (c) Find a permutation of 1 2 3 4 which cannot be obtained with either an input-restricted or an output-restricted deque.

8. [*22*] Are there any permutations of $1\, 2 \ldots n$ which cannot be obtained with the use of a deque that is neither input- nor output-restricted?

9. [*M20*] Let b_n be the number of permutations on n elements obtainable by the use of an input-restricted deque. (Note that $b_4 = 22$, as shown in exercise 7.) Show that b_n is also the number of permutations on n elements obtainable with an *output*-restricted deque.

10. [*M25*] (See exercise 3.) Let S, Q, and X denote respectively the operations of inserting an element at the left, inserting an element at the right, and emitting an element from the left, of an output-restricted deque. For example, the sequence QQXSXSXX will transform the input sequence 1 2 3 4 into 1 3 4 2. The sequence SXQSXSXX gives the same transformation.

Find a way to define the concept of an *admissible* sequence of the symbols S, Q, and X in such a way that (a) each admissible sequence performs a meaningful sequence of operations that defines a permutation of n elements; and that (b) each permutation of n elements that is attainable with an output-restricted deque corresponds to precisely one admissible sequence.

▸ **11.** [*M40*] As a consequence of exercises 9 and 10, the number b_n is the number of admissible sequences of length $2n$. Find a "closed form" for the generating function $\sum_{n\geq 0} b_n z^n$.

12. [*HM34*] Compute the asymptotic values of the quantities a_n and b_n in exercises 4 and 11.

13. [*M49*] How many permutations of n elements are obtainable with the use of a general deque?

2.2.2. Sequential Allocation

The simplest and most natural way to keep a linear list inside a computer is to put the list items in sequential locations, one node after the other. We thus will have

$$\texttt{LOC(X[}j+1\texttt{])} = \texttt{LOC(X[}j\texttt{])} + c,$$

where c is the number of words per node. (Usually $c = 1$. When $c > 1$, it is sometimes more convenient to split a single list into c "parallel" lists, so that the kth word of node `X[`j`]` is stored a fixed distance from the location of the first word of `X[`j`]`. We will continually assume, however, that adjacent groups of c words form a single node.) In general,

$$\texttt{LOC(X[}j\texttt{])} = \texttt{L}_0 + cj, \tag{1}$$

where $\texttt{L}_0$ is a constant called the *base address*, the location of an artificially assumed node `X[0]`.

This technique for representing a linear list is so obvious and well-known that there seems to be no need to dwell on it at any length. But we will be seeing many other "more sophisticated" methods of representation later on in this chapter, and it is a good idea to examine the simple case first to see just how far we can go with it. It is important to understand the limitations as well as the power of the use of sequential allocation.

Sequential allocation is quite convenient for dealing with a *stack*. We simply have a variable `T` called the *stack pointer*. When the stack is empty, we let `T` $= 0$. To place a new element `Y` on top of the stack, we set

$$\texttt{T} \leftarrow \texttt{T} + 1; \qquad \texttt{X[T]} \leftarrow \texttt{Y}. \tag{2}$$

And when the stack is not empty, we can set `Y` equal to the top node and delete that node by reversing the actions of (2):

$$\texttt{Y} \leftarrow \texttt{X[T]}; \qquad \texttt{T} \leftarrow \texttt{T} - 1. \tag{3}$$

(Inside a computer it is usually most efficient to maintain the value c`T` instead of `T`, because of (1). Such modifications are easily made, so we will continue our discussion as though $c = 1$.)

The representation of a *queue* or a more general *deque* is a little trickier. An obvious solution is to keep two pointers, say `F` and `R` (for the front and rear of the queue), with `F` $=$ `R` $= 0$ when the queue is empty. Then inserting an element at the rear of the queue would be

$$\texttt{R} \leftarrow \texttt{R} + 1; \qquad \texttt{X[R]} \leftarrow \texttt{Y}; \tag{4}$$

removing the front node (`F` points just below the front) would be

$$\texttt{F} \leftarrow \texttt{F} + 1; \qquad \texttt{Y} \leftarrow \texttt{X[F]}; \qquad \text{if } \texttt{F} = \texttt{R}, \text{ then set } \texttt{F} \leftarrow \texttt{R} \leftarrow 0. \tag{5}$$

But note what can happen: If R always stays ahead of F (so there is always at least one node in the queue) the table entries used are X[1], X[2], . . . , X[1000], . . . , ad infinitum, and this is terribly wasteful of storage space. The simple method (4), (5) is therefore to be used only in the situation when F is known to catch up to R quite regularly (for example, if all deletions come in spurts, which empty the queue).

To circumvent the problem of the queue overrunning memory, we can set aside M nodes X[1], . . . , X[M] arranged implicitly in a circle with X[1] following X[M]. Then the above processes (4), (5) become (for R = F = M initially)

$$\text{if } R = M \text{ then } R \leftarrow 1, \text{ otherwise } R \leftarrow R + 1; \quad X[R] \leftarrow Y. \tag{6}$$

$$\text{if } F = M \text{ then } F \leftarrow 1, \text{ otherwise } F \leftarrow F + 1; \quad Y \leftarrow X[F]. \tag{7}$$

This circular queuing action is much like that which we have already seen in the discussion of input-output buffering (Section 1.4.4).

The above discussion has been very unrealistic in that we have tacitly assumed nothing could go wrong. When we deleted a node from a stack or queue, we assumed that there was at least one node present. When we inserted a node onto a stack or queue, we assumed there was room for it in memory. But clearly the method (6), (7) allows at most M nodes in the entire queue, and methods (2), (3), (4), (5) allow T and R to reach only a certain maximum amount within any given computer program. The following specifications show how the above actions must be rewritten for the common case where we do not assume that these restrictions are automatically satisfied:

$$X \Leftarrow Y \text{ (insert into stack):} \quad T \leftarrow T + 1; \text{ if } T > M, \text{ then OVERFLOW}; \quad X[T] \leftarrow Y. \tag{2a}$$

$$Y \Leftarrow X \text{ (delete from stack):} \quad \text{if } T = 0, \text{ then UNDERFLOW}; Y \leftarrow X[T]; \quad T \leftarrow T - 1. \tag{3a}$$

$$X \Leftarrow Y \text{ (insert into queue):} \quad \begin{cases} \text{if } R = M, \text{ then } R \leftarrow 1, \\ \quad \text{otherwise } R \leftarrow R + 1; \\ \text{if } R = F, \text{ then OVERFLOW}; X[R] \leftarrow Y. \end{cases} \tag{6a}$$

$$Y \Leftarrow X \text{ (delete from queue):} \quad \begin{cases} \text{if } R = F, \text{ then UNDERFLOW}; \\ \text{if } F = M, \text{ then } F \leftarrow 1, \\ \quad \text{otherwise } F \leftarrow F + 1; Y \leftarrow X[F]. \end{cases} \tag{7a}$$

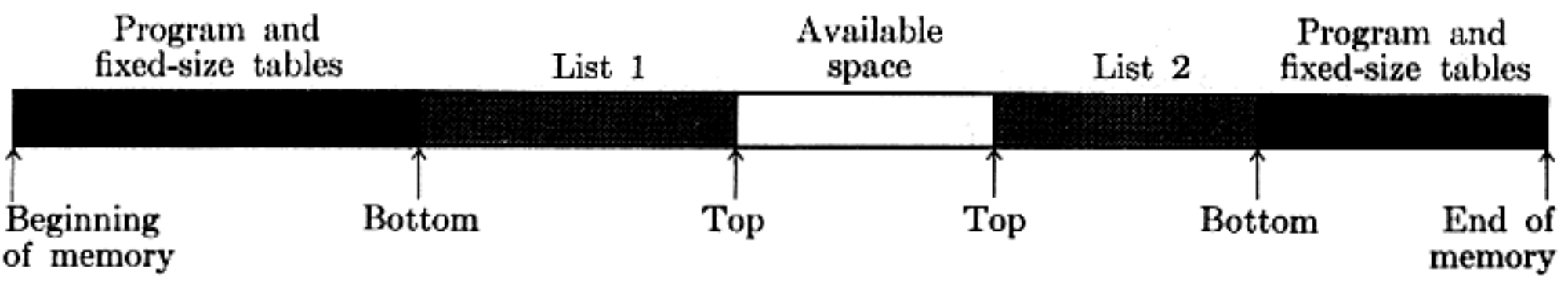
Here we assume that X[1], . . . , X[M] is the total amount of space allowed for the list; OVERFLOW and UNDERFLOW mean an excess or deficiency of items.

The next question is, "What do we do when UNDERFLOW or OVERFLOW occurs?" In the case of UNDERFLOW, we have tried to remove a nonexistent item; this is usually a meaningful condition—not an error situation—which can be used to govern the flow of a program, e.g., we might want to delete items repeatedly

until UNDERFLOW occurs. An OVERFLOW situation, however, is usually an error; it means the table is full already, yet there is still more information that ought to be put in. The usual policy in case of OVERFLOW is to report reluctantly that the program cannot go on because its storage capacity has been exceeded, and the program terminates.

Of course we would hate to give up in an OVERFLOW situation when only one list has gotten too large, while other lists of the same program may very well have plenty of room remaining. In the above discussion we were primarily thinking of a program with only one list. However, we frequently encounter programs which involve several stacks, each of which has a dynamically varying size. In such a situation we would hate to impose a maximum size on each stack, for usually the size is somewhat unpredictable; and even if a maximum size has been determined for each stack, we will rarely find *all* stacks simultaneously filling their maximum capacity.

When there are just two variable size lists, they can coexist together very nicely if we let the lists grow toward each other:



Here list 1 expands to the right, and list 2 (stored in reverse order) expands to the left. OVERFLOW will not occur unless the total size of both lists exhausts all memory space. The lists may independently expand and contract so that the effective maximum size of each one could be significantly more than half of the available space. The above layout of memory space is used very frequently.

The reader may easily convince himself, however, that *there is no way* to store three or more variable-size sequential lists in memory so that (a) OVERFLOW will occur only when the total size of all lists exceeds the total space, and (b) each list has a fixed location for its "bottom" element. When there are, say, ten or more variable size lists—and this is not unusual—the storage allocation problem becomes very significant. If we wish to satisfy condition (a), we must give up condition (b); that is, we must allow the "bottom" elements of the lists to change their positions. This means the location L_0 of Eq. 1 is *not constant* any longer; no reference to the table may be made to an absolute memory address, all references must be relative to the base address L_0. In the case of MIX, the coding to bring a one-word node into register A is changed

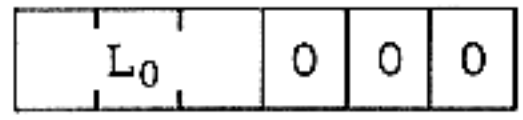
from

```
LD1   I
LDA   L0, 1
```

to, e.g.,

```
LD1   I
LDA   BASE(0:2)
STA   *+1(0:2)
LDA   *,1
```

(8)

where BASE contains 

.

This relative addressing is evidently slower to do than when the base was fixed, although we find it would be only slightly slower if MIX had an "indirect addressing" feature (see exercise 3).

An important special case occurs when each of the variable size lists is a stack. Then, since only the top element of each stack is relevant at any time, we can proceed almost as efficiently as before. Suppose that we have n stacks; the insertion and deletion algorithms above become the following, if BASE[i] and TOP[i] are link variables for the ith stack:

Insertion: TOP[i] $\leftarrow$ TOP[i] $+ c$; if TOP[i] $>$ BASE[$i+1$], then OVERFLOW; otherwise set NODE(TOP[i]) $\leftarrow$ Y. (9)

Deletion: if TOP[i] $=$ BASE[i], then UNDERFLOW; otherwise set Y $\leftarrow$ NODE(TOP[i]), TOP[i] $\leftarrow$ TOP[i] $- c$. (10)

Here BASE[$i+1$] is the beginning location of the $(i+1)$st stack. The condition TOP[i] $=$ BASE[i] means stack i is empty.

In the above situation, OVERFLOW is no longer such a crisis as it was before; we can "repack memory," making room for the table that overflowed by taking some away from tables that aren't yet filled. A number of possible ways to do this suggest themselves, and since these repacking algorithms are very important in connection with sequential allocation of linear lists, we will now consider this problem in detail. We will start by giving the simplest of these methods, and will then consider some of the alternatives.

Assume that there are n stacks, and the values BASE[i] and TOP[i] are to be manipulated as in (9), (10). These stacks are all to share the common memory area consisting of all locations L with $\mathrm{L}_0 \le \mathrm{L} < \mathrm{L}_\infty$. (Here L_0 and L_∞ are constants which specify the total number of words available for use; $\mathrm{L}_\infty - \mathrm{L}_0$ is a multiple of c.) We might start out with all stacks empty, and BASE[i] $=$ TOP[i] $= \mathrm{L}_0 - c$, for all i. We also set BASE[$n+1$] $\equiv \mathrm{L}_\infty - c$ so that (9) will work properly for $i = n$. Now whenever a particular stack, except stack n, gets more items in it than it ever had before, OVERFLOW will occur.

When stack i overflows, there are three possibilities:

a) We find the smallest k for which $i < k \le n$ and TOP[k] $<$ BASE[$k+1$], if any such k exist. Now move things *up* one notch:

Set CONTENTS(L $+ c$) $\leftarrow$ CONTENTS(L), for TOP[k] $+ c >$ L $\ge$ BASE[$i+1$] $+ c$.

(Note that this should be done for decreasing, not increasing, values of L to avoid losing information. It is possible that TOP[k] $=$ BASE[$i+1$], in which case nothing needs to be moved.)

Set BASE[j] $\leftarrow$ BASE[j] $+ c$, TOP[j] $\leftarrow$ TOP[j] $+ c$, for $i < j \le k$.

b) No k can be found as in (a), but we find the largest k for which $1 \le k < i$ and TOP[k] < BASE[$k+1$]. Now move things *down* one notch:

Set CONTENTS(L) $\leftarrow$ CONTENTS(L $+ c$), for BASE[$k+1$] $\le$ L < TOP[i].

(Note that this should be done for increasing values of L.)

Set BASE[j] $\leftarrow$ BASE[j] $- c$, TOP[j] $\leftarrow$ TOP[j] $- c$, for $k < j \le i$.

c) We have TOP[k] = BASE[$k+1$] for all $k \neq i$. Then obviously we cannot find room for the new stack entry, and we must give up.

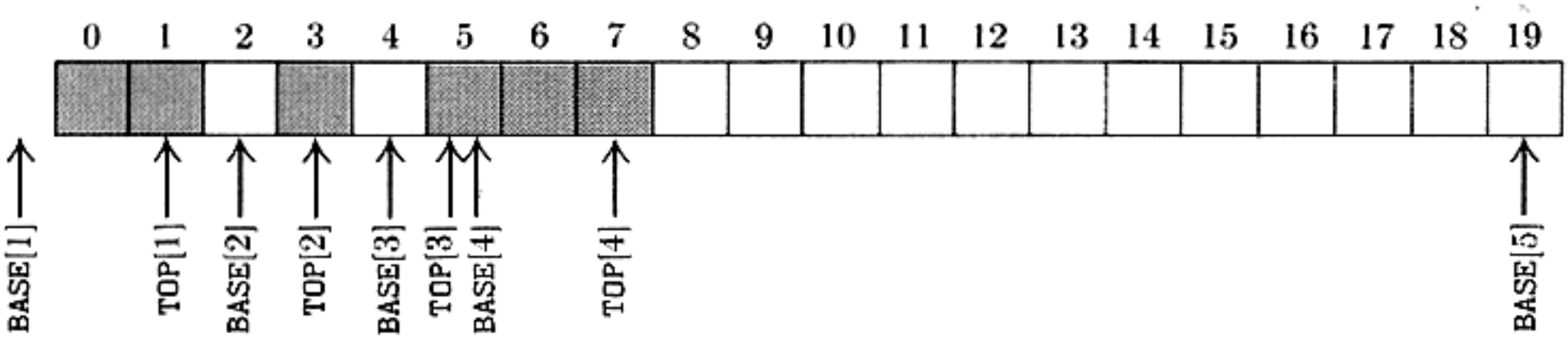


Fig. 4. Example of memory configuration after several insertions and deletions.

Figure 4 illustrates the configuration of memory for the case $n = 4$, $c = 1$, $L_0 = 0$, $L_\infty = 20$, after the successive actions

$$I_1^* \; I_1^* \; I_4 \; I_2^* \; D_1 \; I_3^* \; I_1 \; I_1^* \; I_2^* \; I_4 \; D_2 \; D_1.$$

(Here I_j and D_j refer to insertion and deletion in stack j, and an asterisk refers to an occurrence of OVERFLOW, assuming that no space is initially allocated to stacks 1, 2, and 3.)

It is clear that many of the first stack overflows which occur with this method could be eliminated if we choose our initial conditions wisely, instead of allocating all space initially to the nth stack as suggested above. For example, if we expect each stack to be of the same size, we could start out with

$$\text{BASE}[j] = \text{TOP}[j] = c\left\lfloor \left(\frac{j-1}{n}\right)\left(\frac{L_\infty - L_0}{c}\right)\right\rfloor + L_0 - c. \tag{11}$$

Operating experience with a particular program may suggest better starting values; however, no matter how well the initial allocation is set up, it can save at most a fixed number of overflows, and the effect is noticeable only in the early stages of a program run.

Another possible improvement in the above method would be to make room for more than one new entry each time memory is repacked. The shifting of tables in memory is a time-consuming operation, and we can gain speed by shifting up 2 or 3 at once instead of shifting by 1 several times.

This idea has been exploited by J. Garwick, who suggests a complete repacking of memory when overflow occurs, based on the change in size of each

stack since the last repacking. This algorithm uses an additional array, called OLDTOP[i], $1 \le i \le n$, which retains the value that TOP[i] had just after the previous time memory was allocated. Initially, the tables are set as before, with OLDTOP[i] = TOP[i]. The new algorithm proceeds as follows:

Algorithm G (*Reallocate sequential tables*). Assume that OVERFLOW has occurred in stack i, according to (9). After Algorithm G has been performed, either we will find the memory capacity exceeded or the memory will have been rearranged so that the action NODE(TOP[i]) $\leftarrow$ Y may be done. (Note that TOP[i] has already been increased in (9) before Algorithm G takes place.)

G1. [Initialize.] Set SUM $\leftarrow L_\infty - L_0$, INC $\leftarrow 0$. Then do step G2 for $1 \le j \le n$. (The effect will be to make SUM equal to the total amount of memory space left, and INC equal to the total amount of increases in table sizes since the last allocation.) After this has been done, go on to step G3.

G2. [Gather statistics.] Set SUM $\leftarrow$ SUM $-$ (TOP[j] $-$ BASE[j]). If TOP[j] $>$ OLDTOP[j], set D[j] $\leftarrow$ TOP[j] $-$ OLDTOP[j] and INC $\leftarrow$ INC $+$ D[j]; otherwise set D[j] $\leftarrow 0$.

G3. [Is memory full?] If SUM < 0, we cannot proceed.

G4. [Compute allocation factors.] Set $\alpha = (0.1/n)(\text{SUM}/c)$, $\beta = (0.9/\text{INC})(\text{SUM}/c)$. (Here α and β are fractions, not integers, which are to be computed to reasonable accuracy. In the succeeding step, each list will get 10 percent more available space than it now has; the other 90 per cent of this space will be divided proportionately to the amount of increase in table size since the previous allocation.)

G5. [Compute new base addresses.] Set NEWBASE[1] $\leftarrow$ BASE[1]; then for $j = 2, 3, \ldots, n$ set

$$\text{NEWBASE}[j] \leftarrow \text{NEWBASE}[j-1] + \text{TOP}[j-1] - \text{BASE}[j-1] + c\lfloor\alpha\rfloor + c\lfloor \text{D}[j-1]\beta\rfloor.$$

G6. [Repack.] Perform Algorithm R below, and then set OLDTOP[j] $\leftarrow$ TOP[j] for $1 \le j \le n$. ∎

It is readily seen that since $\lfloor\alpha\rfloor \le \alpha$ and $\lfloor \text{D}[j-1]\beta\rfloor \le \text{D}[j-1]\beta$, the reallocation formulas of step G5 will never give anomalous results. Perhaps the most interesting part of this whole algorithm is the general repacking process, which we now describe. Repacking is not trivial, since some portions of memory shift up and others shift down; it is obviously important not to overwrite any of the good information in memory while it is being moved.

Algorithm R (*Relocate sequential tables*). For $1 \le j \le n$ the information specified by BASE[j] and TOP[j] in accord with the conventions stated above is moved to new positions specified by NEWBASE[j], and the values of BASE[j] and TOP[j] are suitably adjusted.

R1. [Initialize.] Set $j \leftarrow 1$. (Note that stack 1 never needs to be moved, so for efficiency the programmer should put the largest stack first if he knows which one will be largest.)

R2. [Find start of shift.] Increase j in steps of 1 until finding either
a) NEWBASE[j] < BASE[j]: go to R3; or
b) NEWBASE[j] > BASE[j]: go to R4; or
c) $j > n$: the algorithm terminates.

R3. [Shift down.] Set $\delta \leftarrow$ BASE[j] $-$ NEWBASE[j]. Set CONTENTS(L $-\ \delta$) $\leftarrow$ CONTENTS(L), for L = BASE[j] $+ c$, BASE[j] $+ c + 1, \ldots,$ TOP[j] $+ c - 1$. (Note that it is possible for BASE[j] to equal TOP[j], in which case no action is required.) Set BASE[j] $\leftarrow$ NEWBASE[j], TOP[j] $\leftarrow$ TOP[j] $- \delta$. Go to R2.

R4. [Find top of shift.] Find the smallest $k \geq j$ for which NEWBASE[$k + 1$] $\leq$ BASE[$k + 1$]. (*Note:* NEWBASE[$n + 1$] should equal BASE[$n + 1$], so that such a k will always exist.) Then do step R5 for $t = k, k - 1, \ldots, j$; finally set $j \leftarrow k$ and go to R2.

R5. [Shift up.] Set $\delta \leftarrow$ NEWBASE[t] $-$ BASE[t]. Set CONTENTS(L $+\ \delta$) $\leftarrow$ CONTENTS(L), for L = TOP[t] $+ c - 1$, TOP[t] $+ c - 2, \ldots,$ BASE[t] $+ c$. (Note that as in step R3, no action may be needed here.) Set BASE[t] $\leftarrow$ NEWBASE[t], TOP[t] $\leftarrow$ TOP[t] $+ \delta$. ∎

In Algorithms G and R we have purposely made it possible to have

$$\texttt{OLDTOP}[j] \equiv \texttt{D}[j-1] \equiv \texttt{NEWBASE}[j]$$

for $1 \leq j \leq n + 1$, that is, these three tables can share common memory locations since their values are never needed at conflicting times. It will be necessary to perform step G2 for *decreasing* values of j and to set NEWBASE[$n + 1$] $\leftarrow$ BASE[$n + 1$] in step G5 when using this overlap.

We have described these repacking algorithms for stacks, but it is clear that they can be adapted to any relatively addressed tables in which the current information is contained between BASE[j] and TOP[j]. Other pointers (for example, FRONT[j], REAR[j]) could also be attached to the lists, making them serve as a queue or deque. See exercise 8 which considers the case of a queue in detail.

The mathematical analysis of dynamic storage-allocation algorithms is extremely difficult. Some interesting results appear in the exercises below, although they only begin to scratch the surface as far as the general behavior is concerned. Perhaps some of the mathematical results of "queuing theory" can be applied to this study, although the orientation of that theory seems to be different.

As an example of the theory which *can* be derived, suppose we consider the case when the tables grow only by insertion (deletions and subsequent insertions that cancel their effect are ignored), and let us assume further that each table is

expected to fill at the same rate. This situation can be modeled by imagining a sequence of m insertion operations $a_1, a_2, \ldots, a_m$, where each a_i is an integer between 1 and n (representing an insertion on top of stack a_i). For example, the sequence 1, 1, 2, 2, 1 means two insertions to stack 1, followed by two to stack 2, followed by another onto stack 1. We can regard each of the n^m possible specifications $a_1, a_2, \ldots, a_m$ as equally likely, and then we can ask for the average number of times it is necessary to move a word from one location to another during the repacking operations as the entire table is built. For the first algorithm, starting with all available space given to the nth stack, we find that the average number of move operations required is

$$\frac{1}{2}\left(1-\frac{1}{n}\right)\binom{m}{2}. \tag{12}$$

Thus, as we might expect, the number of moves is essentially proportional to the *square* of the number of items in the tables. If we don't start with all the space given to the nth stack, but rather give each of the first $n-1$ stacks t cells to expand in before overflow occurs, we get

$$\frac{1}{2n^t}\sum_{0\le k\le m-t-2}\binom{t-1+k}{k}\binom{m-t-k}{2}\left(1-\frac{1}{n}\right)^{k+1}, \quad \text{for} \quad t\ge 0. \tag{13}$$

This is better than (12), but again it is essentially proportional to m^2. If we put $n=\infty$ in the sum, (13) reduces to

$$\frac{1}{2n^t}\binom{m}{t+2},$$

which is an upper bound that is not too informative; a lower bound is given by the $k=0$ term, namely

$$\frac{1}{2n^t}\left(1-\frac{1}{n}\right)\binom{m-t}{2}.$$

The moral of the story seems to be that a very large number of moves will be made if a reasonably large number of items are put in the tables. This is the price we must pay for the ability to pack a large number of sequential tables together tightly. No theory has been developed to analyze the characteristics of Algorithm G, and it is unlikely that any simple model will be able to describe the characteristics of real-life tables in such an environment anyway.

Experience shows that when memory is only half loaded (i.e., the available space equals half the total space), we need very little rearranging of the tables with Algorithm G; the important thing is perhaps that the algorithm behaves well in the half-full case and that it at least delivers the right answers in the almost-full case.

But let us think about the almost-full case more carefully: When the tables nearly fill memory, Algorithm R takes rather long to perform its job, and to

make matters worse OVERFLOW is much more frequent just before memory space is used up. There are very few programs that will come *close* to filling memory without soon thereafter completely overflowing it; and those that do overflow memory will probably waste enormous amounts of time in Algorithms G and R just before memory is overrun. Unfortunately, an undebugged program frequently will overflow memory capacity. To avoid wasting all this time, a possible suggestion would be to stop Algorithm G in step G3 if SUM is less than $S_{\min}$, where the latter is chosen by the programmer to prevent excessive repacking. When there are many variable-size sequential tables, we should *not* expect to make use of 100 percent of the memory space before storage is exceeded.

EXERCISES

▶ **1.** [*15*] In the queue operations given by (6a), (7a), how many items can be in the queue at one time without OVERFLOW occurring?

▶ **2.** [*22*] Generalize the method of (6a), (7a) to apply to any deque with less than M elements. In other words, give specifications for the other two operations, "delete from rear" and "insert at front."

3. [*21*] Suppose that MIX is extended as follows: The I-field of each instruction is to have the form $8I_1 + I_2$, where $0 \le I_1 < 8$, $0 \le I_2 < 8$. In assembly language one writes "OP ADDRESS, I_1:I_2" or (as presently) "OP ADDRESS, I_2" if $I_1 = 0$. The meaning is to perform first the "address modification" I_1 on ADDRESS, then to perform the "address modification" I_2 on the resulting address, and finally to perform the OP with the new address. The address modifications are defined as follows:

0: M = A
1: M = A + (rI1)
2: M = A + (rI2)
. . .
6: M = A + (rI6)
7: M = resulting address defined from the "ADDRESS,I_1:I_2" fields found in location A. The case $I_1 = I_2 = 7$ in location A is not allowed.

Here A denotes the address before the operation, and M denotes the resulting address after the address modification. In all cases the result is undefined if the value of M does not fit in two bytes plus sign. The execution time is increased by one unit for each "indirect-addressing" (modification 7) operation performed.

As a nontrivial example, suppose that location 1000 contains "NOP 1000,1:7"; location 1001 contains "NOP 1000,2"; and index registers 1 and 2 respectively contain 1 and 2. Then the command "LDA 1000,7:2" is equivalent to "LDA 1004", because

$$1000,7{:}2 = (1000,1{:}7),2 = (1001,7),2 = (1000,2),2 = 1002,2 = 1004.$$

a) Using this indirect addressing feature (if necessary), show how to simplify the coding on the right-hand side of (8) so that two instructions are saved per reference to the table. How much faster is your code than (8)?

b) Suppose there are several tables whose base addresses are stored in locations BASE, BASE + 1, BASE + 2, . . . ; how can the indirect addressing feature be used to bring the I th element of the J th table into register A in one instruction, assuming that I is in rI1 and J is in rI2?

c) What is the effect of the instruction "ENT4 X,7", assuming the (3:3)-field in location X is zero?

4. [*23*] Assume that MIX has been extended as in exercise 3. Show how to give a *single instruction* (plus auxiliary constants) for each of the following actions:

i) To loop indefinitely because indirect addressing never terminates.

ii) To bring into register A the value CONTENTS(CONTENTS(X)), where X and CONTENTS(X) are link variables whose values are stored in the (0:2)-field of their memory locations.

iii) To bring into register A the value CONTENTS(CONTENTS(CONTENTS(X))), under assumptions like those in (ii).

iv) To bring into register A the contents of location (rI1) + (rI2) + · · · + (rI6).

v) To quadruple the current value of rI6.

▸ **5.** [*35*] The extension of MIX suggested in exercise 3 has an unfortunate restriction that "7:7" is not allowed in an indirectly addressed location.

a) Show that without this restriction it would be necessary for the MIX hardware to be capable of maintaining a long internal stack of three-bit items. (This would be prohibitively expensive hardware, even for a mythical computer like MIX.)

b) Show how such a stack is not needed under the present restriction; in other words, design an algorithm with which the hardware of a computer could perform the desired address modifications without much additional register capacity.

c) Give a milder restriction than that of exercise 3 on the use of 7:7 which alleviates the difficulties of exercise 4(iii), yet which can be cheaply implemented in computer hardware.

6. [*10*] Starting with the memory configuration shown in Fig. 4, determine which of the following sequences of operations causes overflow or underflow. (a) I_1; (b) I_2; (c) I_3; (d) $I_4I_4I_4I_4I_4$; (e) $D_2D_2I_2I_2I_2$.

7. [*12*] Step G4 of Algorithm G indicates a division by the quantity INC. Can INC ever be zero at that point in the algorithm?

▸ **8.** [*25*] Explain how to modify (9), (10) and the repacking algorithms for the case that one or more of the lists is a queue being handled circularly as in (6a), (7a).

▸ **9.** [*M27*] Using the mathematical model described near the end of the section, prove that Eq. (12) is the expected number of moves. (Note that the sequence 1, 1, 4, 2, 3, 1, 2, 4, 2, 1 specifies $0+0+0+1+1+3+2+0+3+6 = 16$ moves.)

10. [*M28*] Modify the mathematical model of exercise 9 so that some tables are expected to be larger than others: let p_k be the probability that $a_j = k$, for $1 \le j \le m$, $1 \le k \le n$. Thus $p_1 + p_2 + \cdots + p_n = 1$; the previous exercise considered the special case $p_k = 1/n$ for all k. Determine the expected number of moves, as in Eq. (12), for this more general case. It is possible to rearrange the relative order of the n lists so that lists which are expected to be longer are put to the right (or to the left) of lists expected to be shorter; what is the best relative order for the n lists to minimize the expected number of moves, based on $p_1, p_2, \ldots, p_n$?

11. [*M30*] Generalize the argument of exercise 9 to show that if each stack starts out with t available spaces (so that the first t insertions in any stack cause no movement), then the expected number of moves to make m entries is given by Eq. (13). Note that if $t = 2$, the sequence in exercise 9 specifies $0 + 0 + 0 + 0 + 0 + 3 + 0 + 0 + 3 + 6 = 12$ moves.

12. [*M28*] The advantage of having two tables coexist in memory by growing towards each other, rather than by having them kept in separate independently bounded areas, may be quantitatively estimated (to a certain extent) as follows. Use the model of exercise 9 with $n = 2$; for each of the 2^m equally probable sequences $a_1, a_2, \ldots, a_m$, let there be k_1 1's and k_2 2's. (Here k_1 and k_2 are the respective sizes of the two tables after the memory is full. We are able to run the algorithm with $m = k_1 + k_2$ locations when the tables are adjacent, instead of $2 \max(k_1, k_2)$ locations to get the same effect with separate tables.)

What is the average value of $\max(k_1, k_2)$?

13. [*M50*] The value $\max(k_1, k_2)$ investigated in exercise 12 will be even greater if larger fluctuations in the tables are introduced by allowing random *deletions* as well as random insertions. Suppose we alter the model so that with probability p the sequence value a_j is interpreted as a deletion instead of an insertion; the process continues until $k_1 + k_2$ (the total number of table locations in use) equals m. A deletion from an empty list causes no effect.

For example if $m = 4$, it can be shown that when the above process stops, we get the probability distribution:

the value of (k_1, k_2),	$(4, 0)$ or $(0, 4)$,	$(3, 1)$ or $(1, 3)$,	$(2, 2)$,
occurs with probability:	$\dfrac{1}{16 - 12p + 4p^2}$,	$\dfrac{1}{4}$,	$\dfrac{6 - 6p + 2p^2}{16 - 12p + 4p^2}$.

Thus as p increases, the difference between k_1 and k_2 tends to increase. It is not difficult to show that in the limit as p approaches unity, the distribution of k_1 becomes essentially uniform, and the limiting value of $\max(k_1, k_2)$ is exactly $\frac{3}{4}m$, when m is even. This behavior is quite different from that in the previous exercise (when $p = 0$); however, it may not be extremely significant, since when p approaches unity, the amount of time taken to terminate the process rapidly approaches infinity. The problem posed in this exercise is to examine the dependence of $\max(k_1, k_2)$ on p and m, and to determine asymptotic formulas for fixed p (like $p = \frac{1}{3}$) as m approaches infinity.

14. [*HM40*] Generalize the result of exercise 12 to arbitrary $n \geq 2$, by showing that, when n is fixed and m approaches infinity, the quantity

$$\frac{m!}{n^m} \sum_{\substack{k_1 + \cdots + k_n = m \\ k_1, \ldots, k_n \geq 0}} \frac{\max(k_1, \ldots, k_n)}{k_1! \cdots k_n!}$$

has the asymptotic form $(m/n) + c_n\sqrt{m} + O(1)$. Determine the constants c_2, c_3, c_4, and c_5.

15. [*HM40*] Survey the field of queuing theory to see what the results of that theory tell us about storage allocation algorithms.

16. [*40*] Using a Monte Carlo method, simulate the behavior of Algorithm G under varying distributions of insertions and deletions. What do your experiments imply about the efficiency of Algorithm G? Compare its performance with the algorithm given earlier that shifts up and down one node at a time.

17. [*20*] The text illustrates how two stacks can be located so they grow towards each other, thereby making efficient use of a common memory area. Can two *queues*, or a stack and a queue, make use of a common memory area with the same degree of efficiency?

2.2.3. Linked Allocation

Instead of keeping a linear list in sequential memory locations, we can make use of a much more flexible scheme in which each node contains a link to the next node of the list.

Sequential allocation:

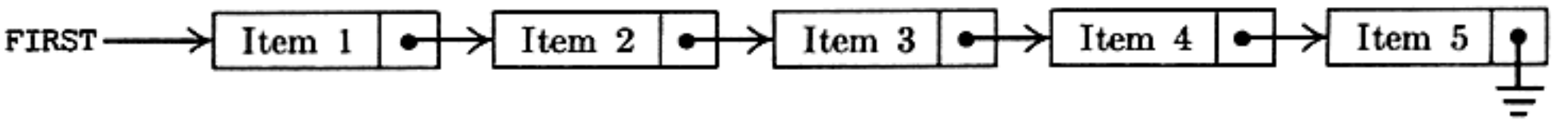
Address	Contents
$L_0 + c$:	Item 1
$L_0 + 2c$:	Item 2
$L_0 + 3c$:	Item 3
$L_0 + 4c$:	Item 4
$L_0 + 5c$:	Item 5

Linked allocation:

Address	Contents	
A:	Item 1	B
B:	Item 2	C
C:	Item 3	D
D:	Item 4	E
E:	Item 5	Λ

Here A, B, C, D, and E are arbitrary locations in the memory, and Λ is the null link (see Section 2.1). The program which uses this table in the case of sequential allocation would have an additional variable or constant whose value indicates that the table is five items in length, or else this information would be specified by a "sentinel" code within item 5 or in the following location. A program for linked allocation would have a link variable or constant that points to A, and from A all the other items of the list can be found.

Recall from Section 2.1 that links are often shown simply by arrows, since the actual memory locations occupied are usually irrelevant. The linked table above might therefore be shown as follows:



(1)

Here FIRST is a link variable pointing to the first node of the list.

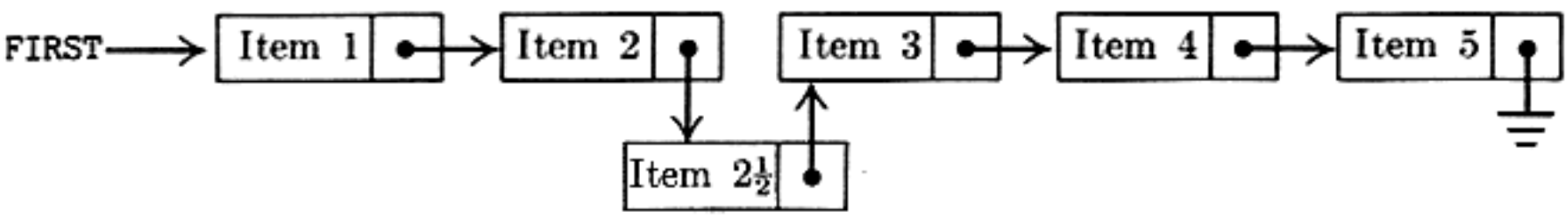
There are several obvious comparisons we can make between these two basic forms of storage:

1) Linked allocation takes up additional memory space for the links. This can be the dominating factor in some situations. However, we frequently find

that the information in a node does not take up a whole word anyway, so there is already space for a link field present. Also, it is possible in many applications to combine several items into one node so that there is only one link for several items of information (see exercise 2.5–2). But even more importantly, there is often an implicit *gain* in storage by the linked memory approach, since tables can overlap, sharing common parts; and in many cases, sequential allocation will not be as efficient as linked allocation unless a rather large number of additional memory locations are left vacant anyway. For example, the discussion at the end of the previous section shows how the systems described there are necessarily inefficient when memory is densely loaded.

2) It is easy to delete an item from within a linked list. For example, to delete item 3 we need only change the link associated with item 2. But with sequential allocation such a deletion generally implies moving a large part of the list up into different locations.

3) It is easy to insert an item into the midst of a list when the linked scheme is being used. For example, to insert an item $2\frac{1}{2}$ into (1) we need change only two links:



(2)

By comparison, this operation would be extremely time-consuming in a long sequential table.

4) References to random parts of the list are much faster in the sequential case. To gain access to the kth item in the list, when k is a variable, takes a fixed time in the sequential case, but it takes k iterations to march down to the right place in the linked case. Thus the usefulness of linked memory is predicated on the fact that in the large majority of applications we want to walk through lists sequentially, not randomly; if items in the middle or at the bottom of the list are needed, we try to keep an additional link variable or list of link variables pointing to the proper places.

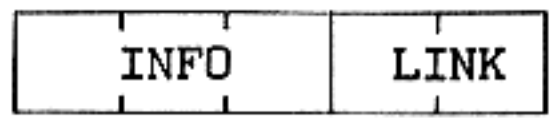
5) The linked scheme makes it easier to join two lists together or to break one apart.

6) The linked scheme lends itself immediately to more intricate structures than simple linear lists. We can have a variable number of variable size lists; any node of the list may be a starting point for another list; the nodes may simultaneously be linked together in several orders corresponding to different lists; and so on.

7) Simple operations, like proceeding sequentially through a list, are slightly faster for sequential lists on many computers. For MIX, the comparison is between "`INC1 1`" and "`LD1 0,1(LINK)`", which is only one cycle different, but many machines do not enjoy the property of being able to load an index register from an indexed location.

Thus we see that the linking technique, which frees us from any constraints imposed by the consecutive nature of computer memory, gives us a good deal more efficiency in some operations, while we lost some capabilities in other cases. It is usually clear which allocation technique will be most appropriate in a given situation, and often both methods are used in different lists of the same program.

In the next few examples we will assume for convenience that a node has one word and that it is broken into the two fields INFO and LINK:



(3)

The use of linked allocation generally implies the existence of some mechanism for finding empty space available for a new node, when we wish to insert some newly created information onto a list. This is usually done by having a special list called the *list of available space*. We will call it the AVAIL list (or, the AVAIL stack, since it is usually treated in a last-in-first-out manner). The set of all nodes not currently in use is linked together in a list just like any other list; the link variable AVAIL refers to the top element of this list. Thus, if we want to set link variable X to the address of a new node, and to reserve that node for future use, we can proceed as follows:

$$\texttt{X} \leftarrow \texttt{AVAIL}, \qquad \texttt{AVAIL} \leftarrow \texttt{LINK(AVAIL)}. \tag{4}$$

This effectively removes the top of the AVAIL stack and makes X point to the node just removed. *Operation* (4) *occurs so often that we have a special notation for it:* "X $\Leftarrow$ AVAIL" will mean X is set to point to a new node.

When a node is deleted and no longer needed, process (4) can be reversed:

$$\texttt{LINK(X)} \leftarrow \texttt{AVAIL}, \qquad \texttt{AVAIL} \leftarrow \texttt{X}. \tag{5}$$

This operation puts the node addressed by X back onto the list of raw material; we denote (5) by "AVAIL $\Leftarrow$ X".

Several important things have been omitted from the above discussion of the AVAIL stack. We did not say how to set it up at the beginning of a program; clearly this can be done by (a) linking together all nodes which are to be used for linked memory, (b) setting AVAIL to the address of the first of these nodes, and (c) making the last node link to Λ. The set of all nodes which can be allocated is called the *storage pool.*

A more important omission in our discussion was the test for overflow: we neglected to check in (4) if all available memory space has been taken. The operation X $\Leftarrow$ AVAIL should really be defined as follows:

$$\text{if } \texttt{AVAIL} = \Lambda, \text{ then } \texttt{OVERFLOW}; \text{ otherwise } \texttt{X} \leftarrow \texttt{AVAIL}, \ \texttt{AVAIL} \leftarrow \texttt{LINK(AVAIL)}. \tag{6}$$

The possibility of overflow must always be considered. Here OVERFLOW generally

means that we terminate the program with regrets; or else we can go into a "garbage collection" routine which attempts to find more available space. Garbage collection is discussed in Section 2.3.5.

There is another important technique for handling the AVAIL stack: We often do not know in advance how much memory space is to be used for the storage pool. There may be a sequential table of variable size which is to coexist in memory with the linked tables; in such a case we do not want the linked memory area to take any more space than is absolutely necessary. So suppose that we wish to place the linked memory area in ascending locations beginning with L_0, and that this area is never to extend past the value of variable SEQMIN (which represents the current lower bound of other tables). Then we can proceed as follows, using a new variable POOLMAX:

a) Initially set AVAIL $\leftarrow \Lambda$ and POOLMAX $\leftarrow L_0 - c$, where c is the node size.
b) The operation X $\Leftarrow$ AVAIL becomes the following:

"If AVAIL $\neq \Lambda$, then X $\leftarrow$ AVAIL, AVAIL $\leftarrow$ LINK(AVAIL).
Otherwise set POOLMAX $\leftarrow$ POOLMAX $+ c$; (7)
now if POOLMAX $\geq$ SEQMIN, then OVERFLOW; otherwise set X $\leftarrow$ POOLMAX."

c) When other parts of the program attempt to decrease the value of SEQMIN, they should sound the OVERFLOW alarm if SEQMIN $\leq$ POOLMAX.
d) The operation AVAIL $\Leftarrow$ X is unchanged from (5).

This idea actually represents little more than the previous method with a special recovery procedure substituted for the OVERFLOW situation in (6). The net effect is to keep the storage pool as small as possible. Many people like to use this idea even when *all* lists occupy the storage pool area (so that SEQMIN is constant), since it avoids the rather time-consuming operation of initially linking all available cells together and it sometimes facilitates debugging.

We now see that it is quite easy to maintain a "pool" of available nodes, in such a way that free nodes can be efficiently found and later returned. These methods give us a source of raw material to use in linked tables. Our discussion was predicated on the implicit assumption that all nodes have a fixed size, c; the cases which arise when different sizes of nodes are present are very important, but we will defer that discussion until Section 2.5. Now we will consider a few of the most common list operations in the special case where stacks and queues are involved.

A stack is the simplest kind of linked list. Figure 5 shows a typical stack, with a pointer T to the top of the stack. When the stack is empty, this pointer will have the value Λ.

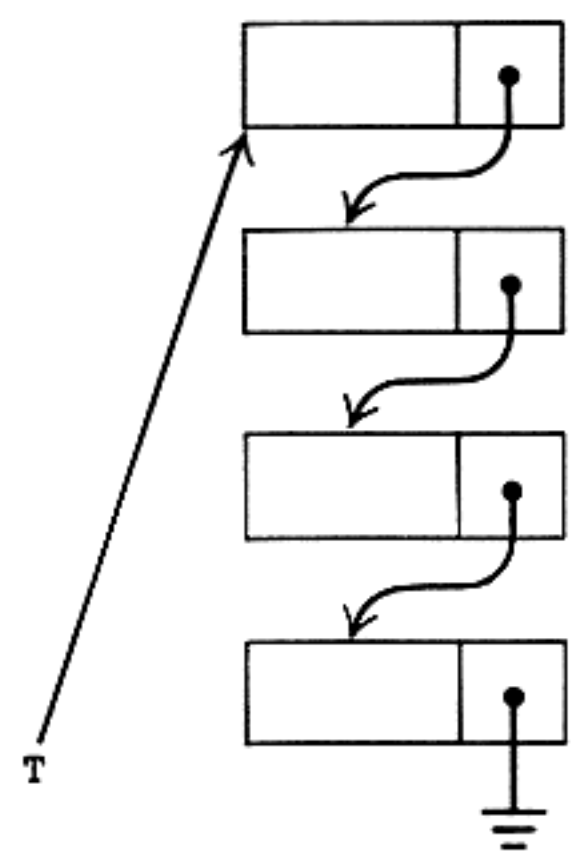


Fig. 5. A linked stack.

It is clear how to insert ("push down") the information Y onto the top of the stack, using an auxiliary pointer variable P:

$$\texttt{P} \Leftarrow \texttt{AVAIL}, \qquad \texttt{INFO(P)} \leftarrow \texttt{Y}, \qquad \texttt{LINK(P)} \leftarrow \texttt{T}, \qquad \texttt{T} \leftarrow \texttt{P}. \tag{8}$$

Conversely, to set Y equal to the information at the top of the stack and to "pop up" the stack:

If $\texttt{T} = \Lambda$, then UNDERFLOW;
otherwise set $\texttt{P} \leftarrow \texttt{T}$, $\texttt{T} \leftarrow \texttt{LINK(P)}$, $\texttt{Y} \leftarrow \texttt{INFO(P)}$, $\texttt{AVAIL} \Leftarrow \texttt{P}$. (9)

These operations should be compared with the analogous mechanisms for sequentially allocated stacks, (2a) and (3a) in Section 2.2.2. The reader should study (8) and (9) carefully, since they are extremely important operations.

Before looking at the case of queues, let us see how these operations can be expressed conveniently in programs for MIX. A program for insertion, with $\texttt{P} \equiv$ rI1, can be written as follows:

```
INFO  EQU  0:3              (Definition of INFO field)
LINK  EQU  4:5              (Definition of LINK field)
      LD1  AVAIL            P ← AVAIL.        }
      J1Z  OVERFLOW         Is AVAIL = Λ?     } P ⇐ AVAIL
      LDA  0,1(LINK)                          }
      STA  AVAIL            AVAIL ← LINK(P).  }
      LDA  Y                                                  (10)
      STA  0,1(INFO)        INFO(P) ← Y.
      LDA  T
      STA  0,1(LINK)        LINK(P) ← T.
      ST1  T                T ← P.
```

This takes 17 cycles, compared to 12 cycles for the comparable operation with a sequential table (although OVERFLOW in the sequential case would in many cases take considerably longer). In this program, as in others to follow in this chapter, OVERFLOW denotes either an ending routine or a *subroutine* which finds more space and returns to location (rJ) − 2.

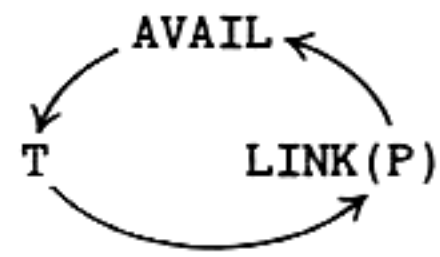
A program for deletion is equally simple:

```
LD1  T                P ← T.
J1Z  UNDERFLOW        Is T = Λ?
LDA  0,1(LINK)
STA  T                T ← LINK(P).
LDA  0,1(INFO)                                            (11)
STA  Y                Y ← INFO(P).
LDA  AVAIL                                }
STA  0,1(LINK)        LINK(P) ← AVAIL.    } AVAIL ⇐ P
ST1  AVAIL            AVAIL ← P.          }
```

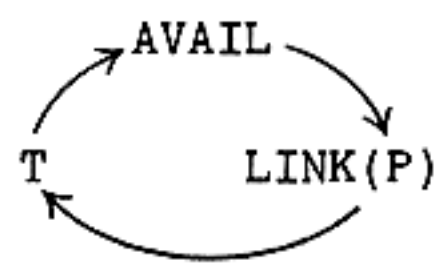
It is worth while to observe that each of these operations involves a cyclic permutation of three links. For example, in the insertion operation let P be the value of AVAIL before the insertion; if P $\neq \Lambda$, we find that after the operation

the value of AVAIL has become the previous value of LINK(P),
the value of LINK(P) has become the previous value of T; and
the value of T has become the previous value of AVAIL.

So the insertion process (except for setting INFO(P) $\leftarrow$ Y) is the cyclic permutation



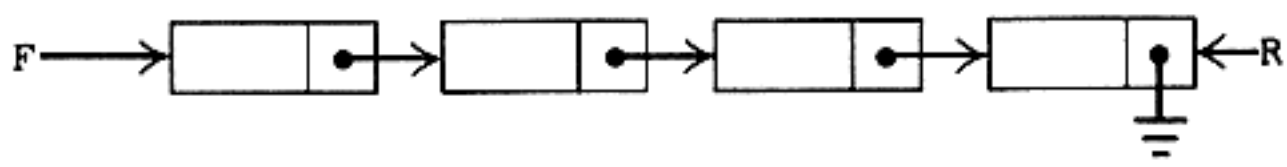
Similarly in the case of deletion, where P has the value of T before the operation and we assume that P $\neq \Lambda$, we have Y $\leftarrow$ INFO(P) and



In these diagrams the fact that the permutation is cyclic is not really a relevant issue, since *any* permutation on three elements that moves every element is cyclic. The important point is rather that precisely three links are permuted in these operations.

The above insertion and deletion algorithms have been described for stacks, but they apply much more generally to insertion and deletion in *any* linear list. Insertion, for example, is performed just before the node pointed to by link variable T. The insertion of item $2\frac{1}{2}$ above [see (2)] would be done by using operation (8) with T = LINK(LINK(FIRST)).

Linked allocation applies in a particularly convenient way to queues. In this case it is easy to see that the links should run from the front of the queue towards the rear, so that when a node is removed from the front, the new front node is directly specified. We will make use of pointers F and R, to the front and rear:



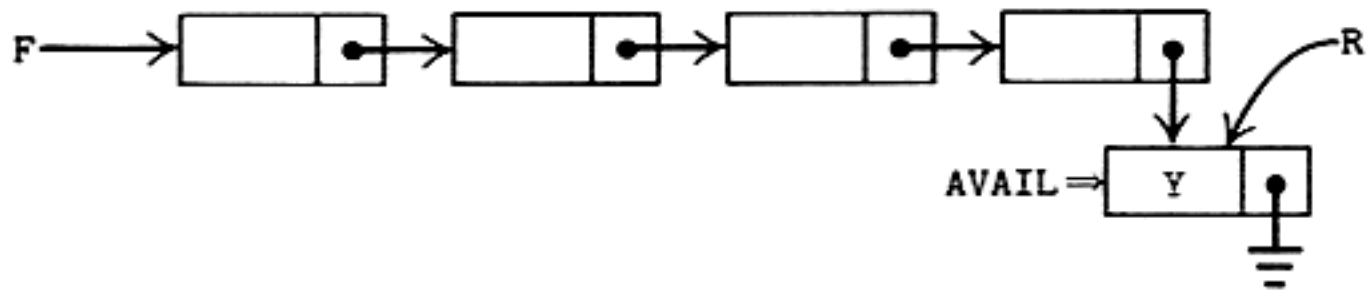
(12)

Except for R, this diagram is abstractly identical to Fig. 5.

Whenever the layout of a list is designed, it is important to specify all conditions carefully, particularly for the case when the list is empty. A failure to do things properly for the case of empty lists is one of the most common programming errors met in connection with linked allocation; the other common error is forgetting to set all of the links when the structure is being manipulated. In order to avoid the first type of error, always examine the "boundary conditions" carefully. To avoid making the second type of error, it is helpful to draw

"before and after" diagrams and to compare them to see which links need to be changed.

Let us illustrate the remarks of the preceding paragraph by applying them to the case of queues. First consider the insertion operation: if (12) is the situation before insertion, the picture after insertion at the rear of the queue should be

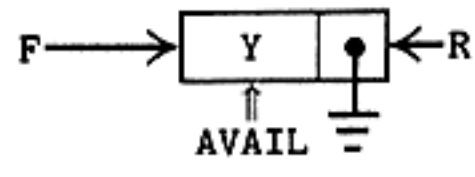


(13)

(The notation used here implies that a new node is obtained from the AVAIL list.) Comparing (12) and (13) shows us how to proceed when inserting the information Y at the rear of the queue:

$$\texttt{P} \Leftarrow \texttt{AVAIL}, \quad \texttt{INFO(P)} \leftarrow \texttt{Y}, \quad \texttt{LINK(P)} \leftarrow \Lambda, \quad \texttt{LINK(R)} \leftarrow \texttt{P}, \quad \texttt{R} \leftarrow \texttt{P}. \tag{14}$$

Let us now consider the "boundary" situation when the queue is empty: in this case the situation before insertion is yet to be determined, and the situation "after" is

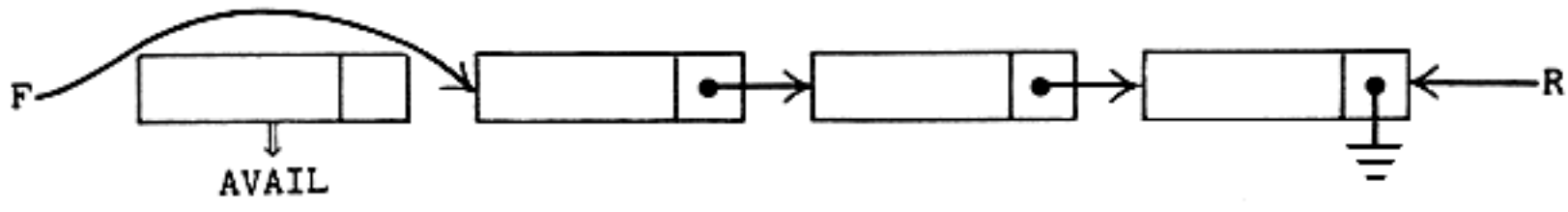


(15)

It is desirable to have operations (14) apply in this case also, even if insertion into an empty queue means that we must change *both* F and R, not only R. We find that (14) will work properly if R = LOC(F) when the queue is empty, *assuming that* F = LINK(LOC(F)); the value of variable F must be stored in the LINK field of its location if this idea is to work. In order to make the testing for an empty queue as efficient as possible, we will let F = Λ in this case. Our policy is therefore that

an empty queue is represented by F = Λ *and* R = LOC(F).

The deletion operation for queues is derived in a similar fashion. If (12) is the situation before deletion, the situation afterward is



(16)

For the boundary conditions we must make sure the deletion operation works when the queue is empty either before or after the operation. These considerations lead us to the following way to do a deletion in general:

If F = Λ, then UNDERFLOW;
otherwise set P ← F, F ← LINK(P), Y ← INFO(P), AVAIL ⇐ P, (17)
and if F = Λ, then set R ← LOC(F).

Note that R must be changed when the queue becomes empty; this is precisely the type of "boundary condition" we should always be watching for.

The above suggestions are not the only way to represent queues in a linearly-linked fashion; we will give other methods later in this chapter. Indeed, none of the operations above are meant to be prescribed as the only way to do something; they are intended as examples of the basic means of operating with linked lists. The reader who has had only a little previous experience with such techniques will find it helpful to reread the present section up to this point before going on.

So far in this chapter we have discussed methods of performing certain operations on tables, but our discussions have always been "abstract" in the sense that we never exhibited actual programs in which the particular techniques were useful. A person is not motivated to study abstractions of a problem until he has seen enough special instances of the problem to arouse his interest. The operations discussed so far (manipulation of variable size lists of information by insertion and deletion, and the use of tables as stacks or queues) are of such wide application, it is hoped that the reader will have encountered them often enough in his own programs that he is already willing to grant their importance. But now we will leave the realm of the abstract as we begin to study a series of significant practical examples of the techniques of this chapter.

Our first example is a problem called *topological sorting*, which is an important process needed in connection with network problems, with so-called PERT charts, and even with linguistics; in fact, it is of potential use whenever we have a problem involving a *partial ordering*. A "partial ordering" of a set S is a relation between the objects of S, which we may denote by the symbol "$\preceq$", satisfying the following properties for any objects x, y, and z (not necessarily distinct) in S:

i) If $x \preceq y$ and $y \preceq z$, then $x \preceq z$. (Transitivity.)

ii) If $x \preceq y$ and $y \preceq x$, then $x = y$. (Antisymmetry.)

iii) $x \preceq x$. (Reflexivity.)

The notation $x \preceq y$ may be read "x precedes or equals y." If $x \preceq y$ and $x \neq y$, we write $x \prec y$ and say "x precedes y." It is easy to see from (i), (ii), and (iii) that we always have

i′) If $x \prec y$ and $y \prec z$, then $x \prec z$. (Transitivity.)

ii′) If $x \prec y$, then $y \not\prec x$. (Asymmetry.)

iii′) $x \not\prec x$. (Irreflexivity.)

The relation denoted by $y \not\prec x$ means "y does not precede x." If we start with a relation $\prec$ satisfying properties (i′), (ii′), and (iii′), we can reverse the above process and define $x \preceq y$ if $x \prec y$ or $x = y$; then properties (i), (ii), and (iii) are true. Therefore we may regard either properties (i), (ii), (iii) or properties

(i′), (ii′), (iii′) as the definition of partial order. [Note that property (ii′) is actually a consequence of (i′) and (iii′).]

Partial orderings occur quite frequently in everyday life as well as in mathematics. As examples from mathematics we can mention the relation $x \leq y$ between real numbers x and y; the relation $x \subseteq y$ between sets of objects; the relation $x \backslash y$ (x divides y) between positive integers. In the case of PERT networks, S is a set of jobs that must be done, and the relation "$x \prec y$" means "x must be done before y."

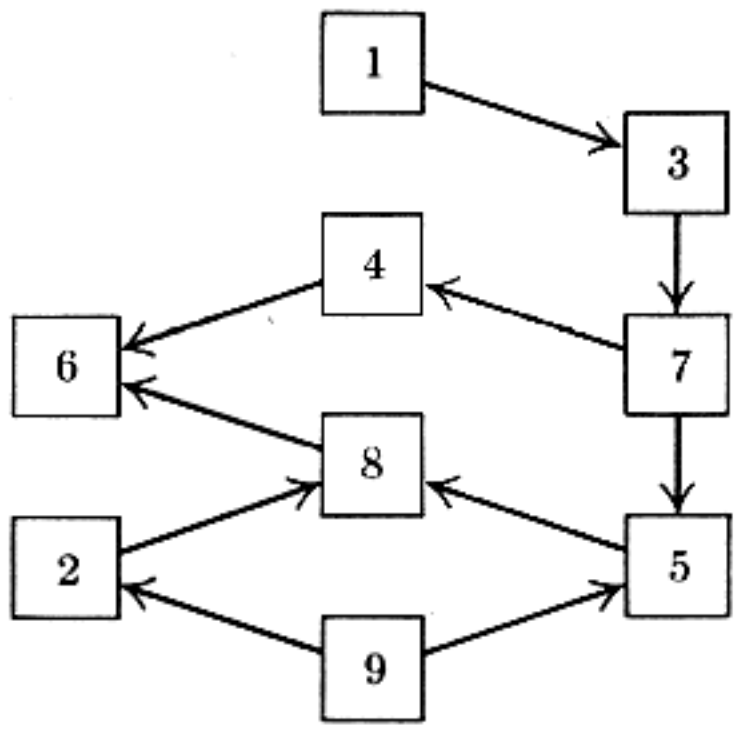


Fig. 6. A partial ordering.

We will naturally assume that S is a finite set, since we want to work with it inside a computer. A partial ordering on a finite set can always be illustrated by drawing a diagram such as Fig. 6, in which the objects are represented by small boxes and the relation is represented by arrows between these boxes; $x \prec y$ means there is a path from the box labeled x to box y which follows the direction of the arrows. Property (ii) of partial ordering means there are *no closed loops* (no paths that close on themselves) in the diagram. If an arrow were drawn from 4 to 1 in Fig. 6, we would no longer have a partial ordering.

The problem of topological sorting is to "embed the partial order in a linear order," i.e., to arrange the objects into a linear sequence $a_1, a_2, \ldots, a_n$ such that whenever $a_j \prec a_k$, we have $j < k$. Graphically, this means that the boxes are to be rearranged into a line so that all arrows go towards the right (see Fig. 7). It is not immediately obvious that such a rearrangement is possible in every case, although such a rearrangement certainly could not be done if any loops were present. Therefore the algorithm we will give is interesting not only because it does a useful operation, but also because it proves that this operation is *possible* for every partial ordering.

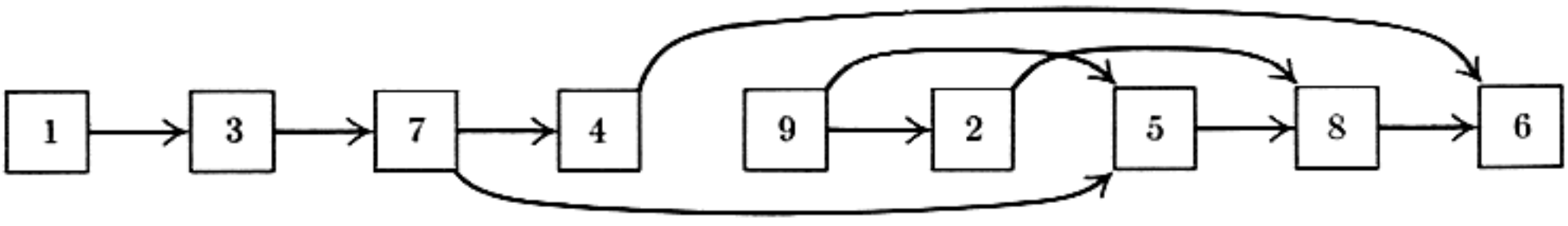


Fig. 7. The ordering relation of Fig. 6 after topological sorting.

As an example of topological sorting, imagine a large glossary containing definitions of technical terms. We can write $w_2 \prec w_1$ if the definition of word w_1 depends directly or indirectly on that of word w_2. This relation is a partial ordering provided that there are no "circular" definitions. The problem of topological sorting in this case is to *find a way to arrange the words in the glossary so that no term is used before it has been defined.* Analogous problems arise in writing programs to process the declarations in certain assembly and compiler languages; they also arise in writing a user's manual describing a computer language or in writing textbooks about information structures.

There is a very simple way to do topological sorting: We start by taking an object which is not preceded by any other object in the ordering. This object may be placed first in the output. Now we remove this object from the set S; the resulting set is again partially ordered, and the process can be repeated until the whole set has been sorted. For example, in Fig. 6 we could start by removing 1 or 9; after 1 has been removed, 3 can be taken, and so on. The only way in which this algorithm could fail would be if there were a nonempty partially ordered set in which every element was preceded by another; for in such a case the algorithm would find nothing to do. But if every element is preceded by another, we could construct an arbitrarily long sequence $b_1, b_2, b_3, \ldots$ in which $b_{j+1} \prec b_j$; since S is finite, we must have $b_j = b_k$ for some $j < k$, but this implies that $b_k \preceq b_{j+1}$, contradicting (ii).

In order to implement this process efficiently by computer, we need to be ready to perform the actions described above, i.e., to locate objects which are not preceded by any others, and to remove them from the set. Our implementation is also influenced by the desired input and output characteristics. The most general program would accept alphabetic names for the objects and would allow for huge numbers of objects to be sorted—more than could possibly fit in the computer memory at once. These complications would obscure the main points we are trying to make here, however; the handling of alphabetic data can be done efficiently by using the methods of Chapter 6, and the handling of large networks is left as an interesting project for the reader.

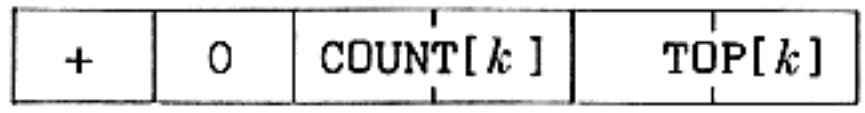
Therefore we will assume that the objects to be sorted are numbered from 1 to n in any order. The input to the program will be on tape unit 1: each tape record contains 50 pairs of numbers; the pair (j, k) means object j precedes object k. The first pair, however, is $(0, n)$, where n is the number of objects. The pair (0, 0) terminates the input. We shall assume that n plus the number of relation pairs will fit comfortably in memory; and we shall assume that it is not necessary to check the input for validity. The output is to be the numbers of the objects in sorted order, followed by the number 0, on tape unit 2.

As an example of the input, we might have the pairs

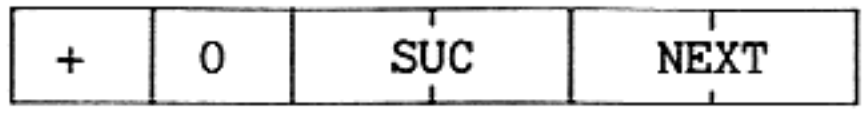
$$\begin{gathered} 9 \prec 2, \quad 3 \prec 7, \quad 7 \prec 5, \quad 5 \prec 8, \\ 8 \prec 6, \quad 4 \prec 6, \quad 1 \prec 3, \\ 7 \prec 4, \quad 9 \prec 5, \quad 2 \prec 8. \end{gathered} \tag{18}$$

It is not necessary to give any more pairs than are needed to characterize the desired partial ordering. Thus additional relations like $9 \prec 8$ (which can be deduced from $9 \prec 5$ and $5 \prec 8$) may be omitted from or added to the input without harm. In general, it is only necessary to give the pairs corresponding to arrows on a diagram such as Fig. 6.

The algorithm which follows uses a sequential table X[1], X[2], . . . , X[n], and each node X[k] has the form



Here COUNT[k] is the *number of direct predecessors* of object k (i.e., the number of pairs $j \prec k$ which have appeared in the input), and TOP[k] is a link to the beginning of the *list of direct successors* of object k. The latter list contains entries in the format



where SUC is a direct successor of k and NEXT is the next item of the list. As an example of these conventions, Fig. 8 shows the schematic contents of memory corresponding to the input (18).

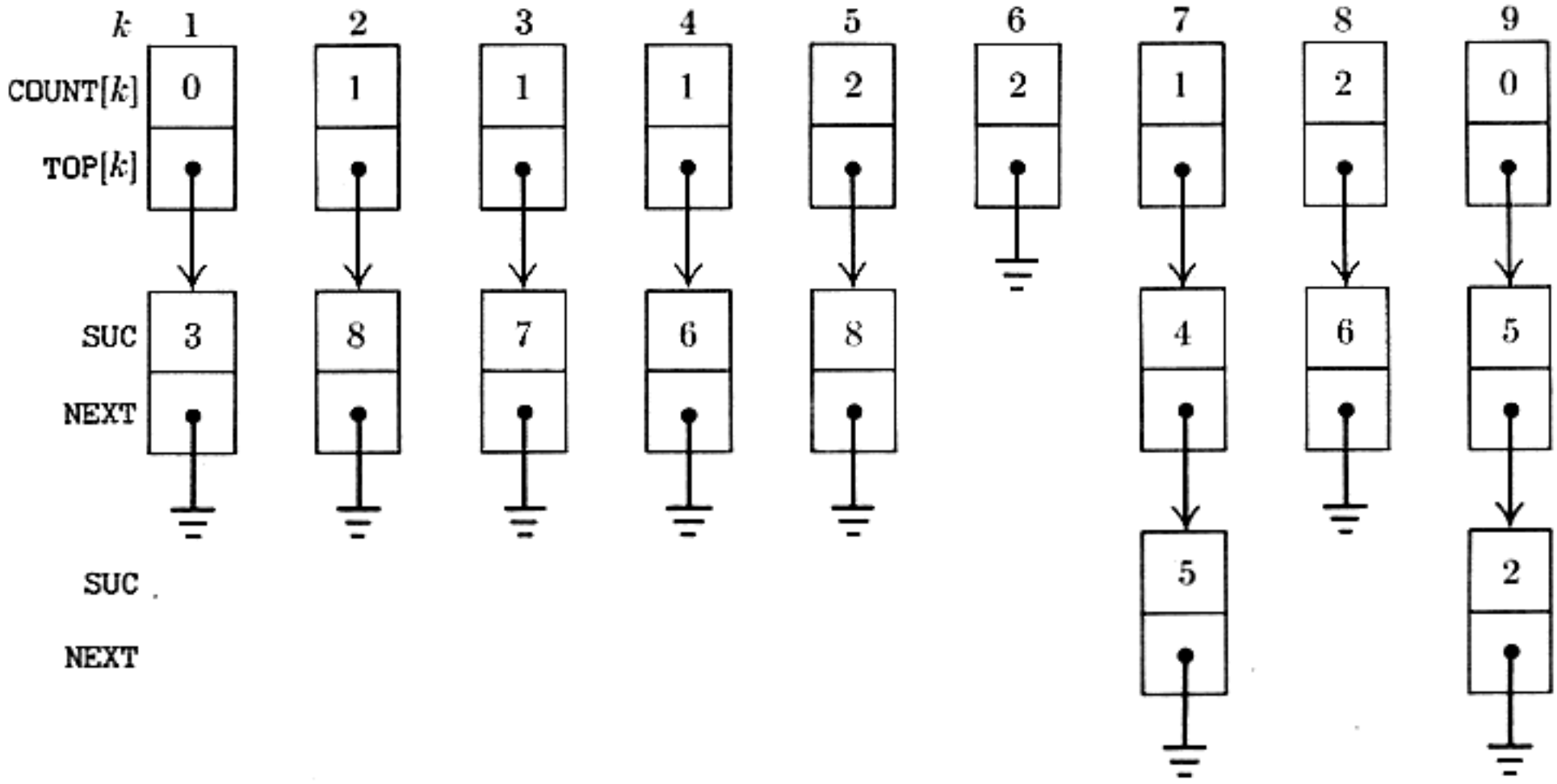


Fig. 8. Computer representation of Fig. 6 corresponding to the relations (18).

Using this memory layout, it is not difficult to work out the algorithm. It is a matter of outputting the nodes whose COUNT field is zero, then decreasing the COUNT fields of all successors of those nodes by one. The trick is to avoid doing any "searching" for nodes whose COUNT field is zero, and this can be done by maintaining a queue containing those nodes whose COUNT field has been reduced to zero but which have not yet been output. The links for this queue

are kept in the COUNT field, which by now has served its previous purpose; for clarity in the algorithm below, we use the notation QLINK[k] to stand for COUNT[k] when that field is no longer being used to keep a count.

Algorithm T (*Topological sort*). This algorithm inputs pairs of relations "$j \prec k$", $1 \le j, k \le n$, indicating that object j precedes object k in a certain partial ordering. The output is the set of objects embedded in linear order. The internal tables used are QLINK[0], COUNT[1] = QLINK[1], COUNT[2] = QLINK[2], ..., COUNT[n] = QLINK[n]; TOP[1], TOP[2], ..., TOP[n]; a storage pool with one node for each input relation and with SUC and NEXT fields as shown above; P, a link variable used to refer to the nodes in the storage pool; F and R, integer-valued variables used to refer to the front and rear of a queue whose links are in the QLINK table; and N, a variable which counts how many objects have yet to be output.

T1. [Initialize.] Input the value of n. Set COUNT[k] $\leftarrow 0$ and TOP[k] $\leftarrow \Lambda$ for $1 \le k \le n$. Set N $\leftarrow n$.

T2. [Next relation.] Get the next relation "$j \prec k$" from the input; if the input has been exhausted, however, go to T4.

T3. [Record the relation.] Increase COUNT[k] by one. Set

$$\texttt{P} \Leftarrow \texttt{AVAIL},\ \texttt{SUC(P)} \leftarrow k,\ \texttt{NEXT(P)} \leftarrow \texttt{TOP}[j],\ \texttt{TOP}[j] \leftarrow \texttt{P}.$$

[This is operation (8).] Go to T2.

T4. [Scan for zeros.] (At this point we have completed the input phase; the input (18) would now have been transformed into the computer representation shown in Fig. 8. Now we initialize the queue of output, which is linked together in the QLINK field.) Set R $\leftarrow 0$ and QLINK[0] $\leftarrow 0$. For $1 \le k \le n$ examine COUNT[k], and if it is zero, set QLINK[R] $\leftarrow k$ and R $\leftarrow k$. After this has been done for all k, set F $\leftarrow$ QLINK[0] (which will contain the first value k encountered for which COUNT[k] was zero).

T5. [Output front of queue.] Output the value of F. If F $= 0$, go to T8; otherwise, set N $\leftarrow$ N $- 1$, and set P $\leftarrow$ TOP[F].

T6. [Erase relations.] If P $= \Lambda$, go to T7. Otherwise decrease COUNT[SUC(P)] by one, and if it has thereby gone down to zero, set QLINK[R] $\leftarrow$ SUC(P) and R $\leftarrow$ SUC(P). Set P $\leftarrow$ NEXT(P) and repeat this step. (We are removing all relations of the form "F $\prec k$" for some k from the system, and putting new nodes into the queue when all their predecessors have been output.)

T7. [Remove from queue.] Set F $\leftarrow$ QLINK[F] and go back to T5.

T8. [End of process.] The algorithm terminates. If N $= 0$, we have output all of the object numbers in the desired "topological order," followed by a zero. Otherwise the N object numbers not yet output contain a loop, in violation of the hypothesis of partial order. (See exercise 23 for an algorithm which prints out the contents of one such loop.) ∎

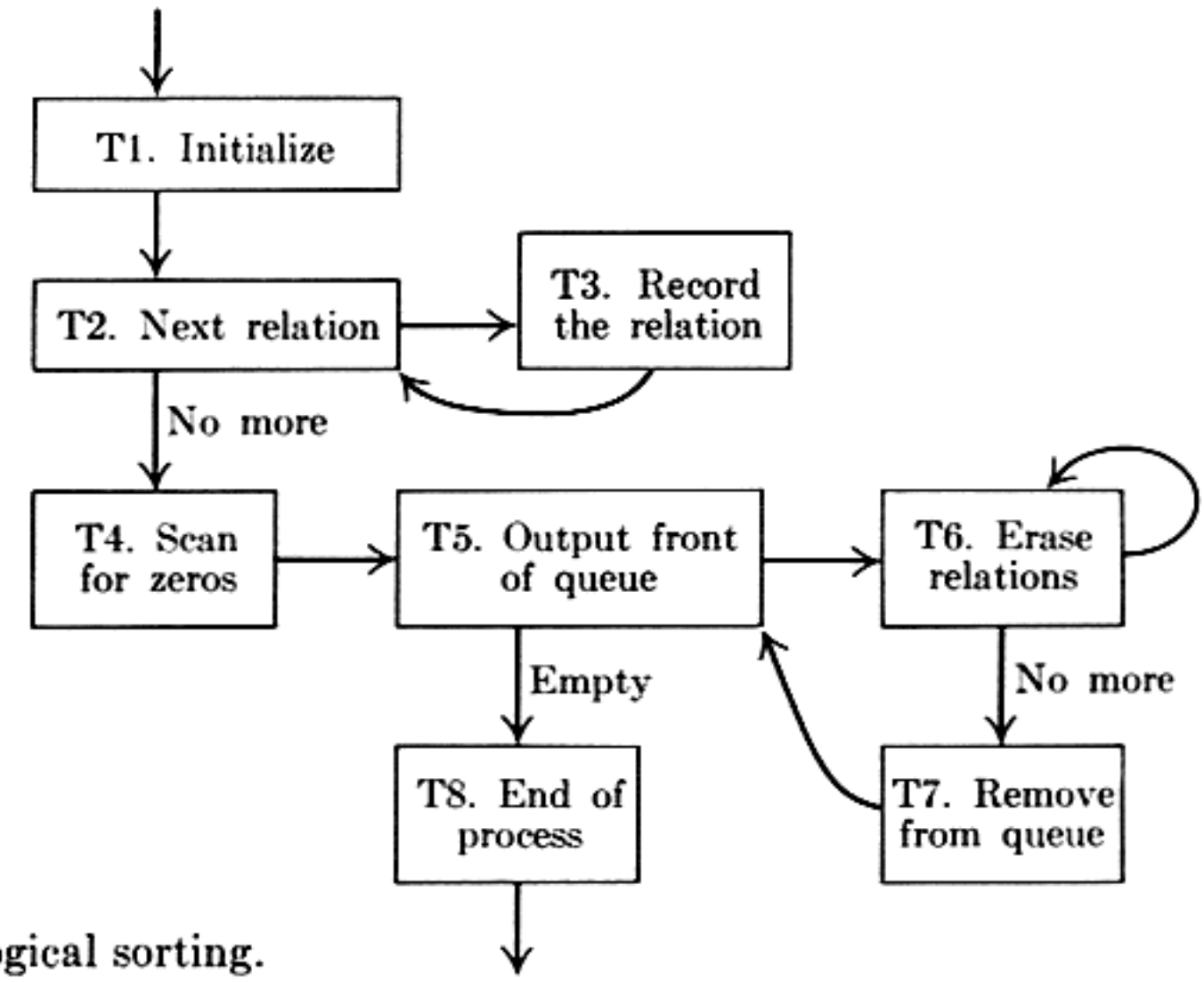


Fig. 9. Topological sorting.

The reader will find it helpful to try this algorithm by hand on the input (18). Algorithm T shows a nice interplay between sequential memory and linked memory techniques. Sequential memory is used for the main table `X[1]`, . . . , `X[`n`]`, which contains the `COUNT[`k`]` and `TOP[`k`]` entries, because we want to make references to "random" parts of this table in step T3. (If the input were alphabetic, however, another type of table would be used for speedier search, as in Chapter 6.) Linked memory is used for the tables of "immediate successors," since these table entries come in random order in the input. The queue of nodes waiting to be output is kept in the midst of the sequential table by linking the nodes together in output order. This linking is done by table index instead of by address; i.e., when the front of the queue is `X[`k`]`, we have `F` $= k$ instead of `F` $=$ `LOC(X[`k`])`. The queue operations used in steps T4, T6, and T7 are not identical to those in (14) and (17), since we are taking advantage of special properties of the queue in this system; no nodes need to be created or returned to available space during this part of the algorithm.

The coding of Algorithm T in `MIX` assembly language has a few more points of interest. Since no deletion from tables is made in the algorithm (because no storage must be freed for later use), the `P` $\Leftarrow$ `AVAIL` operation can be done in an extremely simple way, as shown in lines 19 and 32 below; we need not keep any linked pool of memory, we can choose new nodes consecutively. The program includes complete input and output with magnetic tape, according to the conventions mentioned above, except that for simplicity no buffering is shown. The reader should not find it very difficult to follow the details of the coding in this program, since it corresponds directly with Algorithm T.. The efficient use of index registers, which is an important aspect of linked memory processing, is illustrated here.

Program T (*Topological sort*). In this program, the following equivalences should be noted: rI6 ≡ N, rI5 ≡ buffer pointer, rI4 ≡ k, rI3 ≡ j and R, rI2 ≡ AVAIL and P, rI1 ≡ F, TOP[j] ≡ X + j(4:5), COUNT[k] ≡ QLINK[k] ≡ X + k(2:3).

```
01      *  BUFFER AREA AND FIELD DEFINITIONS
02  COUNT    EQU   2:3                                  Definition of symbolic
03  QLINK    EQU   2:3                                      names of fields
04  TOP      EQU   4:5
05  SUC      EQU   2:3
06  NEXT     EQU   4:5
07  TAPEIN   EQU   1                                    Input is on tape unit 1
08  TAPEOUT  EQU   2                                    Output is on tape unit 2
09  BUFFER   ORIG  *+100                                Tape buffer area
10           CON   -1                                   Sentinel at end of buffer
11      *  INPUT PHASE
12  TOPSORT  IN    BUFFER(TAPEIN)          1           T1. Initialize. Read in first
13           JBUS  *(TAPEIN)                                tape block, wait for completion.
14  1H       LD6   BUFFER+1                 1           N ← n.
15           ENT4  0,6                      1
16           STZ   X,4                      n + 1       Set COUNT[k] ← 0, TOP[k] ← Λ,
17           DEC4  1                        n + 1             for 0 ≤ k ≤ n.
18           J4NN  *-2                      n + 1       (Anticipate QLINK[0] ← 0 in step T3.)
19           ENT2  X,6                      1           Available storage starts after X[n].
20           ENT5  BUFFER+2                 1           Prepare to read first pair (j, k).
21  2H       LD3   0,5                      m + b       T2. Next relation.
22           J3P   3F                       m + b       Is j > 0?
23           J3Z   4F                       b           Is input exhausted?
24           IN    BUFFER(TAPEIN)           b − 1       Sentinel sensed, read another
25           JBUS  *(1)                                     tape record, wait for completion.
26           ENT5  BUFFER                   b − 1       Reset buffer pointer.
27           JMP   2B                       b − 1
28  3H       LD4   1,5                      m           T3. Record the relation.
29           LDA   X,4(COUNT)               m           COUNT[k]
30           INCA  1                        m             + 1
31           STA   X,4(COUNT)               m                → COUNT[k].
32           INC2  1                        m           AVAIL ← AVAIL + 1.
33           LDA   X,3(TOP)                 m           TOP[j]
34           STA   0,2(NEXT)                m             → NEXT(P).
35           ST4   0,2(SUC)                 m           k → SUC(P).
36           ST2   X,3(TOP)                 m           P → TOP[j].
37           INC5  2                        m           Increase buffer pointer.
38           JMP   2B                       m
39  4H       IOC   0(TAPEIN)                1           Rewind input tape.
40           ENT4  0,6                      1           T4. Scan for zeros.
41           ENT5  0                        1           Reset buffer pointer for output.
42           ENT3  0                        1           R ← 0.
43  4H       LDA   X,4(COUNT)               n           Examine COUNT[k].
44           JAP   *+3                      n           Is it nonzero?
45           ST4   X,3(QLINK)               a           QLINK[R] ← k.
46           ENT3  0,4                      a           R ← k.
47           DEC4  1                        n
48           J4P   4B                       n           n ≥ k ≥ 1.
```

```
49    *  SORTING PHASE
50             LD1    X(QLINK)              1      F ← QLINK[0].
51    5H       JBUS   *(TAPEOUT)                   T5. Output front of queue.
52             ST1    BUFFER,5              n+1    Store F in buffer area.
53             J1Z    8F                    n+1    Is F zero?
54             INC5   1                     n      Advance buffer pointer.
55             ENTA   -100,5                n
56             JAN    *+3                   n      Test if buffer is full.
57             OUT    BUFFER(TAPEOUT)       c-1    If so, output a tape record.
58             ENT5   0                     c-1    Reset buffer pointer.
59             DEC6   1                     n      N ← N - 1.
60             LD2    X,1(TOP)              n      P ← TOP[F].
61             J2Z    7F                    n      T6. Erase relations.
62    6H       LD4    0,2(SUC)              m      rI4 ← SUC(P).
63             LDA    X,4(COUNT)            m      COUNT[rI4]
64             DECA   1                     m        - 1
65             STA    X,4(COUNT)            m          → COUNT[rI4].
66             JAP    *+3                   m      Has zero been reached?
67             ST4    X,3(QLINK)            n-a    If so, set QLINK[R] ← rI4.
68             ENT3   0,4                   n-a    R ← rI4.
69             LD2    0,2(NEXT)             m      P ← NEXT(P).
70             J2P    6B                    m      If P ≠ Λ, repeat.
71    7H       LD1    X,1(QLINK)            n      T7. Remove from queue.
72             JMP    5B                    n      F set to QLINK[F]
73    8H       OUT    BUFFER(TAPEOUT)       1      T8. End of process.
74             IOC    0(TAPEOUT)            1      Output last block and rewind.
75             HLT    0,6                   1      Stop, displaying N on console.
76    X        EQU    *                            Beginning of table area
77             END    TOPSORT                      ▮
```

The analysis of Algorithm T is quite simple with the aid of Kirchhoff's law; the execution time has the approximate form $c_1m + c_2n$, where m is the number of input relations, n is the number of objects, and c_1 and c_2 are constants. It is hard to imagine a faster algorithm for this problem! The exact quantities in the analysis are given with Program T above, where a = number of objects with no predecessor, b = number of tape records in input = $\lceil (m + 2)/50 \rceil$, and c = number of tape records in output = $\lceil (n + 1)/100 \rceil$. Exclusive of input/output operations, the total running time in this case is only $(32m + 25n + 7b + 2c + 16)u$.

A topological sorting technique similar to Algorithm T (but without the important feature of the queue links) was first published by A. B. Kahn, *CACM* **5** (1962), 558–562.

EXERCISES

▶ **1.** [*10*] Operation (9) for popping up a stack mentions the possibility of `UNDERFLOW`; why doesn't operation (8) for pushing down a stack mention the possibility of `OVERFLOW`?

2. [*22*] Write a "general purpose" MIX subroutine to do the insertion operation, (10). This subroutine should have the following specifications:

Calling sequence: `JMP  INSERT` Jump to subroutine.
`NOP  T` Location of pointer variable
Entry conditions: rA = information to be put into the INFO field of a new node.
Exit conditions: The stack whose pointer is the link variable T has the new node on top; rI1 = T; rI2, rI3 are altered.

3. [*22*] Write a "general purpose" MIX subroutine to do the deletion operation, (11). This subroutine should have the following specifications:

Calling sequence: `JMP  DELETE` Jump to subroutine.
`NOP  T` Location of pointer variable
`JMP  UNDERFLOW` First exit, if UNDERFLOW sensed
Entry conditions: None
Exit conditions: If the stack whose pointer is the link variable T is empty, the first exit is taken; otherwise the top node of that stack is deleted, and exit is made to three locations following "JMP DELETE". In the latter case, rI1 = T and rA is the contents of the INFO field of the deleted node. In either case, rI2 and rI3 are used by this subroutine.

4. [*22*] The program in (10) is based on the operation P ⇐ AVAIL, as given in (6). Show how to write an OVERFLOW subroutine so that, without *any* change in the coding (10), the operation P ⇐ AVAIL makes use of SEQMIN, as given by (7). For general-purpose use, your subroutine should not change the contents of any registers and it should exit to location (rJ − 2), instead of the usual (rJ).

▸ **5.** [*24*] Operations (14) and (17) give the effect of a queue; show how to define the further operation "insert at front" so as to obtain all the actions of an output-restricted deque. How could the operation "delete from rear" be defined (so that we would have a general deque)?

6. [*21*] In operation (14) we set LINK(P) ← Λ, while the very next insertion at the rear of the queue will change the value of this same link field. Show how the setting of LINK(P) in (14) could be eliminated if we make a change to the testing of "F = Λ" in (17).

▸ **7.** [*23*] Design an algorithm to "invert" a linked linear list such as (1), i.e., to change its links so that the items appear in the opposite order. [Thus, if the list (1) were inverted, we would have FIRST linking to the node containing item 5; that node would link to the one containing item 4; etc.] Assume that the nodes have the form (3).

8. [*24*] Write a MIX program for the problem of exercise 7, attempting to design your program to operate as fast as possible.

9. [*20*] Which of the following relations is a partial ordering on the specified set S? [*Note:* If the relation "$x \prec y$" is defined below, the intent is to define the relation "$x \preceq y \equiv (x \prec y \text{ or } x = y)$," and then to determine whether $\preceq$ is a partial ordering.] (a) S = all rational numbers, $x \prec y$ means $x > y$. (b) S = all people, $x \prec y$ means x is an ancestor of y. (c) S = all integers, $x \preceq y$ means x is a multiple of y (that is, $x \bmod y = 0$). (d) S = all the mathematical results proved in this book, $x \prec y$ means the proof of y depends upon the truth of x. (e) S = all positive integers, $x \preceq y$ means

$x + y$ is even. (f) S = a set of subroutines, $x \prec y$ means "x calls y," that is, y may be in operation while x is in operation.

10. [*M21*] Given that "$\subset$" is a relation which satisfies properties (i) and (ii) of a partial ordering, prove that the relation "$\preceq$", defined by the rule "$x \preceq y$ if and only if $x = y$ or $x \subset y$," satisfies all three properties of a partial ordering.

▶ **11.** [*24*] The result of topological sorting is not always completely determined, since there may be several ways to arrange the nodes and to satisfy the conditions of topological order. Find all possible ways to arrange the nodes of Fig. 6 into topological order.

12. [*M20*] There are 2^n subsets of a set of n elements, and these subsets are partially ordered by the set-inclusion relation. Give two interesting ways to arrange these subsets in topological order.

13. [*M48*] How many ways are there to arrange the 2^n subsets described in exercise 12 into topological order? (Give the answer as a function of n.)

14. [*M24*] A *linear ordering* of a set S is a partial ordering which satisfies the additional condition

(iv) For any two objects x, y in S, either $x \preceq y$ or $y \preceq x$.

Prove directly from the definitions given that a topological sort can result in only one possible output if and only if the relation $\preceq$ is a linear ordering. (You may assume that the set S is finite.)

15. [*M25*] Show that for any partial ordering on a finite set S there is a *unique* set of irredundant pairs of relations [such as (18) corresponding to Fig. 6] which characterizes this ordering. Is the same fact true also when S is an infinite set?

16. [*M22*] Given any partial ordering on a set $S = \{x_1, \ldots, x_n\}$, we can construct its "incidence matrix" (a_{ij}), where $a_{ij} = 1$ if $x_i \preceq x_j$, and $a_{ij} = 0$ otherwise. Show that there is a way to permute the rows and columns of this matrix so that all entries below the diagonal are zero.

▶ **17.** [*21*] What output does Algorithm T produce if it is presented with the input (18)?

18. [*20*] What, if anything, is the significance of the values of `QLINK`[0], `QLINK`[1], . . . , `QLINK`[n] when Algorithm T terminates?

19. [*18*] In Algorithm T we examine the front position of the queue in step T5, but do not remove that element from the queue until step T7. What would happen if we set `F` ← `QLINK`[`F`] at the conclusion of step T5, instead of in T7?

▶ **20.** [*24*] Algorithm T uses `F`, `R`, and the `QLINK` table to obtain the effect of a queue which contains those nodes whose `COUNT` field has become zero but whose successor relations have not yet been removed. Could a stack be used for this purpose instead of a queue? If so, compare the resulting algorithm with Algorithm T.

21. [*21*] Would Algorithm T still perform a valid topological sort if one of the relations "$j \prec k$" were repeated several times in the input? What if the input contained a relation of the form "$j \prec j$"?

22. [*23*] Program T assumes that its input tape contains valid information, but a program that is intended for general use should always make careful tests on its input so that clerical errors can be detected, and the program cannot "destroy itself." For

example, if one of the input relations for k were negative, Program T may erroneously change one of its own instructions when storing into X[k]. Suggest ways to modify Program T so that it is suitable for general use.

▸ **23.** [*27*] When the topological sort algorithm cannot proceed because it has detected a loop in the input (see step T8), it is usually of no use just to stop and say, "There was a loop." It is helpful to print out one of the loops, thereby showing part of the input which was in error. Extend Algorithm T so that it will do this additional printing of a loop when necessary. [*Hint:* The text gives a proof for the existence of a loop when N $>$ 0 in step T8; that proof suggests an algorithm.]

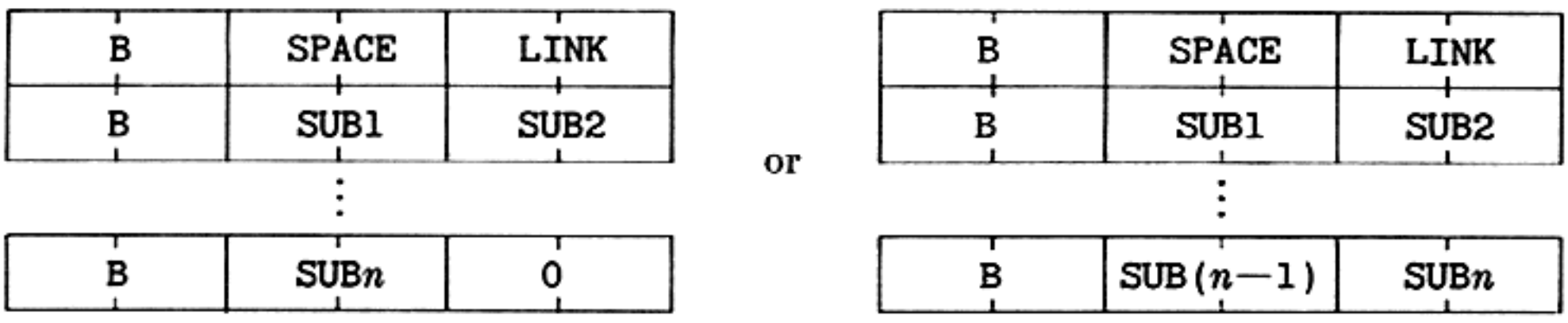
24. [*24*] Incorporate the extensions of Algorithm T made in exercise 23 into Program T.

25. [*44*] Design as efficient an algorithm as possible for doing a topological sort of very large sets S, which have considerably more nodes than the computer memory can contain. Assume that the input, output, and temporary working space are done with magnetic tape. [*Possible hint:* A conventional sort of the input allows us to assume that all relations for a given node appear together. But then what can be done? (In particular, we must consider the worst case in which the given ordering is already a linear ordering that has been wildly permuted; if possible we want to avoid doing $O(n)$ iterations through the entire data tape.)]

26. [*29*] (Subroutine allocation.) Suppose that we have a tape containing the main "subroutine library" for a computer installation in relocatable form. The loading routine wants to determine the amount of relocation for each subroutine used so it can make one pass through the tape to load the necessary routines. The problem is that some subroutines require others to be present in memory. Infrequently used subroutines (which appear toward the end of the tape) may call on frequently used subroutines (which appear toward the beginning of the tape), and we want to know all of the subroutines which are required, before passing through the tape.

One way to tackle this problem is to have a "tape directory" which fits in memory. The loading routine has access to two tables:

a) The tape directory. This table is composed of variable-length nodes having the form



where SPACE is the number of words of memory required by the subroutine; LINK is a link to the directory entry for the subroutine which appears on the tape following this subroutine; SUB1, SUB2, . . . , SUBn ($n \geq 0$) are links to the directory entries for any other subroutines required by this one; B = 0 on all words except the last, B = −1 on the last word of a node. The address of the directory entry for the first subroutine on the library tape is specified by the link variable FIRST.

b) The list of subroutines directly referred to by the program to be loaded. This is stored in consecutive locations X[1], X[2], . . . , X[N], where N $\geq$ 0 is a variable known

to the loading routine. Each entry in this list is a link to the directory entry for the subroutine desired.

The loading routine also knows MLOC, the amount of relocation to be used for the first subroutine loaded.

As a small example, consider the following configuration:

Tape directory

	B	SPACE	LINK
1000:	0	20	1005
1001:	−1	1002	0
1002:	−1	30	1010
1003:	0	200	1007
1004:	−1	1000	1006
1005:	−1	100	1003
1006:	−1	60	1000
1007:	0	200	0
1008:	0	1005	1001
1009:	−1	1006	0
1010:	−1	20	1006

List of subroutines needed

X[1] = 1003
X[2] = 1010

N = 2
FIRST = 1002
MLOC = 2400

The tape directory in this case shows that the subroutines on tape are 1002, 1010, 1006, 1000, 1005, 1003, and 1007 in that order. Subroutine 1007 takes 200 locations and implies the use of subroutines 1005, 1001, and 1006; etc. The program to be loaded requires subroutines 1003 and 1010, which are to be placed into locations $\geq$ 2400. These subroutines in turn imply that 1000, 1006, and 1002 must also be loaded.

The subroutine allocator is to change the X-table so that each entry X[1], X[2], . . . has the form

+	0	BASE	SUB

(except the last entry which is explained below), where SUB is a subroutine to be loaded, and BASE is the amount of relocation. These entries are to be in the order in which the subroutines appear on tape. In the above example one possible answer would be

	BASE	SUB
X[1]:	2400	1002
X[2]:	2430	1010
X[3]:	2450	1006
X[4]:	2510	1000
X[5]:	2530	1003
X[6]:	−2730	0

Note that the last entry contains the negative of the first unused memory address.

(Clearly, this is not the only way to treat a library of subroutines. The proper way to design a library is heavily dependent upon the computer used and the applications to be handled. Several modern computers require an entirely different approach to subroutine libraries.)

The problem in this exercise is to design an algorithm for this subroutine allocation task. The subroutine allocator may transform the tape directory in any way as it prepares its answer, since the tape directory can be read in anew by the subroutine allocator on its next assignment, and the tape directory is not needed by other parts of the loading routine.

27. [*25*] Write a MIX program for the subroutine allocation algorithm of exercise 26.

28. [*40*] The following construction shows how to "solve" a fairly general type of two-person game, including chess, nim, and many simpler games: Consider a finite set of nodes, each of which represents a possible "position" in the game. For each position there are zero or more "moves" which transform that position into some other position. We say that position x is a predecessor of position y (and y is a successor of x) if there is a move from x to y. Certain positions which have no successors are classified as "won" or "lost" positions. The player to move in position x is the opponent of the player to move in the successors of position x.

Given such a configuration of positions, we can compute the complete set of "won" positions (those in which it is possible for the player to force a victory) and the complete set of "lost" positions (those in which the player must lose against an expert opponent) by repeatedly doing the following operation until it yields no change: mark a position "lost" if all its successors are marked "won"; mark a position "won" if at least one of its successors is marked "lost."

After this operation has been repeated as many times as possible, there may be some positions that have not been marked at all; a player in such a position cannot force a victory, nor can he be compelled to lose.

This procedure for obtaining the complete set of "won" and "lost" positions can be adapted to an efficient algorithm for computers that closely resembles Algorithm T. We may keep with each position a count of the number of its successors that have not been marked "won," and a list of all its predecessors.

The problem in this exercise is to work out the details of the algorithm that has just been so vaguely described, and to apply it to some interesting games that do not involve too many possible positions [like the "military game": *Sci. Am.* (October, 1963), 124, or E. Lucas, *Récréations Mathématiques*, **3** (Paris, 1893) 105–116].

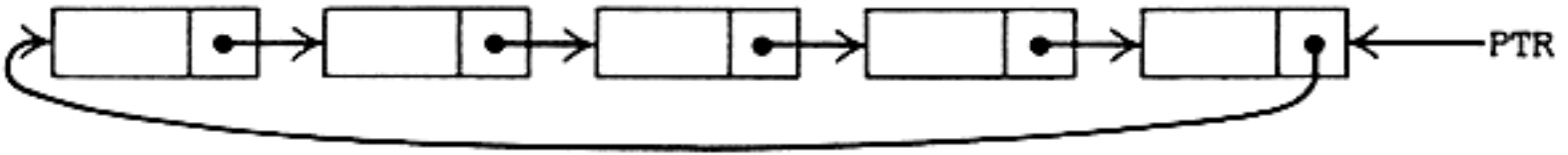
▸ **29.** [*21*] (a) Give an algorithm to "erase" an entire list like (1), i.e., to put all of its nodes on the AVAIL stack, given only the value of FIRST. The algorithm should operate as fast as possible. (b) Repeat part (a) for a list like (12), given the values of F and R.

2.2.4. Circular Lists

A slight change in the manner of linking furnishes us with an important alternative to the methods of the preceding section.

A *circularly-linked list* (briefly: a circular list) has the property that its last node links back to the first instead of to Λ. It is then possible to access all of the list starting at any given point; we also achieve an extra degree of symmetry, and if we choose we need not think of the list as having a "last" or "first" node.

The following situation is typical:



(1)

Assume that the nodes have two fields, INFO and LINK, as in the preceding section. There is a link variable PTR which points to the rightmost node of the list, and LINK(PTR) is the address of the leftmost node. The following primitive operations are most important:

a) Insert Y at left: P ⇐ AVAIL, INFO(P) ← Y, LINK(P) ← LINK(PTR), LINK(PTR) ← P.
b) Insert Y at right: Insert Y at left, then PTR ← P.
c) Set Y to left node and delete: P ← LINK(PTR), Y ← INFO(P), LINK(PTR) ← LINK(P), AVAIL ⇐ P.

Operation (b) is a little surprising at first glance; the operation PTR ← LINK(PTR) effectively moves the leftmost node to the right in the diagram (1), and this is quite easy to understand if the list is regarded as a circle instead of a straight line with connected ends.

The alert reader will observe that we have made a serious mistake in the above operations (a), (b), (c). What is it? *Answer.* We have forgotten to consider the possibility of an *empty* list. If for example operation (c) is applied five times to the list (1), we will have PTR pointing to a node in the AVAIL list, and this can lead to serious difficulties; for example, imagine applying operation (c) *six* times to (1)! If we take the position that PTR will equal Λ in the case of an empty list, we could remedy the above operations by inserting the additional instructions "if PTR = Λ, then PTR ← LINK(P) ← P; otherwise . . ." after "INFO(P) ← Y" in (a) and (b); preceding (c) by the test "if PTR = Λ, then UNDERFLOW"; and following (c) by "if PTR = P, then PTR ← Λ."

Note that the operations (a), (b), and (c) give us the actions of an output-restricted deque, in the sense of Section 2.2.1. Therefore we find in particular that a circular list can be used as either a stack or a queue. Operations (a) and (c) combined give us a stack; operations (b) and (c) give us a queue. These operations are only slightly less direct than their counterparts in the previous section, where we saw that operations (a), (b), and (c) can be performed on linear lists using two pointers F and R.

Other important operations become efficient with circular lists. For example, it is very convenient to "erase" a list, i.e., to put an entire circular list onto the AVAIL stack at once:

$$\text{If} \quad \texttt{PTR} \neq \Lambda, \quad \text{then} \quad \texttt{AVAIL} \leftrightarrow \texttt{LINK(PTR)}. \tag{2}$$

[Recall that the "↔" operation denotes interchange, i.e., P ← AVAIL, AVAIL ← LINK(PTR), LINK(PTR) ← P.] Operation (2) is clearly valid if PTR points *anywhere* in the circular list.

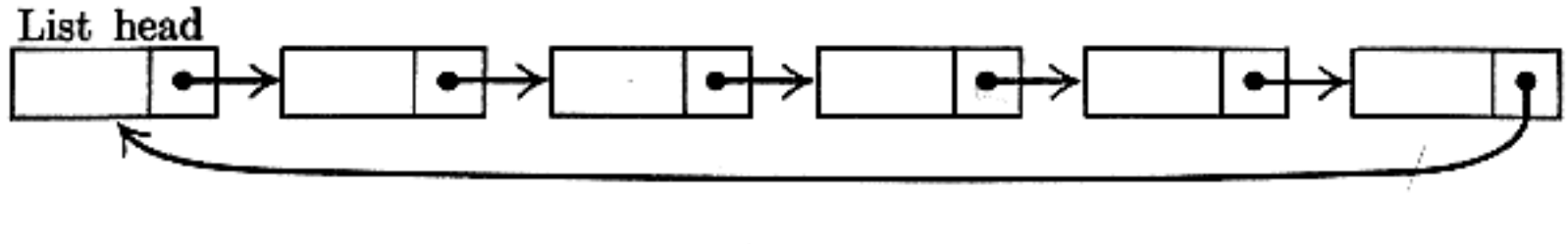
Using a similar technique, if $\texttt{PTR}_1$ and $\texttt{PTR}_2$ point to disjoint circular lists L_1 and L_2, respectively, we can insert the entire list L_2 at the right of L_1:

$$\begin{aligned} &\text{If} \quad \texttt{PTR}_2 \neq \Lambda, \quad \text{then} \\ &(\text{if} \quad \texttt{PTR}_1 \neq \Lambda, \quad \text{then} \quad \texttt{LINK}(\texttt{PTR}_1) \leftrightarrow \texttt{LINK}(\texttt{PTR}_2); \\ &\quad \text{set} \quad \texttt{PTR}_1 \leftarrow \texttt{PTR}_2, \ \texttt{PTR}_2 \leftarrow \Lambda). \end{aligned} \tag{3}$$

Splitting one circular list into two, in various ways, is another simple operation that can be done. These operations correspond to the concatenation and deconcatenation of strings.

Thus we see that a circular list can be used not only to represent inherently circular structures, but also to represent linear structures; a circular list with one pointer to the rear node is essentially equivalent to a straight linear list with two pointers to the front and rear. The natural question to ask, in connection with this observation, is, "How do we find the end of the list, in view of the circular symmetry?" There is no Λ link to signal the end. The answer is that if we are performing some operations while moving through the list from one node to the next, we should stop when we get back to our starting place (assuming, of course, that our starting place is still present in the list).

An alternative solution to the problem just posed is to put a special, recognizable node into each circular list, as a convenient stopping place. This special node is called the *list head*, and in applications we often find it is quite convenient to insist that every circular list have exactly one node which is its list head. One advantage is that the circular list will then never be empty. The diagram (1) now becomes



(4)

Instead of a pointer to the right end of the list, references to lists like (4) are usually made via the list head, which is often in a fixed memory location. In this case, we sacrifice operation (b) stated above.

Diagram (4) may be compared with 2.2.3–(1) at the beginning of the previous section, in which the link associated with "item 5" now points to `LOC(FIRST)` instead of to Λ, and `FIRST` is now thought of as a link within a node, `NODE(LOC(FIRST))`. The principal difference between (4) and 2.2.3–(1) is that with (4) it is possible (though not necessarily efficient) to get to any point of the list from any other point.

As an example of the use of circular lists, we will discuss *arithmetic on polynomials* in the variables x, y, and z, with integer coefficients. There are many problems in which a scientist wants to manipulate polynomials instead of just numbers; we are thinking of operations like the multiplication of

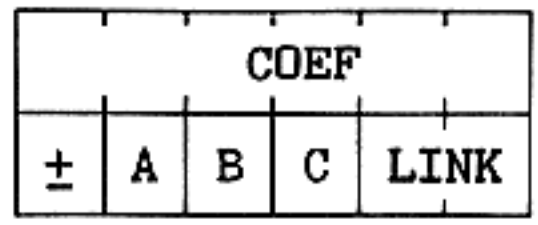
$$(x^4 + 2x^3y + 3x^2y^2 + 4xy^3 + 5y^4) \qquad \text{by} \qquad (x^2 - 2xy + y^2)$$

to get

$$(x^6 - 6xy^5 + 5y^6).$$

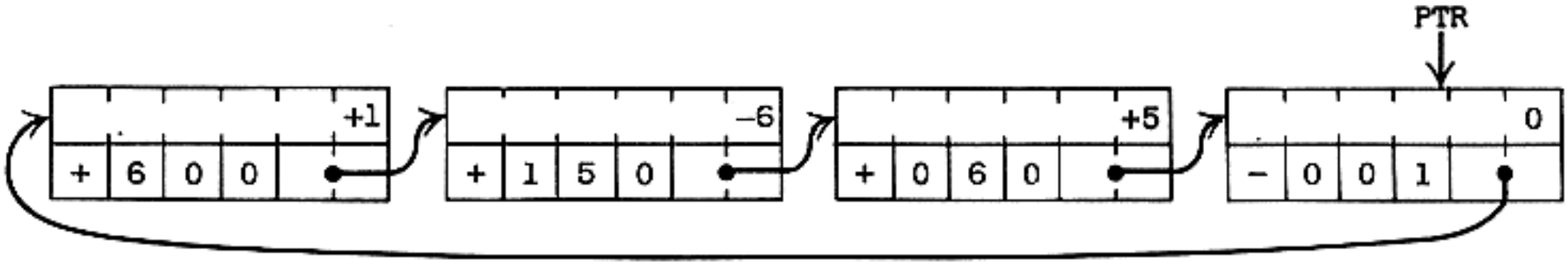
Linked allocation is a natural tool for this purpose, since polynomials can grow to unpredictable sizes and we may want to represent many polynomials in memory at the same time.

We will consider here the two operations of addition and multiplication. Let us suppose that a polynomial is represented as a list in which each node stands for one nonzero term, and has the two-word form



(5)

Here COEF is the coefficient of the term in $x^{\mathtt{A}}y^{\mathtt{B}}z^{\mathtt{C}}$. We will assume that the coefficients and exponents will always lie in the range allowed by this format, and that it is not necessary to check this condition during our calculations. The notation ABC will be used to stand for the ± A B C fields of the node (5), treated as a single unit. The sign of ABC, i.e., the sign of the second word in (5), will always be plus, except that there is a *special node* at the end of every polynomial which has ABC $= -1$ and COEF $= 0$. This special node is a great convenience, analogous to our discussion of a list head above, because it provides a convenient "sentinel" and it avoids the problem of an empty list (corresponding to the polynomial "0"). The nodes of the list always appear in *decreasing order* of the ABC field, if we follow the direction of the links, except that the special node (which has ABC $= -1$) links to the largest value of ABC. For example, the polynomial $x^6 - 6xy^5 + 5y^6$ would be represented thus:



Algorithm A (*Addition of polynomials*). This algorithm adds polynomial(P) to polynomial(Q), assuming that P and Q are pointer variables pointing to polynomials having the form above. The list P will be unchanged, the list Q will retain the sum. Pointer variables P and Q return to their starting points at the conclusion of this algorithm; auxiliary pointer variables Q1 and Q2 are also used.

A1. [Initialize.] Set P ← LINK(P), Q1 ← Q, Q ← LINK(Q). (Now both P and Q point to the leading term of the polynomial. Throughout most of this algorithm the variable Q1 will be "one step behind" Q, in the sense that Q = LINK(Q1).)

A2. [ABC(P):ABC(Q).] If ABC(P) < ABC(Q), set Q1 ← Q and Q ← LINK(Q) and repeat this step. If ABC(P) = ABC(Q), go to step A3. If ABC(P) > ABC(Q), go to step A5.

A3. [Add coefficients.] (We have found terms with equal exponents.) If ABC(P) < 0, the algorithm terminates. Otherwise set COEF(Q) ← COEF(Q) + COEF(P). Now if COEF(Q) = 0, go to A4; otherwise, set Q1 ← Q, P ← LINK(P), Q ← LINK(Q), and go to A2.

A4. [Delete zero term.] Set Q2 ← Q, LINK(Q1) ← Q ← LINK(Q), and AVAIL ⇐ Q2. (A zero term created in step A3 has been removed from polynomial (Q).) Set P ← LINK(P) and go to A2.

A5. [Insert new term.] (Polynomial(P) contains a term that is not present in polynomial(Q), so we insert it in polynomial(Q).) Set Q2 ⇐ AVAIL, COEF(Q2) ← COEF(P), ABC(Q2) ← ABC(P), LINK(Q2) ← Q, LINK(Q1) ← Q2, Q1 ← Q2, P ← LINK(P), and return to step A2. ▮

One of the most noteworthy features of Algorithm A is the manner in which the pointer variable Q1 follows the pointer Q around the list. This is very typical of list processing algorithms, and we will see a dozen more algorithms with the same characteristic. Can the reader see why this idea was used in Algorithm A?

A reader who has little prior experience with linked lists will find it very instructive to study Algorithm A carefully; as a test case, try adding $x + y + z$ to $x^2 - 2y - z$.

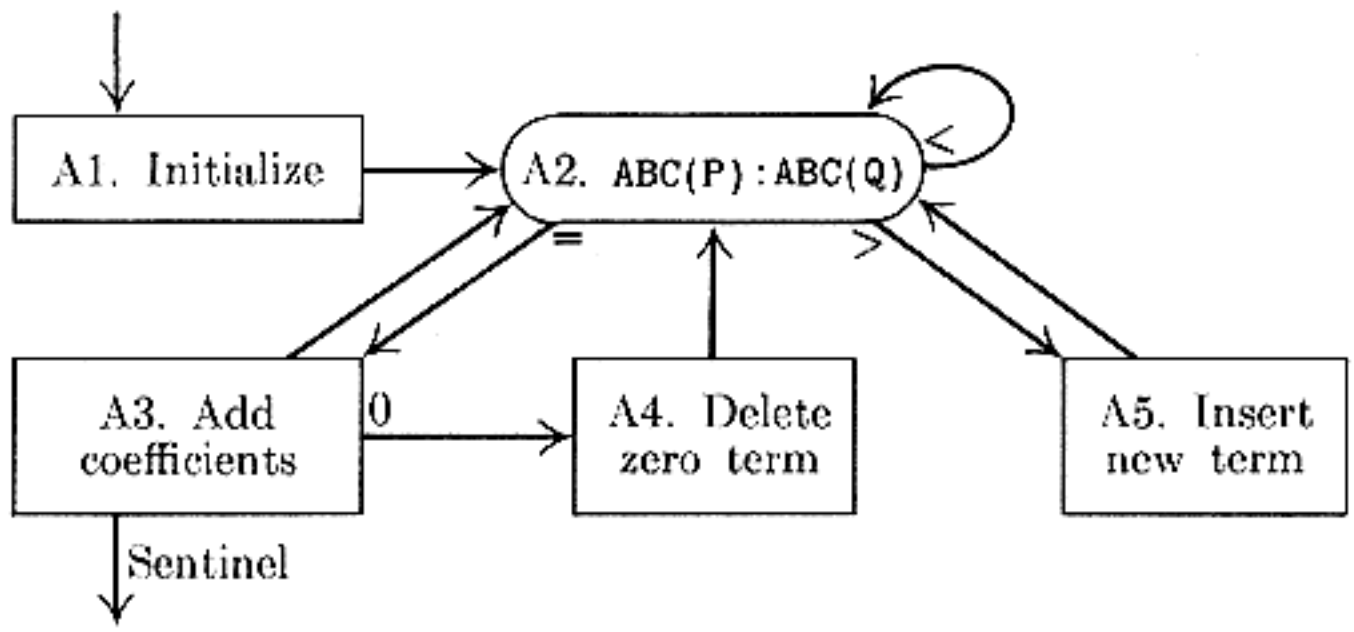


Fig. 10. Addition of polynomials.

Given Algorithm A, the multiplication operation is surprisingly easy:

Algorithm M (*Multiplication of polynomials*). This algorithm, analogous to Algorithm A, replaces polynomial(Q) by polynomial(Q) + polynomial(M) × polynomial(P).

M1. [Next multiplier.] Set M ← LINK(M). If ABC(M) < 0, the algorithm terminates.

M2. [Multiply cycle.] Perform Algorithm A, except wherever the notation "ABC(P)" appears in that algorithm, replace it by "if ABC(P) < 0 then −1, otherwise, ABC(P) + ABC(M)"; wherever "COEF(P)" appears in that algorithm, replace it by "COEF(P) × COEF(M)". Then go back to step M1. ▮

The programming of Algorithm A in MIX language shows again the ease with which linked lists are manipulated in a computer. In the following code we assume that OVERFLOW is a subroutine which either terminates the program (due to lack of memory space) or finds further available space and exits to (rJ) − 2.

Program A (*Addition of polynomials*). This is a subroutine written so that it can be used in conjunction with a multiplication subroutine (see exercise 15).

Calling sequence:	`JMP ADD`
Entry conditions:	rI1 = `P`, rI2 = `Q`.
Exit conditions:	polynomial(`Q`) has been replaced by polynomial(`Q`) + polynomial (`P`); rI1 and rI2 are unchanged; all other registers have undefined contents.

In the coding below, `P` ≡ rI1, `Q` ≡ rI2, `Q1` ≡ rI3, and `Q2` ≡ rI6, in the notation of Algorithm A.

```
01    LINK  EQU   4:5                 Definition of LINK field
02    ABC   EQU   0:3                 Definition of ABC field
03    ADD   STJ   3F          1       Entrance to subroutine
04          ENT3  0,2         1       A1. Initialize. Set Q1 ← Q.
05    0H    LD1   1,1(LINK)   1 + p   P ← LINK(P).
06    SW1   LDA   1,1         1 + p   rA(0:3) ← ABC(P).
07    1H    LD2   1,3(LINK)   x       Q ← LINK(Q1).
08    2H    CMPA  1,2(ABC)    x       A2. ABC(P):ABC(Q).
09          JE    3F          x       If equal, go to A3.
10          JG    5F          p' + q' If greater, go to A5.
11          ENT3  0,2         q'      If less, set Q1 ← Q.
12          JMP   1B          q'      Repeat.
13    3H    JAN   *           m + 1   A3. Add coefficients.
14    SW2   LDA   0,1         m       COEF(P)
15          ADD   0,2         m         + COEF(Q)
16          STA   0,2         m           → COEF(Q).
17          JANZ  6F          m       Is result zero?
18          ENT6  0,2         m'      A4. Delete zero term. Q2 ← Q.
19          LD2   1,2(LINK)   m'      Q ← LINK(Q).
20          LDX   AVAIL       m'    ⎫
21          STX   1,6(LINK)   m'    ⎬ AVAIL ⇐ Q2.
22          ST6   AVAIL       m'    ⎭
23          ST2   1,3(LINK)   m'      LINK(Q1) ← Q.
24          JMP   0B          m'      Go to advance P.
25    6H    ENT3  0,2         m''     Set Q1 ← Q.
26          JMP   0B          m''     Go to advance P, Q.
27    5H    LD6   AVAIL       p'    ⎫ A5. Insert new term.
28          J6Z   OVERFLOW    p'    ⎬ Q2 ⇐ AVAIL.
29          LDX   1,6(LINK)   p'    ⎭
30          STX   AVAIL       p'
31          STA   1,6         p'      ABC(Q2) ← ABC(P).
32    SW3   LDA   0,1         p'      rA ← COEF(P).
33          STA   0,6         p'      COEF(Q2) ← rA.
34          ST2   1,6(LINK)   p'      LINK(Q2) ← Q.
35          ST6   1,3(LINK)   p'      LINK(Q1) ← Q2.
36          ENT3  0,6         p'      Q1 ← Q2.
37          JMP   0B          p'      Go to advance P. ∎
```

Note that Algorithm A traverses each of the two lists just once; it is not necessary to loop around several times. Using Kirchhoff's law, we find that an analysis of the execution presents no difficulties; the execution time depends on the quantities

m' = number of matching terms which cancel with each other;
m'' = number of matching terms which do not cancel;
p' = number of unmatched terms in polynomial(P);
q' = number of unmatched terms in polynomial(Q).

The analysis given with Program A uses the abbreviations

$$m = m' + m'', \qquad p = m + p',$$
$$q = m + q', \qquad x = 1 + m + p' + q';$$

the running time for MIX is $(29m' + 19m'' + 29p' + 8q' + 13)u$. The total number of nodes in the storage pool needed during the execution of the algorithm is at least $2 + p + q$, and at most $2 + p + q + p'$.

EXERCISES

1. [*21*] The text suggests at the beginning of this section that an empty circular list could be represented by PTR = Λ. It might be more consistent with the philosophy of circular lists to have PTR = LOC(PTR) indicate an empty list. Does this convention facilitate operations (a), (b), or (c) described at the beginning of this section?

2. [*20*] Draw "before and after" diagrams illustrating the effect of the concatenation operation (3), assuming that PTR_1 and PTR_2 are $\neq \Lambda$.

▸ **3.** [*20*] What does operation (3) do if PTR_1 and PTR_2 are both pointing to nodes in the *same* circular list?

4. [*21*] Give insertion and deletion operations corresponding to the representation (4), which give the effect of a *stack*.

▸ **5.** [*21*] Design an algorithm which takes a circular list such as (1) and reverses the direction of all the arrows.

6. [*18*] Give diagrams of the list representation for the polynomials (a) "$xz - 3$"; (b) "0".

7. [*10*] Why is it useful to assume that the ABC fields of a polynomial list appear in decreasing order?

8. [*10*] Why is it useful to have Q1 trailing one step behind Q in Algorithm A?

▸ **9.** [*23*] Would Algorithm A work properly if P = Q (i.e., both pointer variables point at the same polynomial)? Would Algorithm M work properly if P = M, if P = Q, or if M = Q?

▸ **10.** [*20*] The algorithms in this section assume that we are using three variables x, y, and z in the polynomials, and their exponents individually never exceed b (where b is the byte size in MIX's case). Suppose that we want instead to do addition and multiplication of polynomials in only one variable, x, and to let its exponent take on values up to b^3. What changes should be made to Algorithms A and M?

11. [*24*] (The purpose of this exercise and many of those following is to create a "package" of subroutines useful for polynomial arithmetic, in conjunction with Program A.) Since Algorithms A and M change the value of polynomial (Q), it is sometimes desirable to have a subroutine that makes a copy of a given polynomial. Write a MIX subroutine with the following specifications:

Calling sequence:	JMP COPY
Entry conditions:	rI1 = P
Exit conditions:	rI2 points to a newly created polynomial equal to polynomial(P); rI1 is unchanged; other registers are undefined.

12. [*21*] Compare the running time of the program in exercise 11 with that of Algorithm A when polynomial(Q) = "0".

13. [*20*] Write a MIX subroutine with the following specifications:

Calling sequence:	JMP ERASE
Entry conditions:	rI1 = P
Exit conditions:	polynomial (P) has been added to the AVAIL list; all register contents are undefined.

[*Note:* This subroutine can be used in conjunction with the subroutine of exercise 11 in the sequence "LD1 Q; JMP ERASE; LD1 P; JMP COPY; ST2 Q" to achieve the effect "polynomial (Q) ← polynomial (P)".]

14. [*22*] Write a MIX subroutine with the following specifications:

Calling sequence:	JMP ZERO
Entry conditions:	None
Exit conditions:	rI2 points to a newly created polynomial equal to "0"; other register contents are undefined.

15. [*24*] Write a MIX subroutine to perform Algorithm M, having the following specifications:

Calling sequence:	JMP MULT
Entry conditions:	rI1 = P, rI2 = Q, rI4 = M.
Exit conditions:	polynomial (Q) ← polynomial (Q) + polynomial (M) × polynomial (P); rI1, rI2, rI4 are unchanged; other registers undefined.

(*Note:* Use Program A as a subroutine, changing the settings of SW1, SW2, and SW3.)

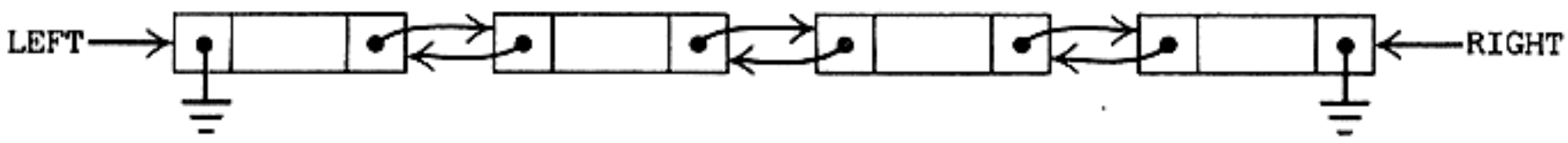
16. [*M22*] Estimate the running time of the subroutine in exercise 15 in terms of some relevant parameters.

▸ **17.** [*22*] What advantage is there in representing polynomials with a circular list as in this section, instead of with a straight linear linked list terminated by Λ as in the previous section?

18. [*25*] Devise a way to represent circular lists inside a computer in such a way that the list can be traversed efficiently in both directions, yet only one link field is used per node. [*Hint:* If we are given two pointers, to two successive nodes x_{i-1} and x_i, it should be possible to locate both x_{i+1} and x_{i-2}.]

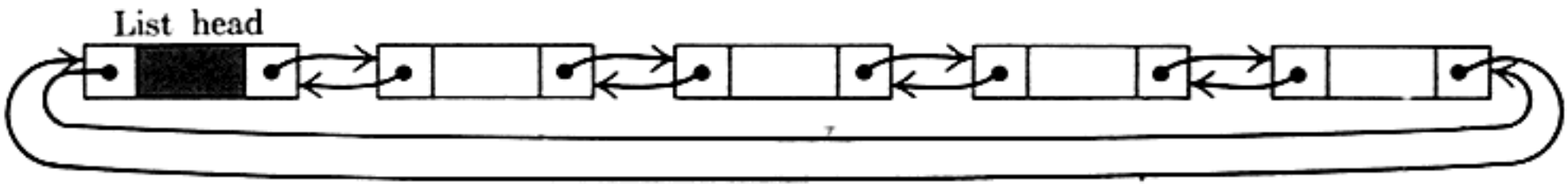
2.2.5. Doubly Linked Lists

For even greater flexibility in the manipulation of linear lists, we can include two links in each node, pointing to the items on either side of that node:



(1)

Here LEFT and RIGHT are pointer variables to the left and right of the list. Each node of the list includes two links, called, for example, LLINK and RLINK. The operations of a general deque are readily performed with the above representation; see exercise 1. However, manipulations of doubly linked lists almost always become much easier if a *list head* node is part of each list, as described in the preceding section. When a list head is present, we have the following typical diagram of a doubly linked list:



(2)

The RLINK and LLINK fields of the list head take the place of LEFT and RIGHT in (1). There is complete symmetry between left and right; the list head could equally well have been shown at the right of (2). If the list is empty, both link fields of the list head point to the head itself.

The list representation (2) clearly satisfies the condition

$$\texttt{RLINK(LLINK(X))} = \texttt{LLINK(RLINK(X))} = \texttt{X} \tag{3}$$

if X is the location of any node in the list (including the head). This fact is the principal reason representation (2) is preferable to (1).

A doubly linked list usually takes more memory space than a singly linked one does (although sometimes there is already room for another link in a node that doesn't fill a complete computer word). The additional operations that can now be performed efficiently are often more than ample compensation for this extra space requirement. Besides the obvious advantage of being able to go back and forth at will when examining a doubly linked list, one of the principal new abilities is the fact that *we can delete* NODE(X) *from the list it is in, given only the value of* X. This deletion operation is easy to derive from a "before and after" diagram (Fig. 11) and it is very simple:

$$\texttt{RLINK(LLINK(X))} \leftarrow \texttt{RLINK(X)}, \qquad \texttt{LLINK(RLINK(X))} \leftarrow \texttt{LLINK(X)},$$
$$\texttt{AVAIL} \Leftarrow \texttt{X}. \tag{4}$$

In a list which has only one-way links, we cannot delete NODE(X) without knowing which node precedes it in the chain, since the preceding node needs to

have its link altered when NODE(X) is deleted. In all the algorithms considered in Sections 2.2.3 and 2.2.4 this additional knowledge was present whenever a node was to be deleted; see, in particular, Algorithm 2.2.4A, where we had pointer Q1 following pointer Q for just this purpose. But we will meet several algorithms which require removing random nodes from the middle of a list, and doubly linked lists are frequently used just for this reason. (We should point out that in a circular list it is possible to delete NODE(X), given X, if we go around the entire circle to find the predecessor of X. But this operation is clearly inefficient when the list is long, so it is rarely an acceptable substitute for doubly linking the list.)

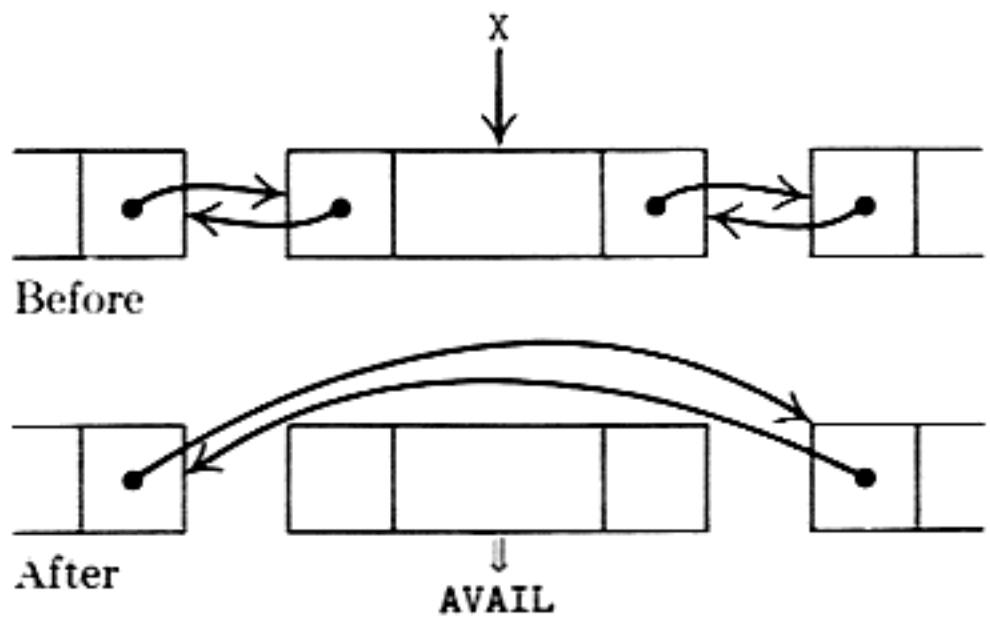


Fig. 11. Deletion from a doubly linked list.

Similarly, a doubly linked list permits the easy insertion of a node adjacent to NODE(X) at either the left or the right. The steps

$$\begin{aligned} &\texttt{P} \Leftarrow \texttt{AVAIL}, \quad \texttt{LLINK(P)} \leftarrow \texttt{X}, \quad \texttt{RLINK(P)} \leftarrow \texttt{RLINK(X)}, \\ &\qquad \texttt{LLINK(RLINK(X))} \leftarrow \texttt{P}, \quad \texttt{RLINK(X)} \leftarrow \texttt{P} \end{aligned} \tag{5}$$

do such an insertion to the right of NODE(X); and by interchanging left and right we get the corresponding algorithm for insertion to the left. Operation (5) changes the settings of five links, so it is a little slower than an insertion operation in a one-way list where only three links need to be changed.

As an example of the use of doubly linked lists, we will now consider the writing of a *discrete simulation* program. "Discrete simulation" means the simulation of a system in which all changes in the state of the system may be assumed to happen at certain discrete instants of time. The "system" being simulated usually is a set of individual activities which are largely independent although they interact with each other; examples are customers at a store, ships in a harbor, people in a corporation. In a discrete simulation, we proceed by doing whatever is to be done at a certain instant of simulated time, then advance the simulated clock to the next time when some action is to occur.

By contrast, a "continuous simulation" would be simulation of activities which are under continuous changes, such as traffic moving on a highway, spaceships traveling to other planets, etc. Continuous simulation can often be satisfactorily approximated by discrete simulation with very small time intervals

between steps; however, in such a case we usually have "synchronous" discrete simulation, in which many parts of the system are slightly altered at each discrete time interval, and that generally calls for a somewhat different type of program organization than the kind considered here.

The program developed below simulates the elevator system in the Mathematics building of the California Institute of Technology. The results of such a simulation will perhaps be of use only to people who make reasonably frequent visits to Caltech; and even for those who do, it may be simpler just to try using the elevator several times instead of writing a computer program. But, as is usual with simulation studies, the methods we will use to achieve the simulation are of much more interest than the answers given by the program. The methods to be discussed below illustrate typical implementation techniques used with discrete simulation programs.

The Mathematics building has five floors: sub-basement, basement, first, second, and third. There is a single elevator, which has automatic controls and can stop at each floor. For convenience we will renumber the floors 0, 1, 2, 3, and 4.

On each floor there are two call buttons, one for `UP` and one for `DOWN`. (Actually floor 0 has only `UP` and floor 4 has only `DOWN`, but we may ignore that anomaly since the excess buttons will never be used.) Corresponding to these buttons, there are ten variables `CALLUP`$[j]$ and `CALLDOWN`$[j]$, $0 \le j \le 4$. There are also variables `CALLCAR`$[j]$, $0 \le j \le 4$, representing buttons within the elevator car which direct it to a destination floor. When a man presses a button he sets the appropriate variable to 1; the elevator clears the variable to 0 after the request has been fulfilled.

The above describes the elevator from a man's point of view; the situation is more interesting as viewed by the elevator. The elevator is in one of three states: `GOINGUP`, `GOINGDOWN`, or `NEUTRAL`. (The current state is indicated to passengers by lighted arrows inside the elevator.) If it is in `NEUTRAL` state and not on floor 2, the machine will close its doors and (if no command is given by the time its doors are shut) it will change to `GOINGUP` or `GOINGDOWN`, heading for floor 2. (This is the "home floor," since most passengers get in there.) On floor 2 in `NEUTRAL` state, the doors will eventually close and the machine will wait silently for another command. The first command received for another floor sets the machine `GOINGUP` or `GOINGDOWN` as appropriate; it stays in this state until there are no commands waiting in the same direction, and then it switches direction or switches to `NEUTRAL` just before opening the doors, depending on what other commands are in the `CALL` variables. The elevator takes a certain amount of time to open and close its doors, to accelerate and decelerate, and to get from one floor to another. All these quantities are indicated in the algorithm below, which is much more precise than this informal description can be. The algorithm we will now study may not reflect the elevator's true principles of operation, but it is believed to be the simplest set of rules which explain all the phenomena observed during several hours of experimentation by the author during the writing of this section.

The elevator system is simulated by using two coroutines, one for the passengers and one for the elevator; these routines specify all the actions to be performed, as well as various time delays which are to be used in the simulation. In the following description, the variable TIME represents the current value of the simulated time clock. All units of time are given in *tenths of seconds*. There are also several other variables:

FLOOR, the current position of the elevator;
D1, a variable which is zero except during the time people are getting in or out of the elevator;
D2, a variable which becomes zero if the elevator has sat on one floor without moving for 30 sec or more;
D3, a variable which is zero except during the time the doors are open but nobody is getting in or out of the elevator;
STATE, the current state of the elevator (GOINGUP, GOINGDOWN, or NEUTRAL).

Initially, FLOOR = 2, D1 = D2 = D3 = 0, and STATE = NEUTRAL.

Coroutine M (*Men*). When each man enters the system, he begins to perform the actions specified below, starting at step M1.

M1. [Enter, prepare for successor.] The following quantities are determined in some manner that will not be specified here:

IN, the floor on which the new man has entered the system;
OUT, the floor to which he wants to go (OUT $\neq$ IN);
INTERTIME, the amount of time before the next man will enter the system;
GIVEUPTIME, the amount of time this man will wait for the elevator before he gives up and decides to walk.

After these quantities have been computed, the simulation program sets things up so that another man enters the system at TIME + INTERTIME.

M2. [Signal and wait.] (The purpose of this step is to call for the elevator; some special cases arise if the elevator is already on the right floor.) If FLOOR = IN and if the elevator's next action is step E6 below (i.e., if the elevator doors are now closing), send the elevator immediately to its step E3 and cancel its activity E6. (This means the doors will open again before the elevator moves.) If FLOOR = IN and if D3 $\neq$ 0, set D3 $\leftarrow$ 0, set D1 $\neq$ 0, and start up the elevator's activity E4 again. (This means the elevator doors are open on this floor, but everyone else has already gotten on or off; elevator step E4 is a sequencing step that grants people permission to enter the elevator according to normal laws of courtesy, and so restarting E4 gives this man a chance to get in before the doors close.) In all other cases, the man sets CALLUP[IN] $\leftarrow$ 1 or CALLDOWN[IN] $\leftarrow$ 1, according as OUT > IN or OUT < IN; and if D2 = 0 or the elevator is in its "dormant" position E1, the DECISION subroutine specified below is performed. (The DECISION subroutine is used to take the elevator out of NEUTRAL state at certain critical times.)

M3. [Enter queue.] Insert this man at the rear of QUEUE[IN], which is a linear list representing the people waiting on this floor. Now this man ceases activity; he will perform action M4 after GIVEUPTIME units of time, unless step E4 of the elevator routine below sends him to M5 earlier.

M4. [Give up.] If FLOOR $\neq$ IN or D1 $= 0$, delete this man from QUEUE[IN] and from the simulated system. (He has decided the elevator is too slow.) If FLOOR $=$ IN and D1 $\neq 0$, he stays and waits (since he knows he will soon be able to get in).

M5. [Get in.] Delete this man from QUEUE[IN], and insert him in ELEVATOR, which is a stack-like list representing the people now in the elevator. Set CALLCAR[OUT] $\leftarrow 1$.

Now if STATE $=$ NEUTRAL, set STATE $\leftarrow$ GOINGUP or GOINGDOWN as appropriate, and set the elevator's activity E5 to be executed after 25 units of time. (This is a special feature of the elevator, that the doors close faster when a man gets in the car and the elevator is in NEUTRAL state. The 25 units of time gives step E4 the opportunity to make sure that D1 is properly set up by the time step E5, the door-closing action, occurs.)

Now the man waits until he is sent to step M6, by step E4 below, when the elevator has reached his floor.

M6. [Get out.] Delete this man from ELEVATOR and from the simulated system. ∎

Coroutine E (*Elevator*). This coroutine represents the actions of the elevator, and also in step E4 the control of when people get in and out.

E1. [Wait for call.] (At this point the elevator is sitting at floor 2 with the doors closed waiting for something to happen.) If someone presses a button, the DECISION subroutine will take us to step E3 or E6. Meanwhile, wait.

E2. [Change of state?] If STATE $=$ GOINGUP and CALLUP[j] $=$ CALLDOWN[j] $=$ CALLCAR[j] $= 0$ for all $j >$ FLOOR, then set STATE $\leftarrow$ NEUTRAL or STATE $\leftarrow$ GOINGDOWN, according as CALLCAR[j] $= 0$ for all $j <$ FLOOR or not, and set all CALL variables for the current floor to zero. If STATE $=$ GOINGDOWN, do similar actions with directions reversed.

E3. [Open door.] Set D1 and D2 to any nonzero values. Set elevator activity E9 to start up independently after 300 units of time. (This activity may be canceled in step E6 below before it occurs.) Also set elevator activity E5 to start up independently after 76 units of time. Then wait 20 units of time (to simulate opening of the doors) and go to E4.

E4. [Let people out, in.] If anyone in the ELEVATOR list has OUT $=$ FLOOR, send the man of this type who has most recently entered immediately to step M6 of his program, wait 25 units, and repeat step E4. If no such men exist, but QUEUE[FLOOR] is not empty, send the front man of that queue immediately to step M5 instead of M4 in his program, wait 25 units, and repeat step E4. But if QUEUE[FLOOR] is empty, set D1 $\leftarrow 0$, D3 $\neq 0$, and wait for

some other activity to initiate further action. (Step E5 will send us to E6, or step M2 will restart E4.)

E5. [Close door.] If D1 $\neq 0$, wait 40 units and repeat this step (the doors flutter a little but spring open again since someone is still getting out or in). Otherwise set D3 $\leftarrow 0$ and set the elevator to start at step E6 after 20 units of time. (This simulates closing the doors after people have finished getting in or out; but if a new man enters on this floor while the doors are closing, they will open again as stated in step M2.)

E6. [Prepare to move.] Set CALLCAR[FLOOR] to zero; also set CALLUP[FLOOR] to zero if STATE $\neq$ GOINGDOWN, and also set CALLDOWN[FLOOR] to zero if STATE $\neq$ GOINGUP. (*Note:* If STATE = GOINGUP, the elevator does not clear out CALLDOWN, since it assumes people who are going down will not have entered; but see exercise 6.) Now perform the DECISION subroutine.

If STATE = NEUTRAL even after the DECISION subroutine has acted, go to E1. Otherwise, if D2 $\neq 0$, cancel the elevator activity E9. Finally, if STATE = GOINGUP, wait 15 units of time (for the elevator to build up speed) and go to E7; if STATE = GOINGDOWN, wait 15 units and go to E8.

E7. [Go up a floor.] Set FLOOR $\leftarrow$ FLOOR + 1 and wait 51 units of time. If now CALLCAR[FLOOR] = 1 or CALLUP[FLOOR] = 1, or if ((FLOOR = 2 or CALLDOWN[FLOOR] = 1) and CALLUP[j] = CALLDOWN[j] = CALLCAR[j] = 0 for all $j >$ FLOOR), wait 14 units (for deceleration) and go to E2. Otherwise, repeat this step.

E8. [Go down a floor.] This step is like E7 with directions reversed, and also the times 51 and 14 are changed to 61 and 23, respectively. (It takes the elevator longer to go down than up.)

E9. [Set inaction indicator.] Set D2 $\leftarrow 0$ and perform the DECISION subroutine. (This independent action is initiated in step E3 but it is almost always canceled in step E6. See exercise 4.) ∎

Subroutine D (DECISION *subroutine*). This subroutine is performed at certain critical times, as specified in the coroutines above, when a decision about the elevator's next direction is to be made.

D1. [Decision necessary?] If STATE $\neq$ NEUTRAL, exit from this subroutine.

D2. [Should door open?] If the elevator is positioned at E1 and if CALLUP[2], CALLCAR[2], or CALLDOWN[2] is not zero, cause the elevator to start its activity E3 after 20 units of time.

D3. [Any calls?] Find the smallest $j \neq$ FLOOR for which CALLUP[j], CALLCAR[j], or CALLDOWN[j] is non-zero, and go on to step D4. But if no such j exists, then set $j \leftarrow 2$ if the elevator is currently enacting step E6; otherwise exit from this subroutine.

D4. [Set STATE.] If FLOOR $> j$, set STATE $\leftarrow$ GOINGDOWN; if FLOOR $< j$, set STATE $\leftarrow$ GOINGUP.

Table 1. SOME ACTIONS OF THE ELEVATOR SYSTEM

TIME	STATE	FLOOR	D1	D2	D3	step	action
0000	N	2	0	0	0	M1	Man no. 1 arrives at floor 0, destination is 2.
0035	D	2	0	0	0	E8	Elevator moving down
0038	D	1	0	0	0	M1	Man no. 2 arrives at floor 4, destination is 1.
0096	D	1	0	0	0	E8	Elevator moving down
0136	D	0	0	0	0	M1	Man no. 3 arrives at floor 2, destination is 1.
0141	D	0	0	0	0	M1	Man no. 4 arrives at floor 2, destination is 1.
0152	D	0	0	0	0	M4	Man no. 1 decides to give up and he leaves.
0180	D	0	0	0	0	E2	Elevator stops.
0180	N	0	X	X	0	E3	Elevator doors start to open.
0256	N	0	0	X	X	E5	Elevator doors start to close.
0291	U	0	0	X	0	M1	Man no. 5 arrives at floor 3, destination is 1.
0291	U	0	0	X	0	E7	Elevator moving up
0342	U	1	0	X	0	E7	Elevator moving up
0364	U	2	0	X	0	M1	Man no. 6 arrives at floor 2, destination is 1.
0393	U	2	0	X	0	E7	Elevator moving up
0444	U	3	0	X	0	E7	Elevator moving up
0509	U	4	0	X	0	E2	Elevator stops.
0509	N	4	X	X	0	E3	Elevator doors start to open.
0529	N	4	X	X	0	M5	Man no. 2 gets in.
0540	D	4	X	X	0	M4	Man no. 6 decides to give up and he leaves.
0554	D	4	0	X	X	E5	Elevator doors start to close.
0589	D	4	0	X	0	E8	Elevator moving down
0602	D	3	0	X	0	M1	Man no. 7 arrives at floor 1, destination is 2.
0673	D	3	0	X	0	E2	Elevator stops.
0673	D	3	X	X	0	E3	Elevator doors start to open.
0693	D	3	X	X	0	M1	Man no. 5 gets in.
0749	D	3	0	X	X	E5	Elevator doors start to close.
0784	D	3	0	X	0	E8	Elevator moving down
0827	D	2	0	X	0	M1	Man no. 8 arrives at floor 1, destination is 0.
0868	D	2	0	X	0	E2	Elevator stops.
0868	D	2	X	X	0	E3	Elevator doors start to open.
0876	D	2	X	X	0	M1	Man no. 9 arrives at floor 1, destination is 3.
0888	D	2	X	X	0	M5	Man no. 3 gets in.
0913	D	2	X	X	0	M5	Man no. 4 gets in.
0944	D	2	0	X	X	E5	Elevator doors start to close.
0979	D	2	0	X	0	E8	Elevator moving down
1048	D	1	0	X	0	M1	Man no. 10 arrives at floor 0, destination is 4.
1063	D	1	0	X	0	E2	Elevator stops.
1063	D	1	X	X	0	E3	Elevator doors start to open.
1083	D	1	X	X	0	M6	Man no. 4 gets out, leaves system.
1108	D	1	X	X	0	M6	Man no. 3 gets out, leaves system.
1133	D	1	X	X	0	M6	Man no. 5 gets out, leaves system.
1139	D	1	X	X	0	E5	Doors flutter.
1158	D	1	X	X	0	M6	Man no. 2 gets out, leaves system.
1179	D	1	X	X	0	E5	Doors flutter.
1183	D	1	X	X	0	M5	Man no. 7 gets in.
1208	D	1	X	X	0	M5	Man no. 8 gets in.
1219	D	1	X	X	0	E5	Doors flutter.
1233	D	1	X	X	0	M5	Man no. 9 gets in.
1259	D	1	0	X	X	E5	Elevator doors start to close.
1294	D	1	0	X	0	E8	Elevator moving down
1378	D	0	0	X	0	E2	Elevator stops.
1378	U	0	X	X	0	E3	Elevator doors start to open.
1398	U	0	X	X	0	M6	Man no. 8 gets out, leaves system.
1423	U	0	X	X	0	M5	Man no. 10 gets in.
1454	U	0	0	X	X	E5	Elevator doors start to close.
1489	U	0	0	X	0	E7	Elevator moving up
1554	U	1	0	X	0	E2	Elevator stops.
1554	U	1	X	X	0	E3	Elevator doors start to open.
1630	U	1	0	X	X	E5	Elevator doors start to close.
1665	U	1	0	X	0	E7	Elevator moving up
...							
4257	N	2	0	X	0	E1	Elevator dormant
4384	N	2	0	X	0	M1	Man no. 17 arrives at floor 2, destination is 3.
4404	N	2	X	X	0	E3	Elevator doors start to open.
4424	N	2	X	X	0	M5	Man no. 17 gets in.
4449	U	2	0	X	X	E5	Elevator doors start to close.
4484	U	2	0	X	0	E7	Elevator moving up
4549	U	3	0	X	0	E2	Elevator stops.
4549	N	3	X	X	0	E3	Elevator doors start to open.
4569	N	3	X	X	0	M6	Man no. 17 gets out, leaves system.
4625	N	3	0	X	X	E5	Elevator doors start to close.
4660	D	3	0	X	0	E8	Elevator moving down
4744	D	2	0	X	0	E2	Elevator stops.
4744	N	2	X	X	0	E3	Elevator doors start to open.
4820	N	2	0	X	0	E5	Elevator doors start to close.
4840	N	2	0	X	0	E1	Elevator dormant
...							

Table 1. SOME ACTIONS OF THE ELEVATOR SYSTEM

TIME	STATE	FLOOR	D1	D2	D3	step	action
0000	N	2	0	0	0	M1	Man no. 1 arrives at floor 0, destination is 2.
0035	D	2	0	0	0	E8	Elevator moving down
0038	D	1	0	0	0	M1	Man no. 2 arrives at floor 4, destination is 1.
0096	D	1	0	0	0	E8	Elevator moving down
0136	D	0	0	0	0	M1	Man no. 3 arrives at floor 2, destination is 1.
0141	D	0	0	0	0	M1	Man no. 4 arrives at floor 2, destination is 1.
0152	D	0	0	0	0	M4	Man no. 1 decides to give up and he leaves.
0180	D	0	0	0	0	E2	Elevator stops.
0180	N	0	X	X	0	E3	Elevator doors start to open.
0256	N	0	0	X	X	E5	Elevator doors start to close.
0291	U	0	0	X	0	M1	Man no. 5 arrives at floor 3, destination is 1.
0291	U	0	0	X	0	E7	Elevator moving up
0342	U	1	0	X	0	E7	Elevator moving up
0364	U	2	0	X	0	M1	Man no. 6 arrives at floor 2, destination is 1.
0393	U	2	0	X	0	E7	Elevator moving up
0444	U	3	0	X	0	E7	Elevator moving up
0509	U	4	0	X	0	E2	Elevator stops.
0509	N	4	X	X	0	E3	Elevator doors start to open.
0529	N	4	X	X	0	M5	Man no. 2 gets in.
0540	D	4	X	X	0	M4	Man no. 6 decides to give up and he leaves.
0554	D	4	0	X	X	E5	Elevator doors start to close.
0589	D	4	0	X	0	E8	Elevator moving down
0602	D	3	0	X	0	M1	Man no. 7 arrives at floor 1, destination is 2.
0673	D	3	0	X	0	E2	Elevator stops.
0673	D	3	X	X	0	E3	Elevator doors start to open.
0693	D	3	X	X	0	M1	Man no. 5 gets in.
0749	D	3	0	X	X	E5	Elevator doors start to close.
0784	D	3	0	X	0	E8	Elevator moving down
0827	D	2	0	X	0	M1	Man no. 8 arrives at floor 1, destination is 0.
0868	D	2	0	X	0	E2	Elevator stops.
0868	D	2	X	X	0	E3	Elevator doors start to open.
0876	D	2	X	X	0	M1	Man no. 9 arrives at floor 1, destination is 3.
0888	D	2	X	X	0	M5	Man no. 3 gets in.
0913	D	2	X	X	0	M5	Man no. 4 gets in.
0944	D	2	0	X	X	E5	Elevator doors start to close.
0979	D	2	0	X	0	E8	Elevator moving down
1048	D	1	0	X	0	M1	Man no. 10 arrives at floor 0, destination is 4.
1063	D	1	0	X	0	E2	Elevator stops.
1063	D	1	X	X	0	E3	Elevator doors start to open.
1083	D	1	X	X	0	M6	Man no. 4 gets out, leaves system.
1108	D	1	X	X	0	M6	Man no. 3 gets out, leaves system.
1133	D	1	X	X	0	M6	Man no. 5 gets out, leaves system.
1139	D	1	X	X	0	E5	Doors flutter.
1158	D	1	X	X	0	M6	Man no. 2 gets out, leaves system.
1179	D	1	X	X	0	E5	Doors flutter.
1183	D	1	X	X	0	M5	Man no. 7 gets in.
1208	D	1	X	X	0	M5	Man no. 8 gets in.
1219	D	1	X	X	0	E5	Doors flutter.
1233	D	1	X	X	0	M5	Man no. 9 gets in.
1259	D	1	0	X	X	E5	Elevator doors start to close.
1294	D	1	0	X	0	E8	Elevator moving down
1378	D	0	0	X	0	E2	Elevator stops.
1378	U	0	X	X	0	E3	Elevator doors start to open.
1398	U	0	X	X	0	M6	Man no. 8 gets out, leaves system.
1423	U	0	X	X	0	M5	Man no. 10 gets in.
1454	U	0	0	X	X	E5	Elevator doors start to close.
1489	U	0	0	X	0	E7	Elevator moving up
1554	U	1	0	X	0	E2	Elevator stops.
1554	U	1	X	X	0	E3	Elevator doors start to open.
1630	U	1	0	X	X	E5	Elevator doors start to close.
1665	U	1	0	X	0	E7	Elevator moving up
...							
4257	N	2	0	X	0	E1	Elevator dormant
4384	N	2	0	X	0	M1	Man no. 17 arrives at floor 2, destination is 3.
4404	N	2	X	X	0	E3	Elevator doors start to open.
4424	N	2	X	X	0	M5	Man no. 17 gets in.
4449	U	2	0	X	X	E5	Elevator doors start to close.
4484	U	2	0	X	0	E7	Elevator moving up
4549	U	3	0	X	0	E2	Elevator stops.
4549	N	3	X	X	0	E3	Elevator doors start to open.
4569	N	3	X	X	0	M6	Man no. 17 gets out, leaves system.
4625	N	3	0	X	X	E5	Elevator doors start to close.
4660	D	3	0	X	0	E8	Elevator moving down
4744	D	2	0	X	0	E2	Elevator stops.
4744	N	2	X	X	0	E3	Elevator doors start to open.
4820	N	2	0	X	0	E5	Elevator doors start to close.
4840	N	2	0	X	0	E1	Elevator dormant
...							

D5. [Elevator dormant?] If the elevator coroutine is positioned at step E1, and if $j \neq 2$, set the elevator to perform step E6 after 20 units of time. Exit from the subroutine. ▮

The elevator system described above is quite complicated by comparison with other algorithms we have seen in this book, but the choice of a real-life system is more typical of a simulation problem than any cooked-up "textbook example" would ever be.

To help understand the system, consider Table 1 which gives part of the history of one simulation. It is perhaps best to start by examining the simple case starting at time 4257: the elevator is idly sitting at floor 2 with its doors shut, when a man arrives (time 4384). Two seconds later, the doors open, and after two more seconds he gets in; by pushing button "3" he starts the elevator moving up; ultimately he gets off at floor 3 and the elevator returns to floor 2. The first entries in Table 1 show a case with much more activity: A man calls the elevator to floor 0, but he is rather impatient and gives up after 15.2 sec. The elevator stops at floor 0 but finds nobody there; then it heads to floor 4 since there are several calls wanting to go downward; etc.

The programming of this system for a computer (in our case, MIX) merits careful study. At any given time during the simulation, we may have many simulated men in the system (in various queues and ready to "give up" at various times), and there is also the possibility of essentially simultaneous execution of steps E4, E5, and E9 if many people are trying to get out as the elevator is trying to close its doors. The passing of simulated time and the handling of "simultaneity" may be programmed by having each entity represented by a node that includes a NEXTTIME field (denoting the time when the next action for this entity is to take place) and a NEXTINST field (denoting the memory address where this entity is to start executing instructions, analogous to ordinary coroutine linkage). Each entity waiting for time to pass is placed in a doubly linked list called the WAIT list; this "agenda" is sorted on the NEXTTIME fields of its nodes, so that the actions may be processed in the correct sequence of simulated times. The program also uses doubly linked lists for the ELEVATOR and for the QUEUE lists.

The node representing each activity (whether a man or an elevator action) has the form

+	IN	LLINK1	RLINK1	
+	NEXTTIME			
+	NEXTINST	0	0	39
+	OUT	LLINK2	RLINK2	

(6)

Here LLINK1 and RLINK1 are the links for the WAIT list; LLINK2 and RLINK2 are used as links in the QUEUE lists or the ELEVATOR. The latter two fields and the

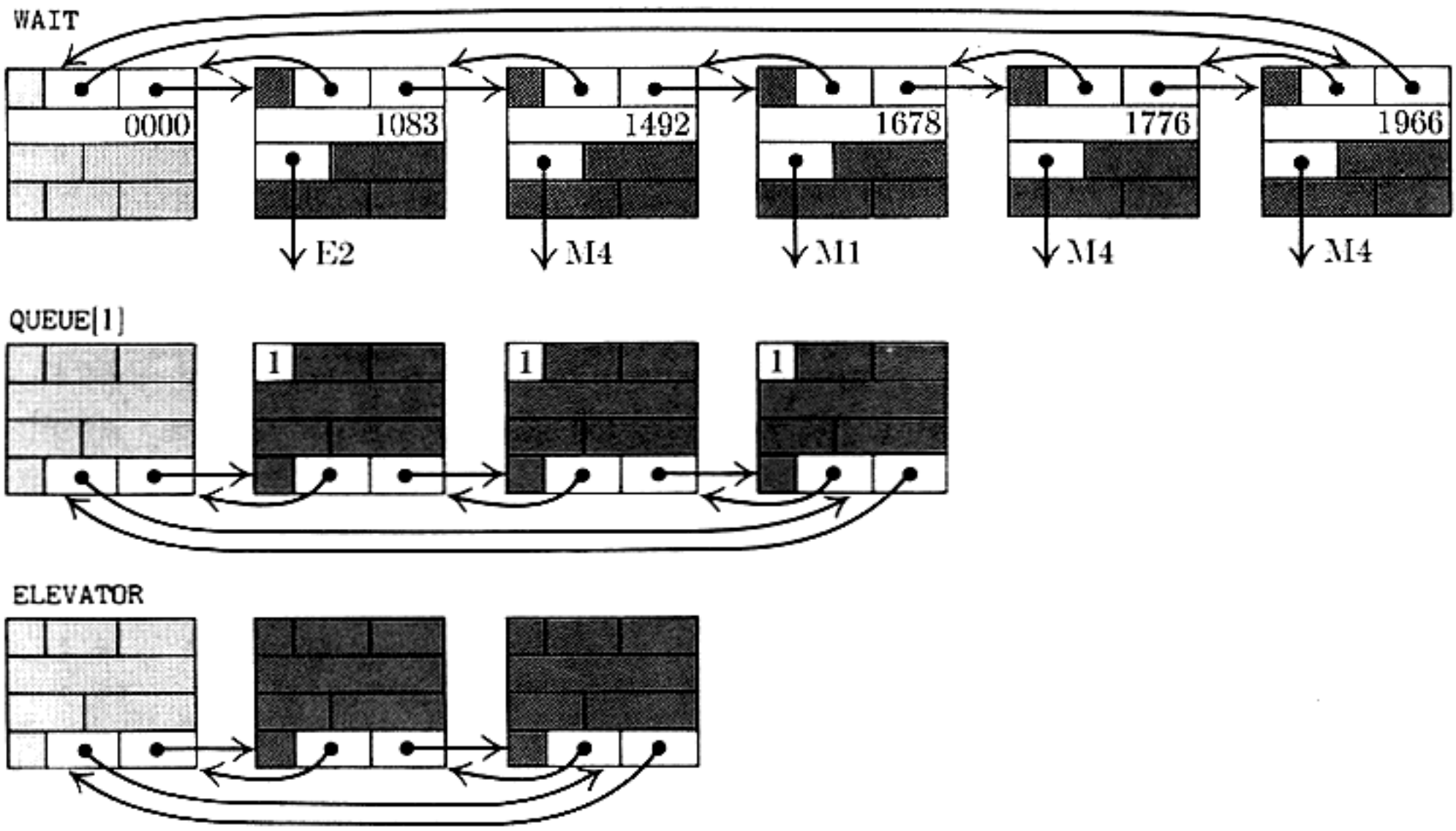


Fig. 12. Some lists used in the elevator simulation program. (List heads appear at the left.)

IN and OUT field are relevant when node (6) represents a man, but they are not relevant for nodes that represent elevator actions. The third word of the node is actually a MIX "JMP" instruction.

Figure 12 shows typical contents of the WAIT list, ELEVATOR list, and one of the QUEUE lists; each node in the QUEUE list is simultaneously in the WAIT list with NEXTINST = M4, but this has not been indicated in the figure, since the complexity of the linking would obscure the basic idea.

Now let us consider the program itself. The program is quite long, although (as with all long programs) it divides into small parts each of which is quite simple in itself. First comes a number of lines of code that just serve to define the initial contents of the tables. There are several points of interest here: We have list heads for the WAIT list (lines 10–11), the QUEUE lists (lines 26–31), and the ELEVATOR list (lines 32–33). Each of these is a node of the form (6), but with unimportant words deleted; the WAIT list head contains only the first two words of a node, and the QUEUE and ELEVATOR list heads require only the last word of a node. We also have four nodes which are always present in the system (lines 12–23): MAN1, a node which is always positioned at step M1 ready to enter a new man into the system; ELEV1, a node which governs the main actions of the elevator at steps E1, E2, E3, E4, E6, E7, and E8; and ELEV2 and ELEV3, nodes which are used for the elevator actions E5 and E9, which take place independently of other elevator actions with respect to simulated time. Each of these four nodes contains only three words, since they never appear in the QUEUE or

ELEVATOR lists. The nodes representing each actual man in the system will appear in a storage pool following the main program.

```
01      * THE ELEVATOR SIMULATION
02      IN          EQU   1:1                    Definition of fields
03      LLINK1      EQU   2:3                        within nodes
04      RLINK1      EQU   4:5
05      NEXTINST    EQU   0:2
06      OUT         EQU   1:1
07      LLINK2      EQU   2:3
08      RLINK2      EQU   4:5
09      * FIXED-SIZE TABLES AND LIST HEADS
10      WAIT        CON   *+2(LLINK1),*+2(RLINK1)  List head for WAIT list
11                  CON   0                            NEXTTIME = 0 always
12      MAN1        CON   *-2(LLINK1),*-2(RLINK1)  This node represents action
13                  CON   0                            M1 and it is initially the
14                  JMP   M1                           sole entry in the WAIT list.
15      ELEV1       CON   0                        This node represents the
16                  CON   0                            elevator actions, except
17                  JMP   E1                           for E5 and E9.
18      ELEV2       CON   0                        This node represents the
19                  CON   0                            independent elevator
20                  JMP   E5                           action at E5.
21      ELEV3       CON   0                        This node represents the
22                  CON   0                            independent elevator
23                  JMP   E9                           action at E9.
24      AVAIL       CON   0                        Link to available nodes
25      TIME        CON   0                        Current simulated time
26      QUEUE       EQU   *-3
27                  CON   *-3(LLINK2),*-3(RLINK2)  List head for QUEUE[0]
28                  CON   *-3(LLINK2),*-3(RLINK2)  List head for QUEUE[1]
29                  CON   *-3(LLINK2),*-3(RLINK2)  All queues initially
30                  CON   *-3(LLINK2),*-3(RLINK2)       are empty
31                  CON   *-3(LLINK2),*-3(RLINK2)  List head for QUEUE[4]
32      ELEVATOR    EQU   *-3
33                  CON   *-3(LLINK2),*-3(RLINK2)  List head for ELEVATOR
34                  CON   0                        }
35                  CON   0                        } "Padding" for CALL table
36                  CON   0                        }      (see lines 183–186)
37                  CON   0                        }
38      CALL        CON   0                        CALLUP[0], CALLCAR[0], CALLDOWN[0]
39                  CON   0                        CALLUP[1], CALLCAR[1], CALLDOWN[1]
40                  CON   0                        CALLUP[2], CALLCAR[2], CALLDOWN[2]
41                  CON   0                        CALLUP[3], CALLCAR[3], CALLDOWN[3]
42                  CON   0                        CALLUP[4], CALLCAR[4], CALLDOWN[4]
43                  CON   0                        }
44                  CON   0                        } "Padding" for CALL table
45                  CON   0                        }      (see lines 178–181)
46                  CON   0                        }
47      D1          CON   0                        Indicates door open, activity
48      D2          CON   0                        Indicates prolonged standstill
49      D3          CON   0                        Indicates door open, inactivity ▮
```

The next part of the program coding contains basic subroutines and the main control routines for the simulation process. Subroutines INSERT and DELETE perform typical manipulations on doubly linked lists; they put the current node into or take it out of a QUEUE or ELEVATOR list. (In the program, the "current node" C is always represented by index register 6.) There are also subroutines for the WAIT list: Subroutine SORTIN adds the current node to the WAIT list, sorting it into the right place based on its NEXTTIME field. Subroutine IMMED inserts the current node at the front of the WAIT list. Subroutine HOLD puts the current node into the WAIT list, with NEXTTIME equal to the current time plus the amount in register A. Subroutine DELETEW deletes the current node from the WAIT list.

The routine CYCLE is the heart of the simulation control: it decides which activity is to act next (namely, the first element of the WAIT list, which we know is nonempty), and jumps to it. There are two special entrances to CYCLE: CYCLE1 first sets NEXTINST in the current node, and HOLDC is the same with an additional call on the HOLD subroutine. Thus, the effect of the instruction "JMP HOLDC" with amount t in register A is to suspend activity for t units of simulated time and then to return to the following location.

```
50      * SUBROUTINES AND CONTROL ROUTINE
51  INSERT  STJ   9F                    Insert NODE(C) to left of NODE(rI1):
52          LD2   3,1(LLINK2)           rI2 ← LLINK2(rI1).
53          ST2   3,6(LLINK2)           LLINK2(C) ← rI2.
54          ST6   3,1(LLINK2)           LLINK2(rI1) ← C.
55          ST6   3,2(RLINK2)           RLINK2(rI2) ← C.
56          ST1   3,6(RLINK2)           RLINK2(C) ← rI1.
57  9H      JMP   *                     Exit from subroutine.
58  DELETE  STJ   9F                    Delete NODE(C) from its list:
59          LD1   3,6(LLINK2)           P ← LLINK2(C).
60          LD2   3,6(RLINK2)           Q ← RLINK2(C).
61          ST1   3,2(LLINK2)           LLINK2(Q) ← P.
62          ST2   3,1(RLINK2)           RLINK2(P) ← Q.
63  9H      JMP   *                     Exit from subroutine.
64  IMMED   STJ   9F                    Insert NODE(C) first in WAIT list:
65          LDA   TIME
66          STA   1,6                   Set NEXTTIME(C) ← TIME.
67          ENT1  WAIT                  P ← LOC(WAIT).
68          JMP   2F                    Insert NODE(C) to right of NODE(P).
69  HOLD    ADD   TIME                  rA ← TIME + rA.
70  SORTIN  STJ   9F                    Sort NODE(C) into WAIT list:
71          STA   1,6                   Set NEXTTIME(C) ← rA.
72          ENT1  WAIT                  P ← LOC(WAIT).
73          LD1   0,1(LLINK1)           P ← LLINK1(P).
74          CMPA  1,1                   Compare NEXTTIME fields, right to left.
75          JL    *-2                   Repeat until NEXTTIME(C) ≥ NEXTTIME(P).
76  2H      LD2   0,1(RLINK1)           Q ← RLINK1(P).
77          ST2   0,6(RLINK1)           RLINK1(C) ← Q.
78          ST1   0,6(LLINK1)           LLINK1(C) ← P.
79          ST6   0,1(RLINK1)           RLINK1(P) ← C.
80          ST6   0,2(LLINK1)           LLINK1(Q) ← C.
81  9H      JMP   *                     Exit from subroutine.
```

```
82      DELETEW  STJ   9F          Delete NODE(C) from WAIT list:
83               LD1   0,6(LLINK1) (This is same as lines 58–63
84               LD2   0,6(RLINK1)      except LLINK1, RLINK1 are used
85               ST1   0,2(LLINK1)      instead of LLINK2, RLINK2)
86               ST2   0,1(RLINK1)
87      9H       JMP   *
88      CYCLE1   STJ   2,6(NEXTINST) Set NEXTINST(C) ← rJ.
89               JMP   CYCLE
90      HOLDC    STJ   2,6(NEXTINST) Set NEXTINST(C) ← rJ.
91               JMP   HOLD          Insert NODE(C) in WAIT, delay (rA).
92      CYCLE    LD6   WAIT(RLINK1)  Set current node C ← RLINK1(LOC(WAIT)).
93               LDA   1,6           NEXTTIME(C)
94               STA   TIME               becomes new value of simulated TIME.
95               JMP   DELETEW       Remove NODE(C) from WAIT list.
96               JMP   2,6           Jump to NEXTINST(C). ∎
```

Now comes the program for coroutine M. At the beginning of step M1, the current node C is MAN1 (see lines 12–14 above), and lines 099–100 of the program cause MAN1 to be reinserted into the WAIT list so that the next man will be generated after INTERTIME units of simulated time. The following lines 101–114 take care of setting up a node for the newly generated man; his IN and OUT floors are recorded in this node position. The AVAIL stack is singly linked in the RLINK1 field of each node. Note that lines 101–108 perform the action "C ⇐ AVAIL" using the POOLMAX technique, 2.2.3–(7); no test for OVERFLOW is necessary here, since the total size of the storage pool (the number of men in the system at any one time) rarely exceeds 10 nodes (40 words). The return of a node to the AVAIL stack appears in lines 156–158.

Throughout the program, index register 4 equals the variable FLOOR, and index register 5 is positive, negative, or zero, depending on whether STATE = GOINGUP, GOINGDOWN, or NEUTRAL, respectively. The variables CALLUP[j], CALLCAR[j], and CALLDOWN[j] occupy the respective fields (1:1), (3:3), and (5:5) of location CALL + j.

```
097     * COROUTINE M.                 M1. Enter, prepare for successor.
098     M1       JMP   VALUES        Compute IN, OUT, INTERTIME, GIVEUPTIME.
099              LDA   INTERTIME     INTERTIME is computed by VALUES subroutine.
100              JMP   HOLD          Put NODE(C) in WAIT, delay INTERTIME.
101              LD6   AVAIL         C ← AVAIL.
102              J6P   1F            If AVAIL ≠ Λ, jump.
103              LD6   POOLMAX
104              INC6  4             C ← POOLMAX + 4.
105              ST6   POOLMAX       POOLMAX ← C.
106              JMP   *+3
107     1H       LDA   0,6(RLINK1)
108              STA   AVAIL         AVAIL ← RLINK1(AVAIL).
109              LD1   INFLOOR       rI1 ← INFLOOR (computed by VALUES above).
110              ST1   0,6(IN)       IN(C) ← rI1.
111              LD2   OUTFLOOR      rI2 ← OUTFLOOR (computed by VALUES).
112              ST2   3,6(OUT)      OUT(C) ← rI2.
113              ENTA  39            Put constant 39 (JMP operation code)
114              STA   2,6                into third word of node format (6).
```

```
115  M2    ENTA  0,4              M2. Signal and wait. Set rA ← FLOOR.
116        DECA  0,1              FLOOR — IN
117        ST6   TEMP             Save value of C.
118        JANZ  2F               Jump if FLOOR ≠ IN.
119        ENT6  ELEV1            Set C ← LOC(ELEV1).
120        LDA   2,6(NEXTINST)    Is elevator positioned at E6?
121        DECA  E6
122        JANZ  3F
123        ENTA  E3               If so, reposition it at E3.
124        STA   2,6(NEXTINST)
125.       JMP   DELETEW          Remove it from WAIT list
126        JMP   4F                   and reinsert it at front of WAIT.
127  3H    LDA   D3
128        JAZ   2F               Jump if D3 = 0.
129        ST6   D1               Otherwise set D1 ≠ 0.
130        STZ   D3               Set D3 ← 0.
131  4H    JMP   IMMED            Insert ELEV1 at front of WAIT list.
132        JMP   M3               (rI1, rI2 have changed.)
133  2H    DEC2  0,1              rI2 ← OUT-IN.
134        ENTA  1
135        J2P   *+3              Jump if going up.
136        STA   CALL,1(5:5)      Set CALLDOWN[IN] ← 1.
137        JMP   *+2
138        STA   CALL,1(1:1)      Set CALLUP[IN] ← 1.
139        LDA   D2
140        JAZ   DECISION         If D2 = 0, call the DECISION subroutine.
141        LDA   ELEV1+2(NEXTINST)
142        DECA  E1               If the elevator is at E1, call
143        JAZ   DECISION             the DECISION subroutine.

144  M3    LD6   TEMP             M3. Enter queue.
145        LD1   0,6(IN)
146        ENT1  QUEUE,1          rI1 ← LOC(QUEUE[IN]).
147        JMP   INSERT           Insert NODE(C) at right end of QUEUE[IN].
148  M4A   LDA   GIVEUPTIME
149        JMP   HOLDC            Wait GIVEUPTIME units.
150  M4    LDA   0,6(IN)          M4. Give up.
151        DECA  0,4              IN(C) — FLOOR
152        JANZ  *+3
153        LDA   D1               FLOOR = IN(C).
154        JANZ  M4A              See exercise 7.
155  M6    JMP   DELETE           M6. Get out. NODE(C) is deleted
156        LDA   AVAIL                from QUEUE or ELEVATOR.
157        STA   0,6(RLINK1)      AVAIL ⇐ C.
158        ST6   AVAIL
159        JMP   CYCLE            Continue simulation.
160  M5    JMP   DELETE           M5. Get in. NODE(C) is deleted
161        ENT1  ELEVATOR             from QUEUE.
162        JMP   INSERT           Insert it at right of ELEVATOR.
163        ENTA  1
164        LD2   3,6(OUT)
165        STA   CALL,2(3:3)      Set CALLCAR[OUT(C)] ← 1.
166        J5NZ  CYCLE            Jump if STATE ≠ NEUTRAL.
167        DEC2  0,4
168        ENT5  0,2              Set STATE to proper direction.
169        ENT6  ELEV2            Set C ← LOC(ELEV2).
170        JMP   DELETEW          Remove E5 action from WAIT list.
171        ENTA  25
172        JMP   E5A              Restart E5 action 25 units from now. ∎
```

The program for coroutine E is a rather straightforward rendition of the semiformal description given earlier. Perhaps the most interesting portion is the preparation for the elevator's independent actions in step E3, and the searching of the ELEVATOR and QUEUE lists in step E4.

```
173     * COROUTINE E.
174  E1A        JMP    CYCLE1          Set NEXTINST ← E1, go to CYCLE.
175  E1         EQU    *               E1. Wait for call. (no action)
176  E2A        JMP    HOLDC
177  E2         J5N    1F              E2. Change of state?
178             LDA    CALL+1,4        State is GOINGUP.
179             ADD    CALL+2,4
180             ADD    CALL+3,4
181             ADD    CALL+4,4
182             JAP    E3              Are there calls for higher floors?
183             LDA    CALL-1,4(3:3)   If not, have passengers in the
184             ADD    CALL-2,4(3:3)         elevator called for lower floors?
185             ADD    CALL-3,4(3:3)
186             ADD    CALL-4,4(3:3)
187             JMP    2F
188  1H         LDA    CALL-1,4        State is GOINGDOWN.
190             ADD    CALL-2,4        Actions are like lines 178-186.
. . .
196             ADD    CALL+4,4(3:3)
197  2H         ENN5   0,5             Reverse direction of STATE.
198             STZ    CALL,4          Set CALL variables to zero.
199             JANZ   E3              Jump if calls for opposite direction,
200             ENT5   0               otherwise, set STATE ← NEUTRAL.
201  E3         ENT6   ELEV3           E3. Open door.
202             LDA    0,6             If activity E9 is already scheduled,
203             JANZ   DELETEW         remove it from the WAIT list.
204             ENTA   300
205             JMP    HOLD            Schedule activity E9 after 300 units.
206             ENT6   ELEV2
207             ENTA   76
208             JMP    HOLD            Schedule activity E5 after 76 units.
209             ST6    D2              Set D2 ≠ 0.
210             ST6    D1              Set D1 ≠ 0.
211             ENTA   20
212  E4A        ENT6   ELEV1
213             JMP    HOLDC
214  E4         ENTA   0,4             E4. Let people out, in.
215             SLA    4               Set OUT field of rA to FLOOR.
216             ENT6   ELEVATOR        C ← LOC(ELEVATOR).
217  1H         LD6    3,6(LLINK2)     C ← LLINK2(C).
218             CMP6   =ELEVATOR=      Search ELEVATOR list, right to left.
219             JE     1F              If C = LOC(ELEVATOR), search is complete
220             CMPA   3,6(OUT)        Compare OUT(C) with FLOOR.
221             JNE    1B              If not equal, continue search,
222             ENTA   M6              otherwise, prepare to send man to M6
223             JMP    2F
224  1H         LD6    QUEUE+3,4(RLINK2)  Set C ← RLINK2(LOC(QUEUE[FLOOR])).
225             CMP6   3,6(RLINK2)     Is C = RLINK2(C)?
226             JE     1F              If so, the queue is empty.
227             JMP    DELETEW         If not, cancel action M4 for this man.
228             ENTA   M5              Prepare to send man to M5.
```

```
229      2H      STA     2,6(NEXTINST)   Set NEXTINST(C).
230              JMP     IMMED           Put him at front of WAIT list.
231              ENTA    25
232              JMP     E4A             Wait 25 units and repeat E4.
233      1H      STZ     D1              Set D1 ← 0.
234              ST6     D3              Set D3 ≠ 0.
235              JMP     CYCLE           Return to simulate other events.
236      E5A     JMP     HOLDC
237      E5      LDA     D1              E5. Close door.
238              JAZ     *+3             Is D1 = 0?
239              ENTA    40              If not, people are still getting in or out.
240              JMP     E5A             Wait 40 units, repeat E5.
241              STZ     D3              If D1 = 0, set D3 ← 0.
242              ENTA    20
243              ENT6    ELEV1
244      E6A     JMP     HOLDC           Wait 20 units, then go to E6.
245      E6      J5N     *+2             E6. Prepare to move.
246              STZ     CALL,4(1:3)     If STATE ≠ GOINGDOWN, CALLUP and CALLCAR
247              J5P     *+2                 on this floor are reset.
248              STZ     CALL,4(3:5)     If ≠ GOINGUP, reset CALLCAR and CALLDOWN.
249              J5Z     DECISION        Perform DECISION subroutine.
250      E6B     J5Z     E1A             If STATE = NEUTRAL, go to E1 and wait.
251              LDA     D2
252              JAZ     *+4
253              ENT6    ELEV3           Otherwise, if D2 ≠ 0,
254              JMP     DELETEW             cancel activity E9
255              STZ     ELEV3               (see line 202).
256              ENTA    15
257              ENT6    ELEV1           Wait 15 units of time.
258              J5N     E8A             If STATE = GOINGDOWN, go to E8.
259      E7A     JMP     HOLDC
260      E7      INC4    1               E7. Go up a floor.
261              ENTA    51
262              JMP     HOLDC           Wait 51 units.
263              LDA     CALL,4(1:3)     Is CALLCAR[FLOOR] or CALLUP[FLOOR] ≠ 0?
264              JAP     1F
265              ENT1    -2,4            If not,
266              J1Z     2F                  is FLOOR = 2?
267              LDA     CALL,4(5:5)     If not, is CALLDOWN[FLOOR] ≠ 0?
268              JAZ     E7              If not, repeat step E7.
269      2H      LDA     CALL+1,4
270              ADD     CALL+2,4
271              ADD     CALL+3,4
272              ADD     CALL+4,4
273              JANZ    E7              Are there calls for higher floors?
274      1H      ENTA    14              It is time to stop the elevator.
275              JMP     E2A             Wait 14 units and go to E2.
276      E8A     JMP     HOLDC
. . .                                    (See exercise 8.)
293      E9      STZ     0,6             E9. Set inaction indicator.
294              STZ     D2              D2 ← 0.
295              JMP     DECISION        Perform DECISION subroutine.
296              JMP     CYCLE           Return to simulation of other events. ▮
```

We will not consider here the DECISION subroutine (see exercise 9), nor the VALUES subroutine which is used to specify the demands on the elevator. At the very end of the program comes the code

```
BEGIN    ENT4  2        Start with FLOOR = 2
         ENT5  0            and STATE = NEUTRAL.
         JMP   CYCLE      Begin simulation.
POOLMAX  CON   *+10       Storage pool follows literals, temp storage
         END   BEGIN
```

The above program does a fine job of simulating the elevator system, as it goes through its paces. But it would be useless to run this program, since there is no output! Actually, the author added a PRINT subroutine which was called at most of the critical steps in the program above, and this was used to prepare Table 1; the details have been omitted, since they are very straightforward but only clutter up the code.

Several programming languages have been devised which make it quite easy to specify the actions in a discrete simulation, and to use a compiler to translate these specifications into machine language. Assembly language was used in this section, of course, since we are concerned here with the techniques of manipulating linked lists, and the details of how discrete simulations are actually performed by a computer (although it has a one-track mind). We will consider the question of higher-level notations for describing these systems in Chapter 8. The technique of using a WAIT list or "agenda" to control the sequencing of coroutines, as we have done in this section, is called *quasi-parallel processing*.

It is quite difficult to give a precise analysis of the running time of such a long program, because of the complex interactions involved; it is easy to time various smaller parts of the program (like the INSERT subroutine) and this gives an indication of its efficiency. It is often useful to employ a special trace routine which executes the program, and which records how often each instruction was performed; this shows the "bottlenecks" in the program, places which should be given special attention. The author made such an experiment with the above program; the program ran for 10000 units of simulated time, and 26 men entered the simulated system. The instructions in the SORTIN loop, lines 73–75, were executed by far the most often, 1432 times, while the SORTIN subroutine itself was called 437 times. The CYCLE routine was performed 407 times, and this suggests that the DELETEW subroutine should not have been called at line 95; the four lines of that subroutine should have been written out in full (to save $4u$ each time CYCLE is used). The special trace routine also showed that the DECISION subroutine was called only 32 times and the loop in E4 (lines 216–218) was executed only 142 times.

It is hoped that some reader will learn as much about simulation from the above example as the author learned about elevators while the example was being prepared.

EXERCISES

1. [*21*] Give specifications for the insertion and deletion of information at the left end of a doubly linked list represented as in (1). (With the dual operations at the right end, which are obtained by symmetry, we therefore have all the actions of a general deque.)

▶ **2.** [*22*] Explain why a list that is singly linked cannot allow efficient operation as a general deque; the deletion of items can be done efficiently at only one end of a singly linked list.

▶ **3.** [*22*] The elevator system described in the text uses three call variables, `CALLUP`, `CALLCAR`, and `CALLDOWN`, for each floor, representing what buttons have been pushed by the men in the system. It is conceivable that internally the elevator needs only one or two relay circuits (i.e., binary variables) for the call buttons on each floor, instead of three. Show how a man could push buttons in a certain sequence with this elevator system to *prove* that there are three separate relays for each floor (except the top and bottom floors).

4. [*24*] Activity E9 in the elevator coroutine is usually canceled by step E6, and even when it hasn't been canceled, it doesn't do very much. Explain under what circumstances the elevator would behave differently (i.e., it would operate at a different speed, or visit floors in a different order) if activity E9 were deleted from the system.

5. [*20*] In Table 1, man no. 10 arrived on floor 0 at time 1048. Suppose he had arrived on floor 2 instead of floor 0; show that under these conditions the elevator would have gone *up* after receiving its passengers on floor 1, instead of down, in spite of the fact that man no. 8 wants to go down to floor 0.

6. [*23*] Note that in Table 1, time 1183–1233, men nos. 7, 8, and 9 all get in the elevator on floor 1; then the elevator goes down to floor 0 and only man no. 8 gets out. Now the elevator stops on floor 1, presumably to pick up men nos. 7 and 9 who are already aboard, and nobody is actually on floor 1 waiting to get in. (This situation occurs not infrequently at Caltech; if you get on the elevator going the wrong way, you must wait for an extra stop as you go by your original floor again.) In many elevator systems, men nos. 7 and 9 would not have gotten in the elevator at time 1183, since lights outside the elevator would show it was going down, not up; these men would have waited until the elevator came back up and stopped for them. On the system described, there are no such lights and it is impossible to tell which way the elevator is going to go until you are in it; hence Table 1 reflects the actual situation.

What changes should be made to coroutines M and E if we were to simulate the same elevator system, but with indicator lights, so that people do not get on the elevator when its state is contrary to their desired direction?

7. [*25*] Although "bugs" in programs are often embarrassing to a programmer, if we are to learn from our mistakes we should record them and tell other people about them instead of forgetting them! The following error (among others) was made by the author when he first wrote the program in this section: line 154 said "`JANZ CYCLE`" instead of "`JANZ M4A`". The reasoning was that if indeed the elevator had arrived at this man's floor and he would soon be able to get on, there was no need for him to perform his "give up" activity M4 any more, so we could simply go to `CYCLE` and continue simulating other activities. What is the error?

8. [*22*] Write the code for step E8, lines 277–292, which have been omitted from the program in the text.

9. [*23*] Write the code for the DECISION subroutine which has been omitted from the program in the text.

10. [*40*] It is perhaps significant to note that although the author had used the elevator system for years and thought he knew it well, it wasn't until he attempted to write this section that he realized there were quite a few facts about the elevator's system of choosing directions that he did not know. He went back to experiment with the elevator six separate times, each time believing he had finally achieved a complete understanding of its *modus operandi*. (Now he is reluctant to ride it for fear some new facet of its operation will appear, contradicting the algorithms given.) We often fail to realize how little we know about a thing until we attempt to simulate it on a computer.

Try to specify the actions of some elevator you are familiar with. Check the algorithm by experiments with the elevator itself (looking at its circuitry is not fair!); then design a discrete simulator for the system and run it on a computer.

▸ **11.** [*25*] (An "update-memory.") The following problem often arises in *synchronous* simulations: The system has n variables V[1], . . . , V[n], and at every simulated step new values for some of these are calculated from the old values. These calculations are assumed done "simultaneously" in the sense that the variables do not change to their new values until after all assignments have been made. Thus, the two statements

$$\texttt{V[1]} \leftarrow \texttt{V[2]} \qquad \text{and} \qquad \texttt{V[2]} \leftarrow \texttt{V[1]}$$

appearing at the same simulated time would interchange the values of V[1] and V[2], and this is quite different from what would happen in a sequential calculation.

The desired action can of course be simulated by keeping an additional table NEWV[1], . . . , NEWV[n]. Before each simulated step, we could set NEWV[k] ← V[k] for $1 \le k \le n$, then record all changes of V[k] in NEWV[k], and finally, after the step we could set V[k] ← NEWV[k], $1 \le k \le n$. But this "brute force" approach is often not completely satisfactory, for the following reasons: (1) Often n is very large, but the number of variables changed per step is rather small. (2) The variables are often not arranged in a nice table V[1], . . . , V[n], but are scattered throughout memory in a rather chaotic fashion. (3) This method does not detect the situation (usually an error in the model) when one variable is given two values in the same simulated step.

Assuming that the number of variables changed per step is rather small, design an efficient algorithm that simulates the desired actions, using two auxiliary tables NEWV[k] and LINK[k], $1 \le k \le n$. If possible, your algorithm should give an error stop if the same variable is being given two different values on the same step.

▸ **12.** [*22*] Why is it a good idea to use doubly linked lists instead of singly linked or sequential lists in the simulation program of this section?

2.2.6. Arrays and Orthogonal Lists

One of the simplest generalizations of a linear list is a two-dimensional or higher-dimensional array of information. For example, consider the case of an $m \times n$ matrix

$$\begin{pmatrix} \mathtt{A}[1,1] & \mathtt{A}[1,2] & \ldots & \mathtt{A}[1,n] \\ \mathtt{A}[2,1] & \mathtt{A}[2,2] & \ldots & \mathtt{A}[2,n] \\ \vdots & & & \vdots \\ \mathtt{A}[m,1] & \mathtt{A}[m,2] & \ldots & \mathtt{A}[m,n] \end{pmatrix}. \tag{1}$$

In this two-dimensional array, each node A[j, k] belongs to two linear lists: the "row j" list A[j, 1], A[j, 2], . . . , A[j, n], and the "column k" list A[1, k], A[2, k], . . . , A[m, k]. These orthogonal row and column lists essentially account for the two-dimensional structure of a matrix. Similar remarks apply to higher-dimensional arrays of information.

Sequential Allocation. When an array is stored in *sequential* memory locations, storage is usually allocated so that

$$\texttt{LOC(A[J,K])} = a_0 + a_1\texttt{J} + a_2\texttt{K}, \tag{2}$$

where a_0, a_1, and a_2 are constants. Let us consider a more general case: Suppose we have a four-dimensional array with one-word elements Q[I,J,K,L] for $0 \le \texttt{I} \le 2$, $0 \le \texttt{J} \le 4$, $0 \le \texttt{K} \le 10$, $0 \le \texttt{L} \le 2$. We would like to allocate storage so that

$$\texttt{LOC(Q[I,J,K,L])} = a_0 + a_1\texttt{I} + a_2\texttt{J} + a_3\texttt{K} + a_4\texttt{L}. \tag{3}$$

This means that a change in I, J, K, or L leads to a readily calculated change in the location of Q[I,J,K,L]. The most natural (and most commonly used) way to allocate storage is to let the array appear in memory in the "lexicographic order" of its indices, sometimes called "row major order":

Q[0, 0, 0, 0], Q[0, 0, 0, 1], Q[0, 0, 0, 2], Q[0, 0, 1, 0], Q[0, 0, 1, 1], . . . ,
Q[0, 0, 10, 2], Q[0, 1, 0, 0], . . . , Q[0, 4, 10, 2], Q[1, 0, 0, 0], . . . ,
Q[2, 4, 10, 2].

It is easy to see that this order satisfies the requirements of (3), and we have

$$\texttt{LOC(Q[I, J, K, L])} = \texttt{LOC(Q[0, 0, 0, 0])} + 165\texttt{I} + 33\texttt{J} + 3\texttt{K} + \texttt{L}. \tag{4}$$

In general, given a k-dimensional array with c-word elements A[$\texttt{I}_1$, $\texttt{I}_2$, . . . , $\texttt{I}_k$] for $0 \le \texttt{I}_1 \le d_1$, $0 \le \texttt{I}_2 \le d_2$, . . . , $0 \le \texttt{I}_k \le d_k$, we can store it in memory as

$$\begin{aligned}\texttt{LOC(A[I}_1, \texttt{I}_2, \ldots, \texttt{I}_k\texttt{])} &= \texttt{LOC(A[0, 0, . . . , 0])} \\ &\quad + c(d_2+1)\cdots(d_k+1)\texttt{I}_1 + \cdots \\ &\quad + c(d_k+1)\texttt{I}_{k-1} + c\texttt{I}_k \\ &= \texttt{LOC(A[0, 0, . . . , 0])} + \sum_{1\le r\le k} a_r\texttt{I}_r,\end{aligned}$$

where

$$a_r = c \prod_{r<s\le k} (d_s + 1). \tag{5}$$

To see why this formula works, observe that a_r is the amount of memory needed to store the subarray A[$\texttt{I}_1$, . . . , $\texttt{I}_r$, $\texttt{J}_{r+1}$, . . . , $\texttt{J}_k$] if $\texttt{I}_1$, . . . , $\texttt{I}_r$ are constant and $\texttt{J}_{r+1}$, . . . , $\texttt{J}_k$ vary through all values $0 \le \texttt{J}_{r+1} \le d_{r+1}$, . . . , $0 \le \texttt{J}_k \le d_k$; hence by the nature of lexicographic order the address of A[$\texttt{I}_1$, . . . , $\texttt{I}_k$] should change by precisely this amount when $\texttt{I}_r$ changes by 1.

Note the similarity between formula (5) and the value of the number $\mathtt{I}_1\ \mathtt{I}_2 \ldots \mathtt{I}_k$ in a mixed-radix number system. For example, if we had the array TIME[W, D, H, M, S] with $0 \le \mathtt{W} < 4$, $0 \le \mathtt{D} < 7$, $0 \le \mathtt{H} < 24$, $0 \le \mathtt{M} < 60$, and $0 < \mathtt{S} < 60$, the location of TIME[W, D, H, M, S] would be the location of TIME[0, 0, 0, 0, 0] plus the quantity "W weeks + D days + H hours + M minutes + S seconds" converted to seconds. Of course, it takes a pretty large computer and a pretty fancy application to make use of an array which has 2419200 elements.

The above method for storing arrays is generally suitable when the array has a complete rectangular structure, i.e., when all elements $\mathtt{A}[\mathtt{I}_1, \mathtt{I}_2, \ldots, \mathtt{I}_k]$ are present for indices in the independent ranges $l_1 \le \mathtt{I}_1 \le u_1$, $l_2 \le \mathtt{I}_2 \le u_2$, $\ldots$, $l_k \le \mathtt{I}_k \le u_k$. There are many situations in which this is not the case; most common among these is the *triangular matrix*, where we want to store only the entries A[j, k] for, say, $0 \le k \le j \le n$:

$$\begin{pmatrix} \mathtt{A}[0,0] & & & \\ \mathtt{A}[1,0] & \mathtt{A}[1,1] & & \\ \vdots & & & \\ \mathtt{A}[n,0] & \mathtt{A}[n,1] & \ldots & \mathtt{A}[n,n] \end{pmatrix} \tag{6}$$

We may know that all other entries are zero, or that $\mathtt{A}[j,k] = \mathtt{A}[k,j]$, so only half of the values need to be stored. If we want to store the lower triangular matrix (6) in $\frac{1}{2}(n+1)(n+2)$ consecutive memory positions, we are forced to give up the possibility of linear allocation as in Eq. (2), but we can now ask instead for an allocation arrangement of the form

$$\mathtt{LOC(A[J,K])} = a_0 + f_1(\mathtt{J}) + f_2(\mathtt{K}) \tag{7}$$

where f_1 and f_2 are functions of one variable. (The constant a_0 may be absorbed into either f_1 or f_2 if desired.) When the addressing has the form (7), a random element A[j, k] can be quickly accessed if we keep two (rather short) auxiliary tables of the values of f_1 and f_2, so that these functions need to be calculated only once.

It turns out that lexicographic order of indices for the array (6) satisfies condition (7), and, assuming one-word entries, we have in fact the rather simple formula

$$\mathtt{LOC(A[J,K])} = \mathtt{LOC(A[0,0])} + \frac{\mathtt{J(J+1)}}{2} + \mathtt{K}. \tag{8}$$

There is a far better way to store triangular matrices if we are fortunate enough to have two of them with the same size. If A[j, k] and B[j, k] are both to be stored for $0 \le k \le j \le n$, we can interlace them within a single matrix C[j, k] for $0 \le j \le n$, $0 \le k \le n+1$, using the convention

$$\mathtt{A}[j,k] = \mathtt{C}[j,k], \qquad \mathtt{B}[j,k] = \mathtt{C}[k,j+1]. \tag{9}$$

Thus

$$\begin{pmatrix} \mathtt{C[0,0]} & \mathtt{C[0,1]} & \mathtt{C[0,2]} & \ldots & \mathtt{C[0,}n+1\mathtt{]} \\ \mathtt{C[1,0]} & \mathtt{C[1,1]} & \mathtt{C[1,2]} & \ldots & \mathtt{C[1,}n+1\mathtt{]} \\ \vdots & & & & \vdots \\ \mathtt{C[}n\mathtt{,0]} & \mathtt{C[}n\mathtt{,1]} & \mathtt{C[}n\mathtt{,2]} & \ldots & \mathtt{C[}n\mathtt{,}n+1\mathtt{]} \end{pmatrix}$$

$$\equiv \begin{pmatrix} \mathtt{A[0,0]} & \mathtt{B[0,0]} & \mathtt{B[1,0]} & \ldots & \mathtt{B[}n\mathtt{,0]} \\ \mathtt{A[1,0]} & \mathtt{A[1,1]} & \mathtt{B[1,1]} & \ldots & \mathtt{B[}n\mathtt{,1]} \\ \vdots & & & & \vdots \\ \mathtt{A[}n\mathtt{,0]} & \mathtt{A[}n\mathtt{,1]} & \mathtt{A[}n\mathtt{,2]} & \ldots & \mathtt{B[}n\mathtt{,}n\mathtt{]} \end{pmatrix}$$

The two triangular matrices are packed together tightly within the space of $(n+1)(n+2)$ locations, and we have linear addressing as in (2).

The generalization of triangular matrices to higher dimensions is called a *tetrahedral array*. This interesting topic is the subject of exercises 6 through 8.

As an example of typical programming techniques for use with sequentially stored arrays, see exercise 1.3.2–10 and the two answers given for that exercise. The fundamental techniques for efficient traversal of rows and columns, as well as the uses of sequential stacks, are of particular interest within those programs.

Linked Allocation. Linked memory allocation also applies to higher-dimensional arrays of information in a natural way. In general, our nodes can contain k link fields, one for each list the node is in. The use of linked memory is generally for cases in which the arrays are not strictly rectangular in character.

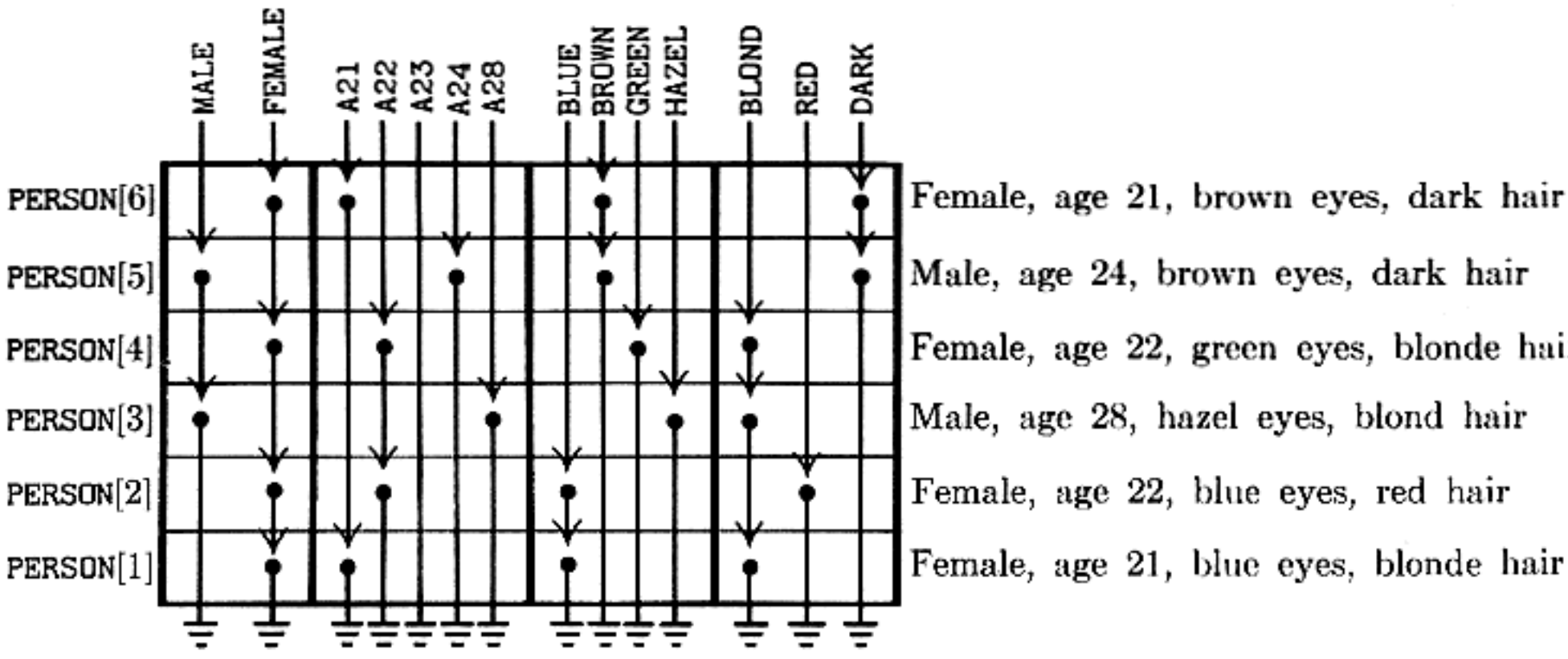


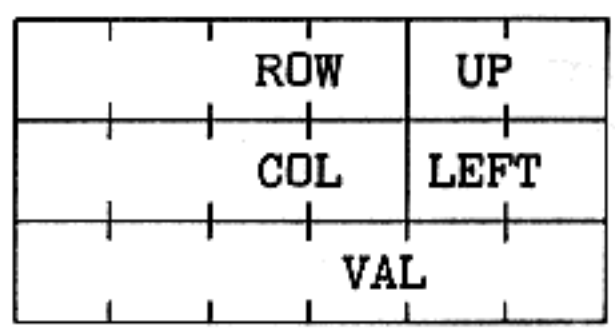
Fig. 13. Each node in four different lists.

As an example, suppose that we have a list in which every node is to represent a person, and that there are four link fields, SEX, AGE, EYES, and HAIR. In the EYES field we link together all nodes with the same eye color, etc. (See Fig. 13.) It is easy to visualize algorithms for inserting new people into this list. (Deletion would be harder, without double linking.) We can also conceive of algorithms

of varying degrees of efficiency for doing things like "Find all blue-eyed blonde women of ages 21 through 23"; see exercises 9 and 10. Problems in which each node of a list is to reside in several kinds of other lists at once arise rather frequently; indeed, the elevator system simulation described in the preceding section has nodes which are in both the QUEUE and WAIT lists at once.

As a detailed example of the use of linked allocation for orthogonal lists, we will consider the case of *sparse matrices* (i.e., matrices of large order in which most of the elements are zero). The goal is to operate on these matrices as though the entire matrix were present, but to save great amounts of memory space because the zero entries need not be represented. One way to do this, intended for random references to elements of the matrix, would be to use the storage-and-retrieval methods of Chapter 6, to find A[j, k] from the key "j, k"; however, there is another way to deal with sparse matrices that is usually preferable because it reflects the matrix structure more appropriately, and this is the method we will discuss here.

The representation we will discuss consists of circularly linked lists for each row and column. Each node of the matrix contains three words and five fields:



(10)

Here ROW and COL are the row and column indices of the node; VAL is the value stored at that part of the matrix; LEFT and UP are links to the next nonzero entry to the left in the row, or upward in the column, respectively. There are special list head nodes, BASEROW[i] and BASECOL[j], for every row and column. These nodes are identified by

$$\texttt{COL(LOC(BASEROW[}i\texttt{]))} < 0 \qquad \text{and} \qquad \texttt{ROW(LOC(BASECOL[}j\texttt{]))} < 0.$$

As usual in a circular list, the LEFT link in BASEROW[i] is the location of the rightmost value in that row, and UP in BASECOL[j] is the lowest value in that column. For example, the matrix

$$\begin{pmatrix} 50 & 0 & 0 & 0 \\ 10 & 0 & 20 & 0 \\ 0 & 0 & 0 & 0 \\ -30 & 0 & -60 & 5 \end{pmatrix} \tag{11}$$

would be represented as shown in Fig. 14.

Using sequential allocation of storage, a 200 × 200 matrix would take 40000 words, and this is more memory than many computers have; but a suitably sparse 200 × 200 matrix can be represented as above even in MIX's 4000-word memory. (See exercise 11.) The amount of time taken to access a

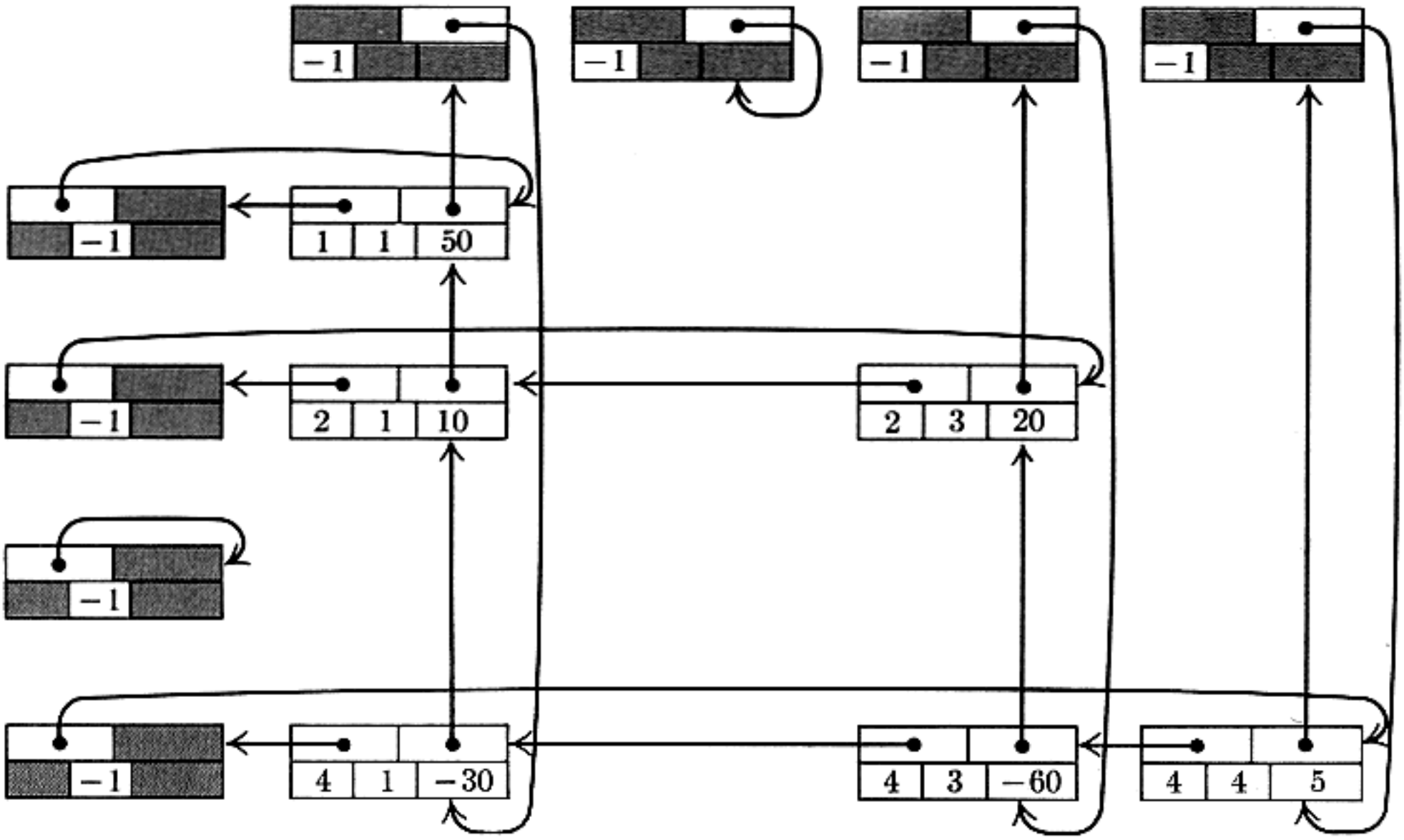
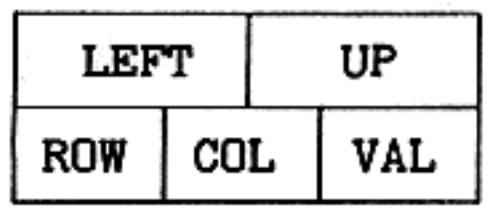


Fig. 14. Representation of the matrix (11); the nodes are illustrated in the format



List heads appear at the left and the top.

random element A[j, k] is also quite reasonable, *if* there are but few elements in each row or column; and since most matrix algorithms proceed by walking sequentially through a matrix, instead of accessing elements at random, this linked representation entails little loss of running speed.

As a typical example of a nontrivial algorithm dealing with sparse matrices in this notation, we will consider the *pivot step* operation, which is an important part of algorithms for solving linear equations, for inverting matrices, and for solving linear programming problems by the simplex method. A pivot step is the following matrix transformation:

$$\begin{array}{r} \\ \text{Pivot row} \\ \\ \text{Any other row} \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} & \text{Pivot column} & & \text{Any other column} \end{array} \\ \text{Before pivot step} \\ \begin{pmatrix} \vdots & \vdots & \\ \cdots\ a & \cdots\ b & \cdots \\ \vdots & \vdots & \\ \cdots\ c & \cdots\ d & \cdots \\ \vdots & \vdots & \end{pmatrix} \end{array} , \quad \begin{array}{c} \begin{array}{cccc} & \text{Pivot column} & & \text{Any other column} \end{array} \\ \text{After pivot step} \\ \begin{pmatrix} \vdots & \vdots & \\ \cdots\ 1/a & \cdots\ b/a & \cdots \\ \vdots & \vdots & \\ \cdots\ -c/a & \cdots\ d - \dfrac{bc}{a} & \cdots \\ \vdots & \vdots & \end{pmatrix} \end{array} . \tag{12}$$

It is assumed that the "pivot element" a is nonzero. For example, a pivot step applied to matrix (11), with the element 10 in row 2 column 1 as pivot, leads to

$$\begin{pmatrix} -5 & 0 & -100 & 0 \\ 0.1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 5 \end{pmatrix}. \tag{13}$$

Our goal is to design an algorithm which performs this pivot operation on sparse matrices that are represented as in Fig. 14. It is clear that the transformation (12) affects only those rows of a matrix for which there is a nonzero element in the pivot column, and it affects only those columns for which there is a nonzero entry in the pivot row. Hence when a large sparse matrix is being considered, we are not only achieving a reduction in space by the linked representation for nonzero elements, but are also perhaps achieving an increase in the speed of pivoting.

The pivoting algorithm is in many ways a straightforward application of linking techniques we have already discussed; in particular, it bears strong resemblances to Algorithm 2.2.4A for addition of polynomials. There are two things, however, which make the problem a little tricky: if in (12) we have $b \neq 0$ and $c \neq 0$ but $d = 0$, the sparse matrix representation has no entry for d and we must insert a new entry; and if $b \neq 0$, $c \neq 0$, $d \neq 0$, but $d - bc/a = 0$, we must delete the entry that was formerly there. These insertion and deletion operations in a two-dimensional array are more interesting than the one-dimensional case; to do them we must know what links are affected. Our algorithm processes the matrix rows successively from bottom to top. The efficient ability to insert and delete involves the introduction of a set of pointer variables `PTR[`j`]`, one for each column considered, which traverse the columns upwards and thereby give us the ability to update the proper links in both dimensions.

Algorithm S (*Pivot step in a sparse matrix*). Given a matrix represented as in Fig. 14, we perform the pivot operation (12). Assume that `PIVOT` is a link variable pointing to the pivot element. The algorithm makes use of an auxiliary table of link variables `PTR[`j`]`, one for each column of the matrix.

S1. [Initialize.] Set `IO` ← `ROW(PIVOT)`, `JO` ← `COL(PIVOT)`, `ALPHA` ← 1.0/`VAL(PIVOT)`, `VAL(PIVOT)` ← 1.0, `PO` ← `LOC(BASEROW[IO])`, `QO` ← `LOC(BASECOL[JO])`. (*Note:* The variable `ALPHA` and the `VAL` field of each node are assumed to be floating-point or rational quantities, while everything else in this algorithm has integer values.)

S2. [Process pivot row.] Set `PO` ← `LEFT(PO)`, `J` ← `COL(PO)`. If `J` < 0, go on to step S3 (the pivot row has been traversed). Otherwise set `PTR[J]` ← `LOC(BASECOL[J])` and `VAL(PO)` ← `ALPHA` × `VAL(PO)`, and repeat step S2.

S3. [Find new row.] Set `QO` ← `UP(QO)`. (The remainder of the algorithm deals successively with each row, from bottom to top, for which there is an entry

in the pivot column.) Set I ← ROW(Q0). If I < 0, the algorithm terminates. If I = I0, repeat step S3 (we have already done the pivot row). Otherwise set P ← LOC(BASEROW[I]), P1 ← LEFT(P). (The pointers P and P1 will now proceed across row I from right to left, as P0 goes in synchronization across row I0; Algorithm 2.2.4A is analogous. At this point,

$$P0 = LOC(BASEROW[I0]).)$$

S4. [Find new column.] Set P0 ← LEFT(P0), J ← COL(P0). If J < 0, set VAL(Q0) ← −ALPHA × VAL(Q0) and return to S3. If J = J0, repeat step S4. (Thus we process the pivot column entry in row I *after* all other column entries have been processed; the reason is that VAL(Q0) is needed in step S7.)

S5. [Find I, J element.] If COL(P1) > J, set P ← P1, P1 ← LEFT(P), and repeat step S5. If COL(P1) = J, go to step S7. Otherwise go to step S6 (we need to insert a new element in column J of row I).

S6. [Insert I, J element.] If ROW(UP(PTR[J])) > I, then set PTR[J] ← UP(PTR[J]), and repeat step S6. (Otherwise, we will have ROW(UP(PTR[J])) < I; the new element is to be inserted just after NODE(PTR[J]) in the vertical dimension, and after NODE(P) in the horizontal dimension.) Otherwise set X ⇐ AVAIL, VAL(X) ← 0, ROW(X) ← I, COL(X) ← J, LEFT(X) ← P1, UP(X) ← UP(PTR[J]), LEFT(P) ← X, UP(PTR[J]) ← X, P1 ← X, PTR[J] ← X.

S7. [Pivot.] Set VAL(P1) ← VAL(P1) − VAL(Q0) × VAL(P0). If now VAL(P1) = 0, go to S8. (*Note:* When floating-point arithmetic is being used, this test "VAL(P1) = 0" should be replaced by "|VAL(P1)| < EPSILON" or better yet by the condition "most of the significant figures of VAL(P1) were lost in the subtraction.") Otherwise, set PTR[J] ← P1, P ← P1, P1 ← LEFT(P), and go back to S4.

S8. [Delete I, J element.] If UP(PTR[J]) ≠ P1 (or, what is essentially the same thing, if ROW(UP(PTR[J])) > I), set PTR[J] ← UP(PTR[J]) and repeat step S8; otherwise, set UP(PTR[J]) ← UP(P1), LEFT(P) ← LEFT(P1), AVAIL ⇐ P1, P1 ← LEFT(P). (NODE(P1) has been deleted.) Go back to S4. ∎

The programming of this algorithm is left as a very instructive exercise for the reader (see exercise 15). It is worth pointing out here that it is necessary to allocate only one word of memory to each of the nodes BASEROW[i], BASECOL[j], since most of their fields are irrelevant. (See the shaded areas in Fig. 14, and see the program of Section 2.2.5.) Furthermore, the value −PTR[j] can be stored as ROW(LOC(BASECOL[j])) for additional storage space economy. The running time of Algorithm S is very roughly proportional to the number of matrix elements affected by the pivot operation.

EXERCISES

1. [*17*] Give a formula for `LOC(A[J,K])` if `A` is the matrix of (1), and if each node of the array is two words long, assuming that the nodes are stored consecutively in lexicographic order of the indices.

▸ **2.** [*21*] Formula (5) has been derived from the assumption $0 \le \mathtt{I}_r \le d_r$ for $1 \le r \le k$; give a general formula that applies to the case $l_r \le \mathtt{I}_r \le u_r$, where l_r and u_r are any lower and upper bounds on the dimensionality.

3. [*21*] The text considers lower triangular matrices A[j, k] for $0 \le k \le j \le n$. How can the discussion of such matrices readily be modified for the case that subscripts start at 1 instead of 0, i.e., the case that $1 \le k \le j \le n$?

4. [*22*] Show that if we store the *upper* triangular array A[j, k] for $0 \le j \le k \le n$ in lexicographic order of the indices, the allocation satisfies the condition of Eq. (7). Find a formula for `LOC(A[J,K])` in this case.

5. [*20*] Show that it is possible to bring the value of `A[J,K]` into register A in one `MIX` instruction, using the "indirect addressing" feature of exercise 2.2.2–3, even when `A` is a *triangular* matrix as in (8). (Assume that the values of `J` and `K` are in index registers.)

▸ **6.** [*M24*] Consider the "tetrahedral arrays" A[i, j, k], B[i, j, k], where $0 \le k \le j \le i \le n$ in `A`, and $0 \le i \le j \le k \le n$ in `B`. Suppose that both of these arrays are stored in consecutive memory locations in lexicographic order of the indices; show that `LOC(A[I,J,K])` $= a_0 + f_1(\mathtt{I}) + f_2(\mathtt{J}) + f_3(\mathtt{K})$ for certain functions f_1, f_2, f_3, but that `LOC(B[I,J,K])` *cannot* be expressed in such a simple form.

7. [*M23*] Find a general formula to allocate storage for the k-dimensional tetrahedral array A[$i_1, i_2, \ldots, i_k$], where $0 \le i_k \le \cdots \le i_2 \le i_1 \le n$.

8. [*33*] (P. Wegner.) Suppose that `A[I,J,K]`, `B[I,J,K]`, `C[I,J,K]`, `D[I,J,K]`, `E[I,J,K]`, and `F[I,J,K]` are six tetrahedral arrays that are to be stored in memory for $0 \le \mathtt{K} \le \mathtt{J} \le \mathtt{I} \le n$. Is there a neat way to accomplish this, analogous to (9) in the two-dimensional case?

9. [*22*] Suppose a table, like that indicated in Fig. 13, but much larger, has been set up so that all links go in the same direction as shown there (i.e., `LINK(X)` $<$ `X` for all nodes and links). Design an algorithm which finds the addresses of all blue-eyed blonde girls of ages 21 through 23, by going through the various link fields in such a way that upon completion of the algorithm at most one pass has been made through each of the lists `FEMALE`, `A21`, `A22`, `A23`, `BLOND`, and `BLUE`.

10. [*26*] Can you think of a better way to organize a personnel table so that searches as described in the previous exercise would be more efficient? (The answer to this exercise is *not* merely "yes" or "no.")

11. [*11*] Suppose that we have a 200×200 matrix in which there are at most four nonzero entries per row. How much storage is required to represent this matrix as in Fig. 14, if we use three words per node except for list heads, which will use one word?

▸ **12.** [*20*] What are `VAL(Q0)`, `VAL(P0)`, and `VAL(P1)` at the beginning of step S7, in terms of the notation a, b, c, d used in (12)?

▸ **13.** [*22*] Why were circular lists used in Fig. 14 instead of straight linear lists? Could Algorithm S be rewritten so that it does not make use of the circular linkage?

14. [*22*] Algorithm S actually saves pivoting time in a sparse matrix, since it avoids consideration of those columns in which the pivot row has a zero entry. Show that this savings in running time can be achieved in a large sparse matrix that is stored sequentially, with the help of an auxiliary table `LINK`[j], $1 \le j \le n$.

▸ **15.** [*29*] Write a MIXAL program for Algorithm S. Assume that the VAL field is a floating-point number, and that MIX's floating-point arithmetic operators FADD, FSUB, FMUL, and FDIV can be used for operations on this field. Assume for simplicity that FADD and FSUB return the answer zero when the operands added or subtracted cancel most of the significance, so that the test "VAL(P1) = 0" may be safely used in step S7.

16. [*25*] Design an algorithm to *copy* a sparse matrix. (In other words, the algorithm is to yield two distinct representations of a matrix in memory, having the form of Fig. 14, given just one such representation initially.)

17. [*26*] Design an algorithm to *multiply* two sparse matrices; given matrices A and B, form a new matrix C, where $C[i, j] = \sum_k A[i, k]B[k, j]$. The two input matrices and the output matrix should be represented as in Fig. 14.

18. [*22*] The following algorithm replaces a matrix by the inverse of that matrix, assuming that the entries are $A[i, j]$, for $1 \le i, j \le n$, and using "Gauss-Jordan reduction":

a) For $k = 1, 2, \ldots, n$ do the following: Search row k in all columns not yet used as a pivot column, to find the entry with the greatest absolute value; set $C[k]$ equal to the column in which this entry was found, and do a pivot step with this entry as pivot. (If all such entries are zero, the matrix is singular and has no inverse.)

b) Permute rows and columns so that what was row k becomes row $C[k]$, and what was column $C[k]$ becomes column k.

The problem in this exercise is to use the above algorithm to find the inverse of the matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix}$$

by hand calculation.

19. [*31*] Modify the "Gauss-Jordan reduction" algorithm described in exercise 18 so that it obtains the inverse of a sparse matrix that is represented in the form of Fig. 13. Pay special attention to making the row- and column-permutation operations of step (b) efficient.

2.3. TREES

We now turn to a study of trees, the most important nonlinear structures arising in computer algorithms. Generally speaking, tree structure means a "branching" relationship between nodes, much like that found in the trees of nature.

Let us define a *tree* formally as a finite set T of one or more nodes such that

a) There is one specially designated node called the *root* of the tree, root (T); and
b) The remaining nodes (excluding the root) are partitioned into $m \geq 0$ disjoint sets $T_1, \ldots, T_m$, and each of these sets in turn is a tree. The trees $T_1, \ldots, T_m$ are called the *subtrees* of the root.

The definition just given is recursive, i.e., we have defined a tree in terms of trees. Of course, there is no problem of circularity involved here, since trees with one node must consist of only the root, and trees with $n > 1$ nodes are defined in terms of trees with less than n nodes; hence the concept of a tree with two nodes, three nodes, or ultimately any number of nodes, is determined by the definition given. There are nonrecursive ways to define trees (for example, see exercises 10, 12, and 14, and Section 2.3.4), but a recursive definition seems most appropriate since recursion is an innate characteristic of tree structures. The recursive character of trees is present also in nature, since buds on young trees eventually grow into subtrees with buds of their own, etc. It is easy to give rigorous proofs of important facts about trees based on a recursive definition such as the one above, by using induction on the number of nodes in a tree; for example, see exercise 3.

It follows from our definition that every node of a tree is the root of some subtree contained in the whole tree. The number of subtrees of a node is called the *degree* of that node. A node of degree zero is called a *terminal node* or sometimes a "leaf." A nonterminal node is often called a *branch node*. The *level* of a node with respect to T is defined by saying that the root has level 1, and other nodes have a level that is one higher than they have with respect to the subtree of the root, T_j, which contains them.

These concepts are illustrated in Fig. 15, which shows a tree with seven nodes. The root is A, and it has the two subtrees $\{B\}$ and $\{C, D, E, F, G\}$. The tree $\{C, D, E, F, G\}$ has node C as its root. Node C is on level 2 with respect to the whole tree, and it has three subtrees $\{D\}$, $\{E\}$, and $\{F, G\}$;

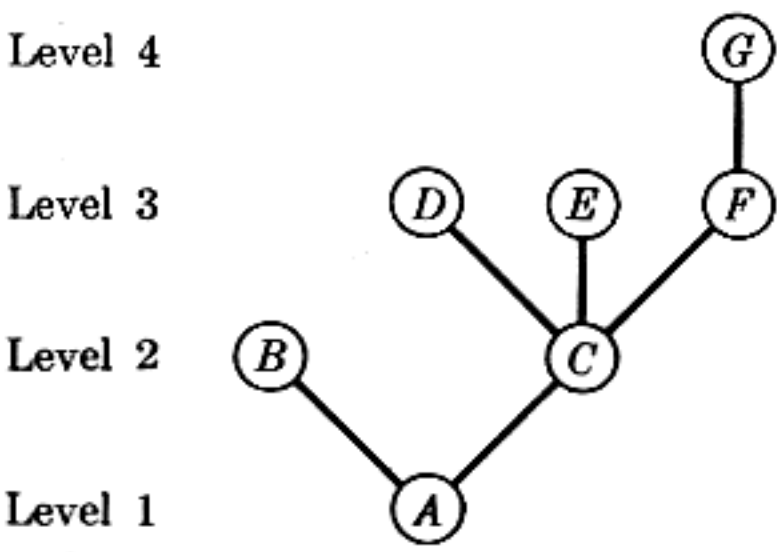


Fig. 15. A tree.

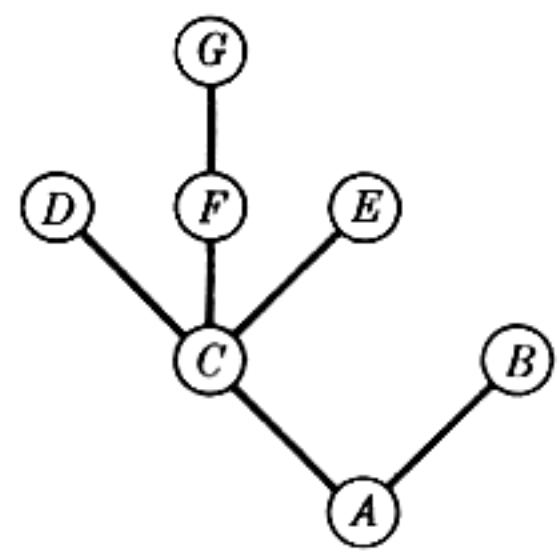


Fig. 16. Another tree.

therefore C has degree 3. The terminal nodes in Fig. 15 are B, D, E, and G; F is the only node with degree 1; G is the only node with level 4.

If the relative order of the subtrees $T_1, \ldots, T_m$ in (b) of the definition is important, we say the tree is an *ordered tree;* when $m \geq 2$ in an ordered tree, it makes sense to call T_2 the "second subtree" of the root, etc. If we do not care to regard two trees as different when they differ only in the respective ordering of subtrees of nodes, the tree is said to be *oriented*, since only the relative orientation of the nodes, not their order, is being considered. The very nature of computer representation defines an implicit ordering for any tree, so in most cases ordered trees are of greatest interest to us. We will therefore tacitly assume that *all trees we discuss are ordered, unless it is explicitly stated otherwise.* Accordingly, the trees of Figs. 15 and 16 will generally be considered to be different, although they would be the same as oriented trees.

A *forest* is a set (usually an ordered set) of zero or more disjoint trees. Another way to phrase part (b) of the definition of tree would be to say that *the nodes of a tree excluding the root form a forest.* (Some authors use the term "n-tuply rooted tree" to denote a forest of n trees.)

There is very little distinction between abstract forests and trees; if we delete the root of a tree, we have a forest, and, conversely, if we add just one node to any forest we get a tree. Therefore the words tree and forest are often used almost interchangeably during informal discussions about tree structures.

There are many ways to draw diagrams of trees. Besides the diagram of Fig. 15, three of the principal alternatives are shown in Fig. 17, depending on where the root is placed. It is not a frivolous joke to worry about how a tree structure is drawn in diagrams, since there are many occasions in which we would like to say one node is "above" or "higher than" another node, or to refer to the "rightmost" element, etc. Certain algorithms for dealing with tree structures have become known as "top down" methods, as opposed to "bottom up." This terminology leads to confusion unless we adhere to a uniform convention for drawing trees.

It may seem that the form of Fig. 15 would be preferable simply because that is how trees grow in nature; in the absence of any compelling reason to

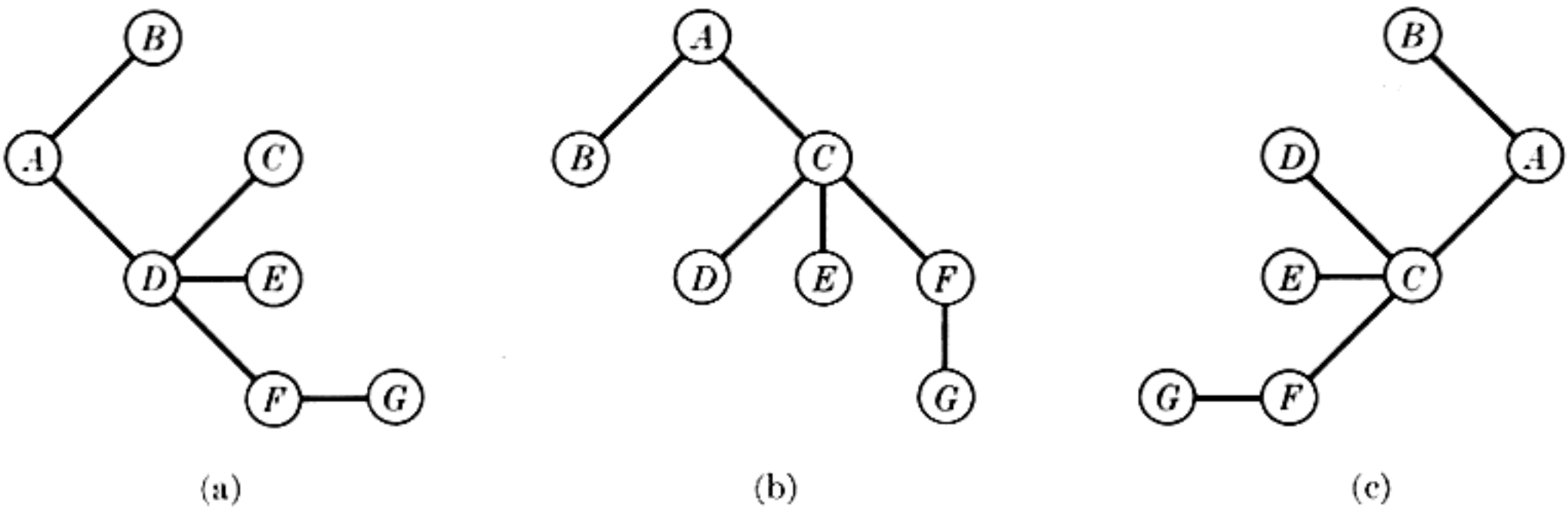


Fig. 17. How shall we draw a tree?

adopt any of the other three forms, we might as well adopt nature's time-honored tradition. With this in mind, the author consistently followed a root-at-the-bottom convention as the present set of books was being prepared, but after two years of trial it was found to be a mistake: observations of the computer literature and numerous informal discussions with computer scientists about a wide variety of algorithms showed that trees were drawn with the *root at the top* in over 80 percent of the cases examined. There is an overwhelming tendency to make hand-drawn charts grow downwards instead of upward (and this is easy to understand in view of the way we write); even the word "subtree" (as opposed to "supertree") tends to connote a downward relationship. From these considerations we conclude that *Fig. 15 is upside down.* Henceforth we will almost always draw trees as in Fig. 17(b), with the root at the top and leaves at the bottom. Until everyone is accustomed to trees with this orientation, however, it may be wise to refrain from using words that imply any vertical relationship, unless accompanying diagrams resolve the ambiguity.

It is necessary to have good descriptive terminology for talking about trees. Instead of the ambiguous references to "above" and "below" in a tree, genealogical words taken from the terminology of *family trees* have been found very appropriate for this purpose. Figure 19 shows two common types of family trees. The two types are obviously quite different: a "pedigree" shows the ancestors of a given individual, while a "lineal chart" shows the descendants.

If "cross-breeding" occurs, a family tree is not really a tree, because different branches of a tree (as we have defined it) can never be joined together. To compensate for this discrepancy, note that Queen Victoria and Prince Albert appear twice in the sixth generation in Fig. 19(a), and that King Christian IX actually appears in both the fifth and sixth generations. The family tree is a true tree if each of its nodes represents, not a person, but "a person in his role as parent of so-and-so."

Standard terminology for tree structures is taken from the *second* form of family tree, the lineal chart: Each root is said to be the *father* of the roots of its subtrees, and the latter are said to be *brothers*, and they are *sons* of their father. The root of the entire tree has no father. For example, in Fig. 18, C has three sons, D, E, and F; E is the father of G; B and C are brothers. Extension of this terminology (e.g., B is an uncle of F; A is the great-grandfather of G; H and F are first cousins) is clearly possible. Some authors use the feminine designations "mother, daughter, sister" instead of "father, son, brother"; but for some reason the masculine words seem more professional. Other authors, wishing to promote equality

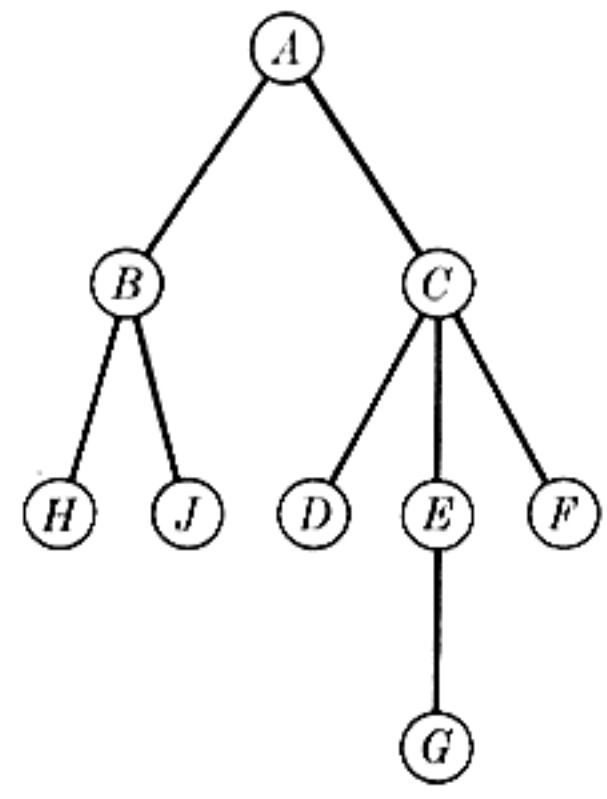


Fig. 18. Conventional tree diagram.

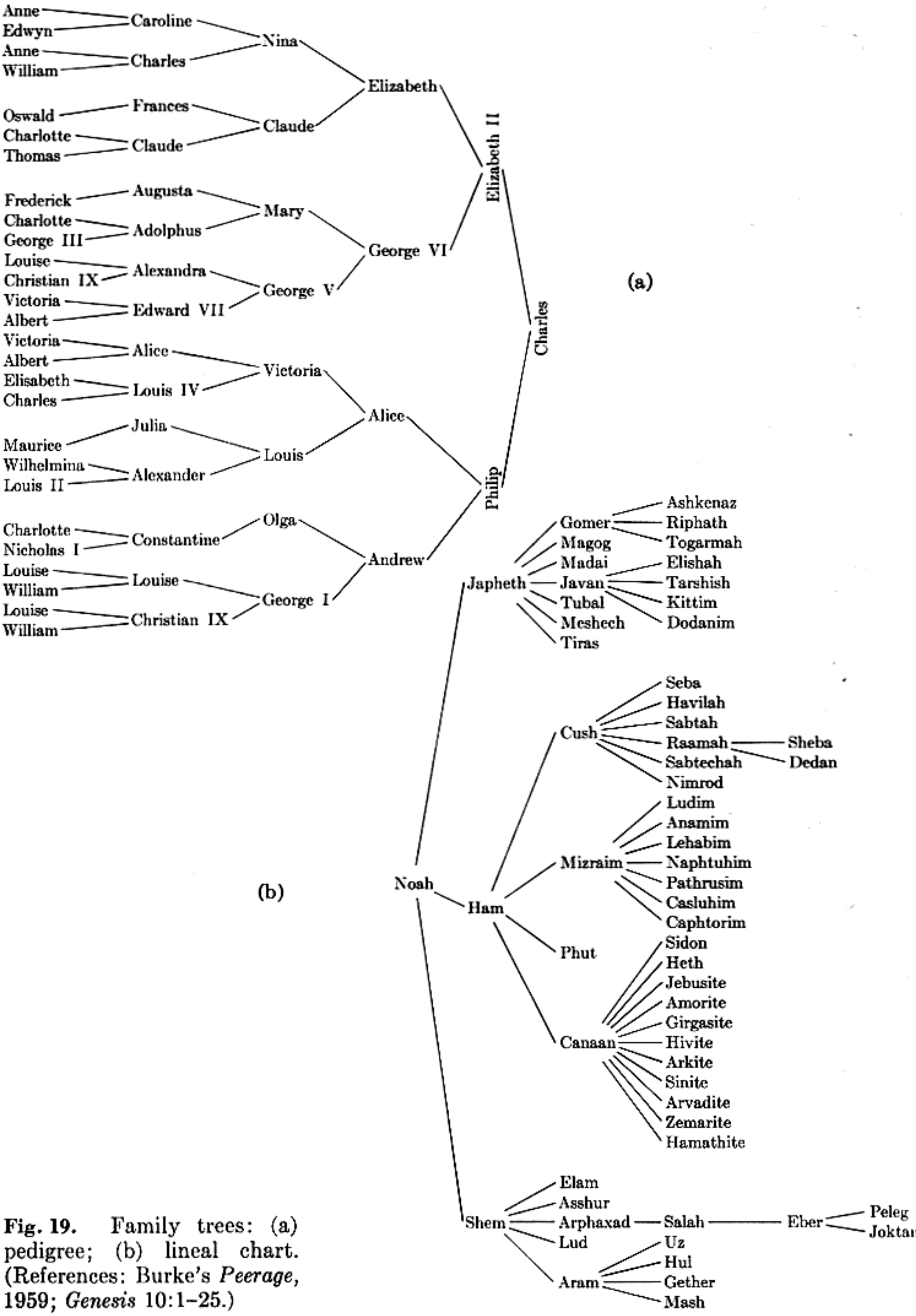
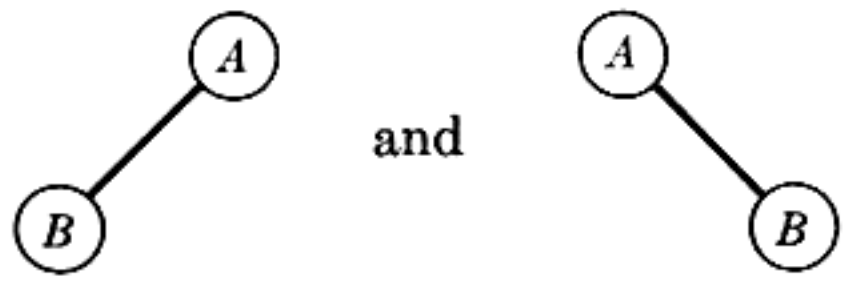


Fig. 19. Family trees: (a) pedigree; (b) lineal chart. (References: Burke's *Peerage*, 1959; *Genesis* 10:1–25.)

of the sexes, use the neutral words "parent, offspring, sibling" instead. In any case we use the words *ancestor* and *descendant* to denote a relationship that may span several levels of the tree: The descendants of C in Fig. 18 are D, E, F, and G; the ancestors of G are A, C, and E.

Figure 19(a) is an example of a *binary tree*, which is an important type of tree structure. The reader has undoubtedly seen other examples of binary trees in connection with tennis tournaments, etc. In a binary tree each node has at most two subtrees, and when there is only one subtree present, we distinguish between the left and right subtree. More formally, let us define a binary tree as *a finite set of nodes which either is empty, or consists of a root and two disjoint binary trees* called the left and right subtrees of the root.

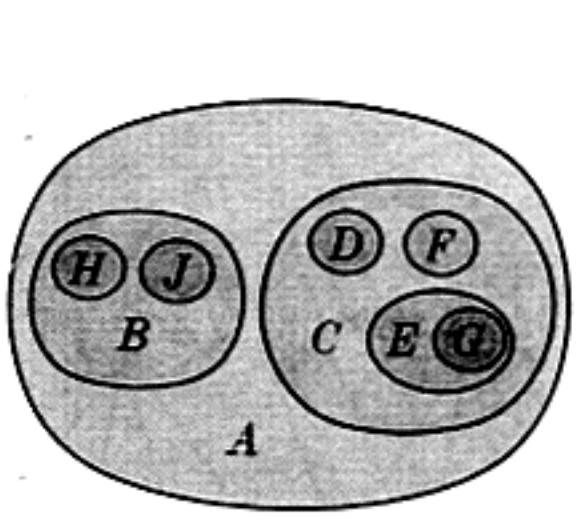
This recursive definition of binary tree should be studied carefully. Note that a binary tree is *not* a special case of a tree; it is another concept entirely (although we will see many relations between the two concepts). For example, the binary trees



(1)

are distinct (the root has an empty left subtree in one case and a nonempty left subtree in the other), although as trees they would be identical. Therefore we will always be careful to use the word "binary" to distinguish between binary trees and ordinary trees. Some authors define binary tree in a slightly different manner (see exercise 20).

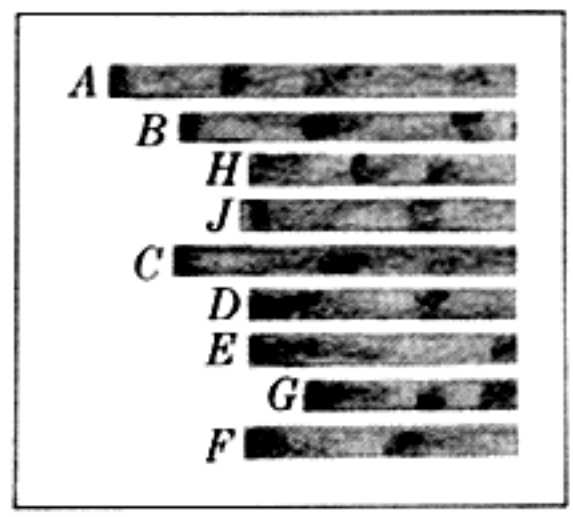
Tree structure can be represented graphically in several other ways bearing no resemblance to actual trees. Figure 20 shows three diagrams which reflect the structure of Fig. 18: Figure 20(a) essentially represents Fig. 18 as an *oriented* tree; this diagram is a special case of the general idea of *nested sets*, i.e., a collection of sets in which any pair of sets is either disjoint or one contains the other. (See exercise 10.) Part (b) of the figure shows nested sets in a line,



(a)

$(A(B(H)(J))(C(D)(E(G))(F)))$

(b)



(c)

Fig. 20. Further ways to show tree structure: (a) Nested sets; (b) Nested parentheses; (c) Indentation.

much as part (a) shows them in a plane; in part (b) the ordering of the tree is also indicated. Part (b) may also be regarded as an outline of an algebraic formula involving "nested parentheses." Part (c) shows still another common way to represent tree structure, using *indentation*. The number of different representation methods in itself is ample evidence for the importance of tree structures in everyday life as well as in computer programming. Any hierarchical classification scheme leads to a tree structure.

The similarity between an indented list like Fig. 20(c) and outlines or tables of contents in books is noteworthy. This book itself has a tree structure: the tree structure of Chapter 2 is shown in Fig. 21. Here we notice a significant idea: *the method used to number sections in this book is another way to specify tree structure.* Such a method is often called "Dewey decimal notation" for trees, by analogy with the similar classification scheme of this name used in libraries. The Dewey decimal notation for the tree of Fig. 18 is

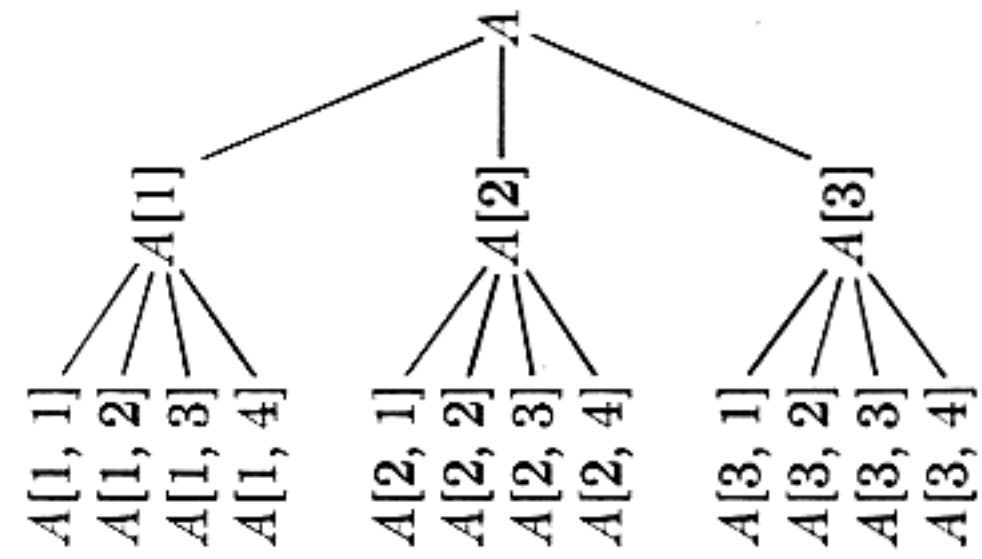
$$1\ A;\quad 1.1\ B;\quad 1.1.1\ H;\quad 1.1.2\ J;\quad 1.2\ C;\quad 1.2.1\ D;\quad 1.2.2\ E;$$
$$1.2.2.1\ G;\quad 1.2.3\ F.$$

Dewey decimal notation applies to any forest: the root of the kth tree in the forest is given number k; and if α is the number of any node of degree m, its sons are numbered $\alpha.1, \alpha.2, \ldots, \alpha.m$. The Dewey decimal notation satisfies many simple mathematical properties, and it is a useful tool in the analysis of trees. One example of this is the natural sequential ordering it gives to the nodes of an arbitrary tree, analogous to the ordering of sections within this book.

There is an intimate relation between Dewey decimal notation and the notation for indexed variables which we have already been using extensively. If F is a forest of trees, we may let $F[1]$ denote the first tree, $F[1][2] \equiv F[1, 2]$ the second subtree of the root of this first tree, $F[1, 2, 1]$ the first subtree of the latter, etc.; node $a.b.c.d$ in Dewey decimal notation is root($F[a, b, c, d]$). This notation is an extension of the index notation in that the admissible range of each index depends on the values in the preceding index positions.

We see that, in particular, any rectangular array can be thought of as a special case of a tree structure. For example, here are two representations of a 3×4 matrix:

$$\begin{pmatrix} A[1,1] & A[1,2] & A[1,3] & A[1,4] \\ A[2,1] & A[2,2] & A[2,3] & A[2,4] \\ A[3,1] & A[3,2] & A[3,3] & A[3,4] \end{pmatrix}$$



It is important to observe, however, that this tree structure does not faithfully

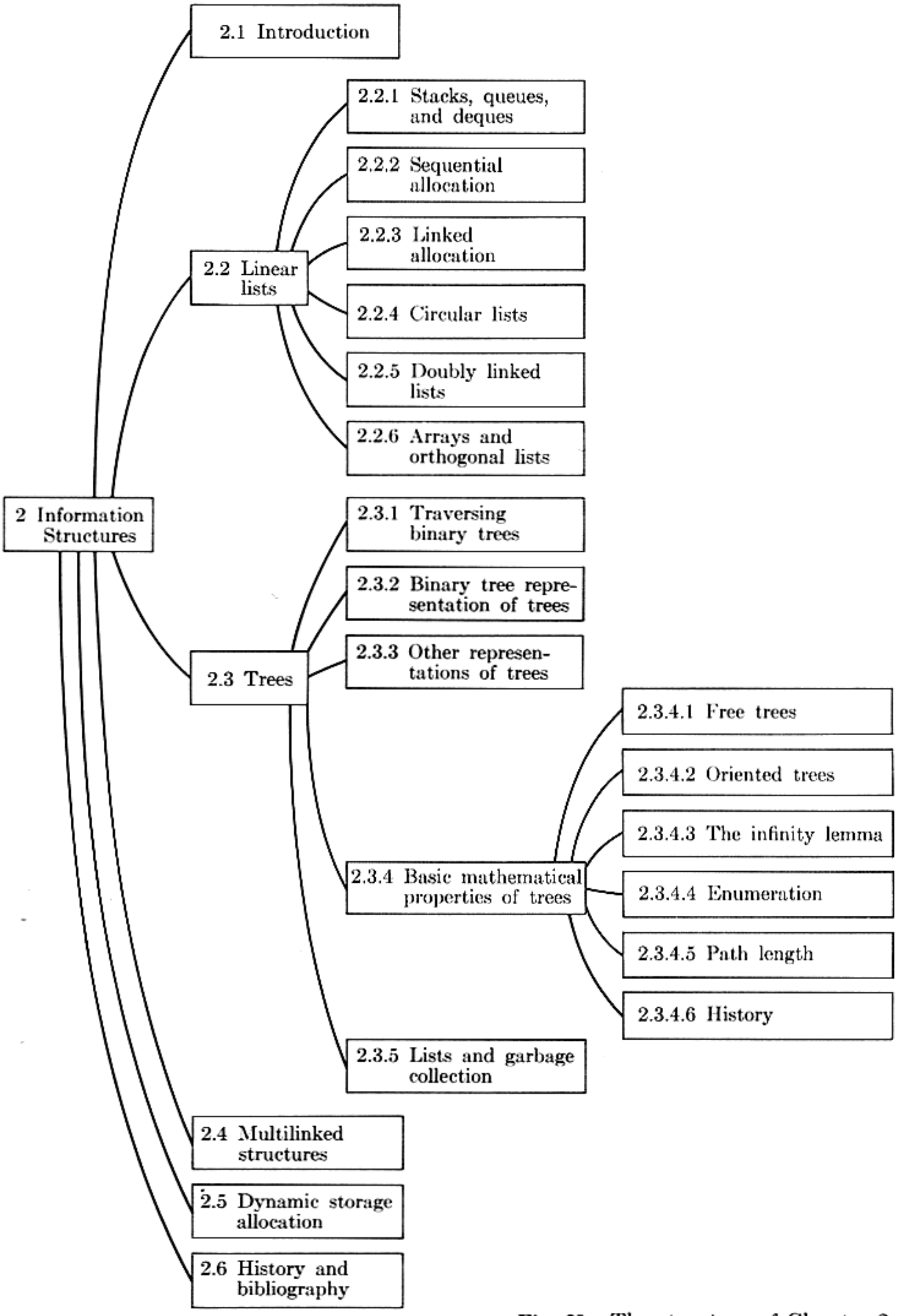


Fig. 21. The structure of Chapter 2.

reflect all of the matrix structure; the row relationships appear explicitly in the tree but the column relationships do not.

Algebraic formulas provide us with another example of tree structure. Figure 22 shows a tree corresponding to the arithmetic expression

$$a - b(c/d + e/f). \tag{2}$$

The connection between formulas and trees is very important in applications, as we shall see later (especially in Chapters 10 and 12).

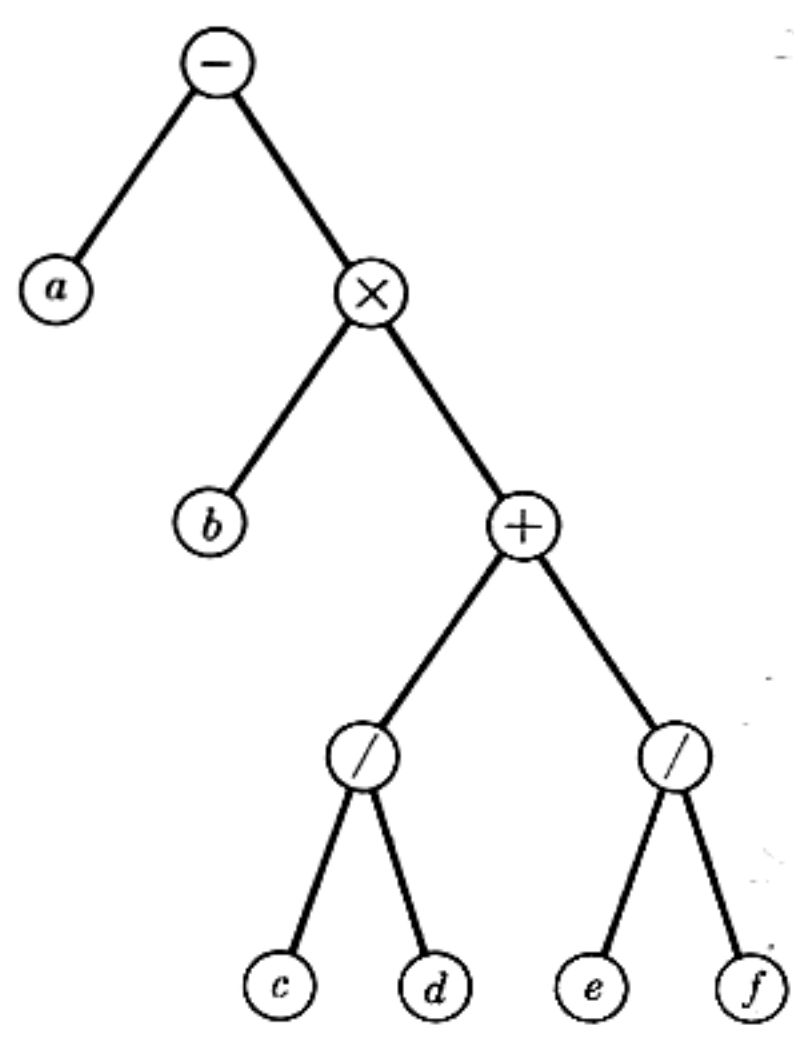


Fig. 22. Tree representing formula (2).

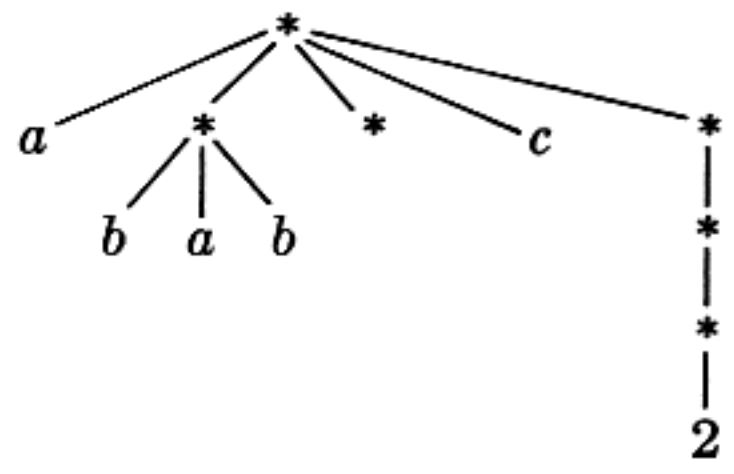
In addition to trees, forests, and binary trees, there is a fourth type of structure, commonly called a "list structure," which is closely related to both of the former two. The word "list" is being used here in a very technical sense, and to distinguish this use of the word we will always capitalize it, "List." A List is defined (recursively) as *a finite sequence of zero or more atoms or Lists.* Here "atom" is an undefined concept referring to elements from any universe of objects that might be desired, so long as it is possible to distinguish an atom from a List. By means of an obvious notational convention involving commas and parentheses, we can distinguish between atoms and Lists and we can conveniently display the ordering within a List. As an example, consider

$$L = (a,\ (b, a, b),\ (\ ),\ c,\ (((2)))) \tag{3}$$

which is a List with five elements: first the atom a, then the List (b, a, b), then the empty List "( )", then the atom c, and finally the List $(((2)))$. The latter List consists of the List $((2))$, which consists of the List (2), which consists of the atom 2.

The following tree structure corresponds to L:

$L =$



(4)

The asterisks in this diagram indicate the definition and appearance of a List,

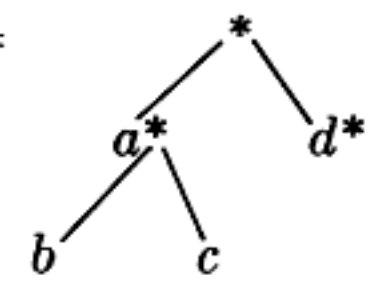
as opposed to the appearance of an atom. Index notation applies to Lists as it does to trees, e.g., $L[2] = (b, a, b)$, $L[2, 2] = a$.

No data is carried in the nodes for the Lists in (4) other than the fact that they are Lists. It is possible to label the elements of Lists with information, as we have done for trees and other structures; e.g.,

$$A = (a{:}(b, c), d{:}(\,))$$

would correspond to a tree we can draw as follows:

$A =$



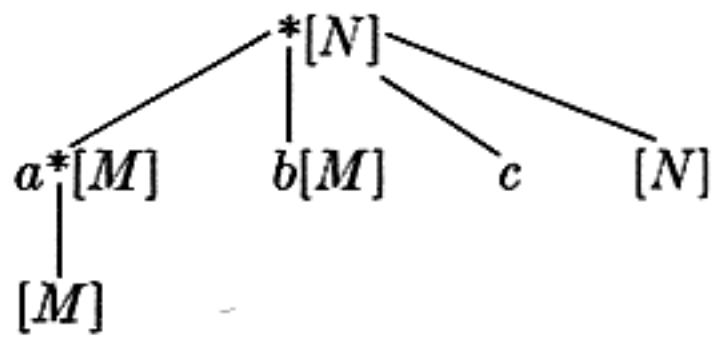
The big difference between Lists and trees is that Lists may overlap (i.e., sub-Lists need not be disjoint) and they may even be recursive (may contain themselves). The List

$$M = (M) \tag{5}$$

corresponds to no tree structure, nor does the List

$$N = (a{:}M, b{:}M, c, N). \tag{6}$$

(In these examples, capital letters refer to Lists, lower case letters to labels and atoms.) We might diagram (5) and (6) as follows:



(7)

Actually, Lists are not so complicated as the above examples might indicate; they are, in essence, a rather simple generalization of linear lists such as we have considered in Section 2.2, with the additional proviso that the elements of the linear lists may be link variables which point to other linear lists (and possibly to themselves).

Summary: Four kinds of closely related information structures (trees, forests, binary trees, Lists) arise from many sources, and they are therefore important in computer algorithms. We have seen various methods of diagraming these structures, and we have considered some notations and terminology which is useful in talking about them. The following sections develop these ideas in greater detail.

EXERCISES

1. [*18*] How many different trees are there with three nodes, A, B, and C?

2. [*20*] How many different *oriented* trees are there with three nodes, A, B, and C?

▶ **3.** [*M20*] Prove rigorously from the definitions that for every node X in a tree there is a unique "path up to the root," i.e., a unique sequence of $k \geq 1$ nodes $X_1, X_2, \ldots, X_k$ such that X_1 is the root of the tree, $X_k = X$, and X_j is the father of X_{j+1} for $1 \leq j < k$. (This proof will be typical of the proofs of nearly all the elementary facts about tree structures.) [*Hint:* Use induction on the number of nodes in the tree.]

4. [*01*] True or false: In a conventional tree diagram (root at the top), if node X has a *higher* level number than node Y, then node X appears *lower* in the diagram than node Y.

5. [*02*] If node A has three brothers and B is the father of A, what is the degree of B?

▶ **6.** [*21*] Define the statement "X is an mth cousin of Y, n times removed" as a meaningful relation between nodes X and Y of a tree, by analogy with family trees, if $m > 0$ and $n \geq 0$. (See a dictionary for the meaning of these terms in regard to family trees.)

7. [*23*] Extend the definition given in the previous exercise to all $m \geq -1$ and to all integers $n \geq -(m+1)$ in such a way that for any two nodes X and Y of a tree there are unique m and n such that X is an mth cousin of Y, n times removed.

8. [*03*] What binary tree is not a tree?

9. [*00*] In the two binary trees of (1), which node is the root (B or A)?

10. [*M20*] A collection of nonempty sets is said to be *nested* if, given any pair X, Y of the sets, either $X \subseteq Y$ or $X \supseteq Y$ or X and Y are disjoint. (In other words, $X \cap Y$ is either X, Y, or $\emptyset$.) Figure 20(a) indicates that any tree corresponds to a collection of nested sets; conversely, does every such collection correspond to a tree?

▶ **11.** [*HM32*] Extend the definition of tree to infinite trees by considering collections of nested sets as in exercise 10. Can the concepts of level, degree, father, and son be defined for each node of an infinite tree? Give examples of nested sets of real numbers which correspond to a tree in which

a) every node has uncountable degree and there are infinitely many levels;
b) there are nodes with uncountable level;
c) every node has degree at least 2 and there are uncountably many levels.

12. [*M23*] Under what conditions does a partially ordered set (cf. Section 2.2.3) correspond to an unordered tree or forest?

13. [*10*] Suppose that node X is numbered $a_1 \,.\, a_2 \,.\, \cdots \,.\, a_k$ in the Dewey decimal system; what are the numbers of the nodes in the path from X to the root (see exercise 3)?

14. [*M22*] Let S be any nonempty set of elements having the form "$1 \,.\, a_1 \,.\, \cdots \,.\, a_k$", where $k \geq 0$ and $a_1, \ldots, a_k$ are positive integers. Show that S specifies a tree when it is finite and satisfies the following condition: "If $\alpha \,.\, m$ is in the set, then so is

$\alpha . (m - 1)$ if $m > 1$, or α if $m = 1$." (This condition is clearly satisfied in the Dewey decimal notation for a tree, so as a consequence of this exercise it is another way to characterize tree structure.)

▸ **15.** [*20*] Invent a notation for binary trees corresponding to the Dewey decimal notation for trees.

16. [*20*] Draw trees analogous to Fig. 22 corresponding to the arithmetic expressions

a) "$2(a - b/c)$";
b) "$a + b + 5c$".

17. [*01*] If T is the tree of Fig. 18, what node is root(T[2, 2])?

18. [*08*] In List (3), what is L[5, 1, 1]? What is L[3, 1]?

▸ **19.** [*15*] Draw a List diagram analogous to (7) for the List $L = (a, (L))$. What is L[2] in this list? What is L[2, 1, 1]?

20. [*M21*] Define a "b-tree" as a tree in which each node has exactly zero or two sons. (Formally, a "b-tree" consists of a single node, called its root, plus 0 or 2 disjoint b-trees.) Show that every b-tree has an odd number of nodes; and give a one-to-one correspondence between binary trees with n nodes and (ordered) b-trees with $2n + 1$ nodes.

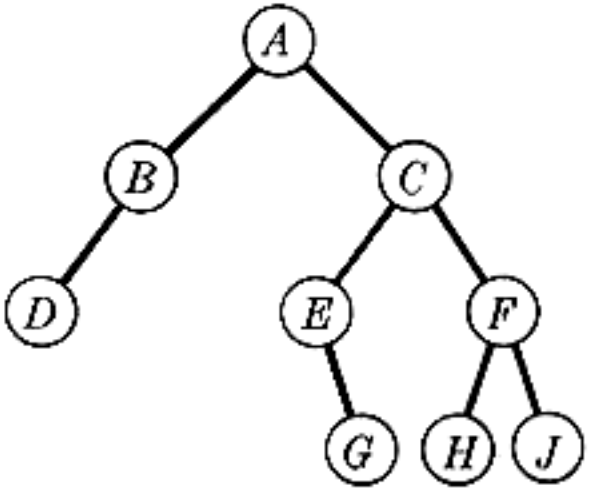
21. [*M22*] If a tree has n_1 nodes of degree 1, n_2 nodes of degree 2, . . . , n_m nodes of degree m, then how many terminal nodes does it have?

22. [*45*] Develop a computer system to display tree structures graphically on a cathode ray tube, with facilities for on-line manipulation of the structures.

2.3.1. Traversing Binary Trees

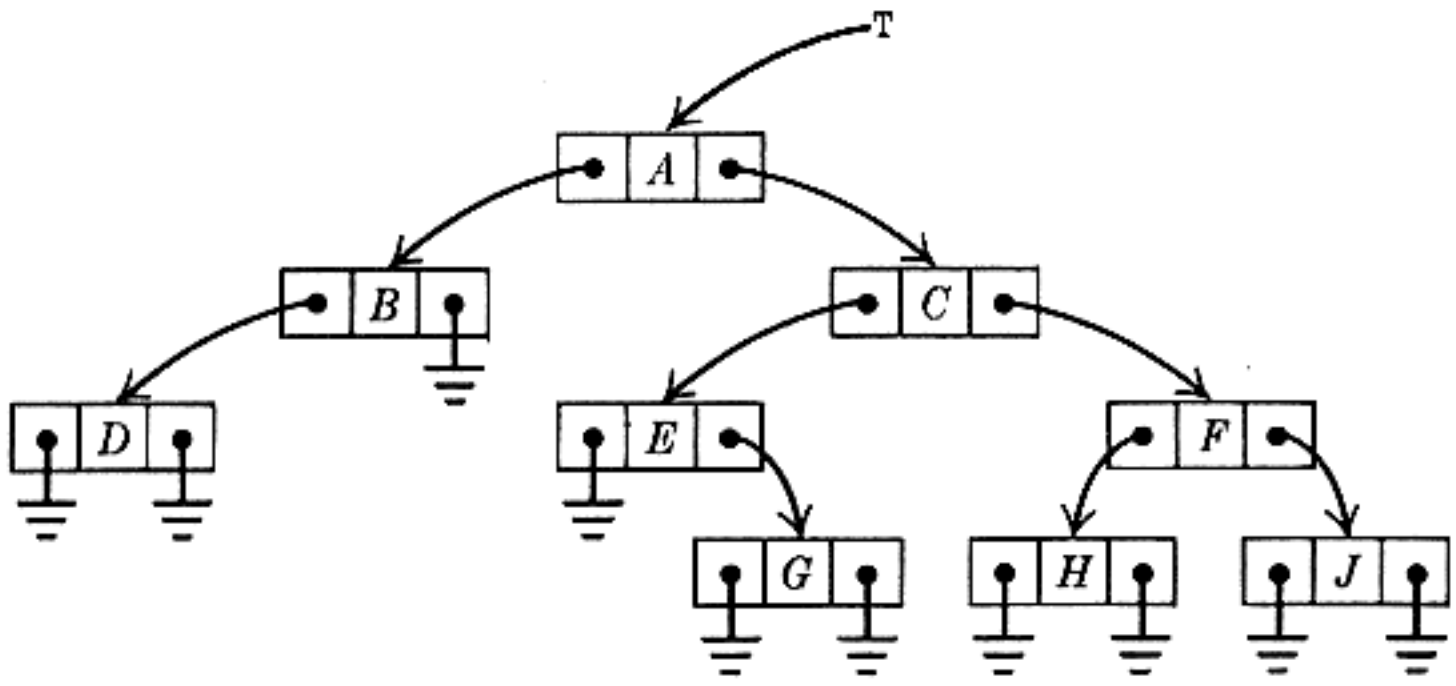
It is important to acquire a good understanding of the properties of binary trees before making further investigations of trees, since general trees are usually represented in terms of some equivalent binary tree inside a computer.

We have defined a binary tree as a finite set of nodes that either is empty, or consists of a root together with two binary trees. This definition suggests a natural way to represent binary trees within a computer: We may have two links, `LLINK` and `RLINK`, within each node, and a link variable `T` which is a "pointer to the tree." If the tree is empty, `T` = Λ; otherwise `T` is the address of the root of the tree, and `LLINK(T)`, `RLINK(T)` are pointers to the left and right, respectively, subtrees of the root. These rules recursively define the memory representation of any binary tree; for example,



(1)

is represented by



(2)

This simple and natural memory representation accounts for the special importance of binary tree structures. Besides the fact that general trees are conveniently representable as binary trees, many trees that arise in applications are themselves inherently binary, so binary trees are of interest in their own right.

There are many algorithms for manipulation of tree structures, and one idea that occurs repeatedly in these algorithms is the notion of *traversing* or "walking through" a tree. This is a method of examining the nodes of the tree systematically so that each node is visited exactly once. A complete traversal of the tree gives us a linear arrangement of the nodes, and many algorithms are facilitated if we can talk about the "next" node following or preceding a given node in such a sequence.

Three principal ways may be used to traverse a binary tree: we can visit the nodes in *preorder*, *postorder*, or *endorder*. These three methods are defined recursively; when the binary tree is empty, it is "traversed" by doing nothing, and otherwise the traversal proceeds in three steps:

Preorder traversal	Postorder traversal
Visit the root	Traverse the left subtree
Traverse the left subtree	Visit the root
Traverse the right subtree	Traverse the right subtree

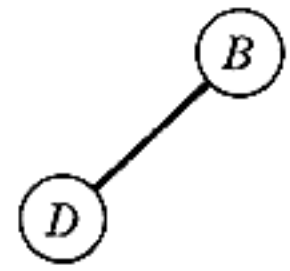
Endorder traversal

Traverse the left subtree
Traverse the right subtree
Visit the root

If we apply these definitions to the binary tree (2), we find that the nodes in preorder are

$$A\ B\ D\ C\ E\ G\ F\ H\ J. \tag{3}$$

(First comes the root A, then comes the left subtree



in preorder, and finally comes the right subtree in preorder.) In postorder we visit the root between visits to the nodes of each subtree, essentially as though the nodes were "projected" down onto a single horizontal line, and this gives the order

$$D\ B\ A\ E\ G\ C\ H\ F\ J. \tag{4}$$

The endorder for the nodes of this binary tree is, similarly,

$$D\ B\ G\ E\ H\ J\ F\ C\ A.$$

We will see that these three ways of arranging the nodes of a binary tree into a sequence are extremely important, as they are intimately connected with most of the computer methods dealing with trees. (*Note:* The names *preorder* and *postorder* come from the fact that each root is visited before or after its left subtree; this terminology is motivated by the special significance given to left subtrees when binary trees are used to represent general trees, as explained in the following section. In many applications of binary trees, there is more symmetry between the meanings of left subtrees and right subtrees, and in such cases the term *symmetric order* is used as a synonym for postorder. Clearly, postorder, which puts the root in the middle, is essentially symmetric between left and right: if the tree is reflected about a vertical axis, the symmetric order is simply reversed.)

A recursively stated definition, such as the one just given for the three basic orders, must be reworked in order to make it directly applicable to computer implementation. General methods for doing this are discussed in Chapter 8; we usually make use of an auxiliary stack, as in the following algorithm:

Algorithm T (*Traverse binary tree in postorder*). Let `T` be a pointer to a binary tree having a representation as in (2); this algorithm visits all the nodes of the binary tree in postorder, making use of an auxiliary stack `A`.

T1. [Initialize.] Set stack `A` empty, and set the link variable `P` $\leftarrow$ `T`.

T2. [`P` $=$ Λ?] If `P` $=$ Λ, go to step T4.

T3. [Stack $\Leftarrow$ `P`.] (Now `P` points to a nonempty binary tree which is to be traversed.) Set `A` $\Leftarrow$ `P`, i.e., push the value of `P` onto stack `A`. (See Section 2.2.1.) Then set `P` $\leftarrow$ `LLINK(P)` and return to step T2.

T4. [`P` $\Leftarrow$ Stack.] If stack `A` is empty, the algorithm terminates; otherwise set `P` $\Leftarrow$ `A`.

T5. [Visit P.] "Visit" NODE(P). (See below.) Then set P ← RLINK(P) and return to step T2. ▮

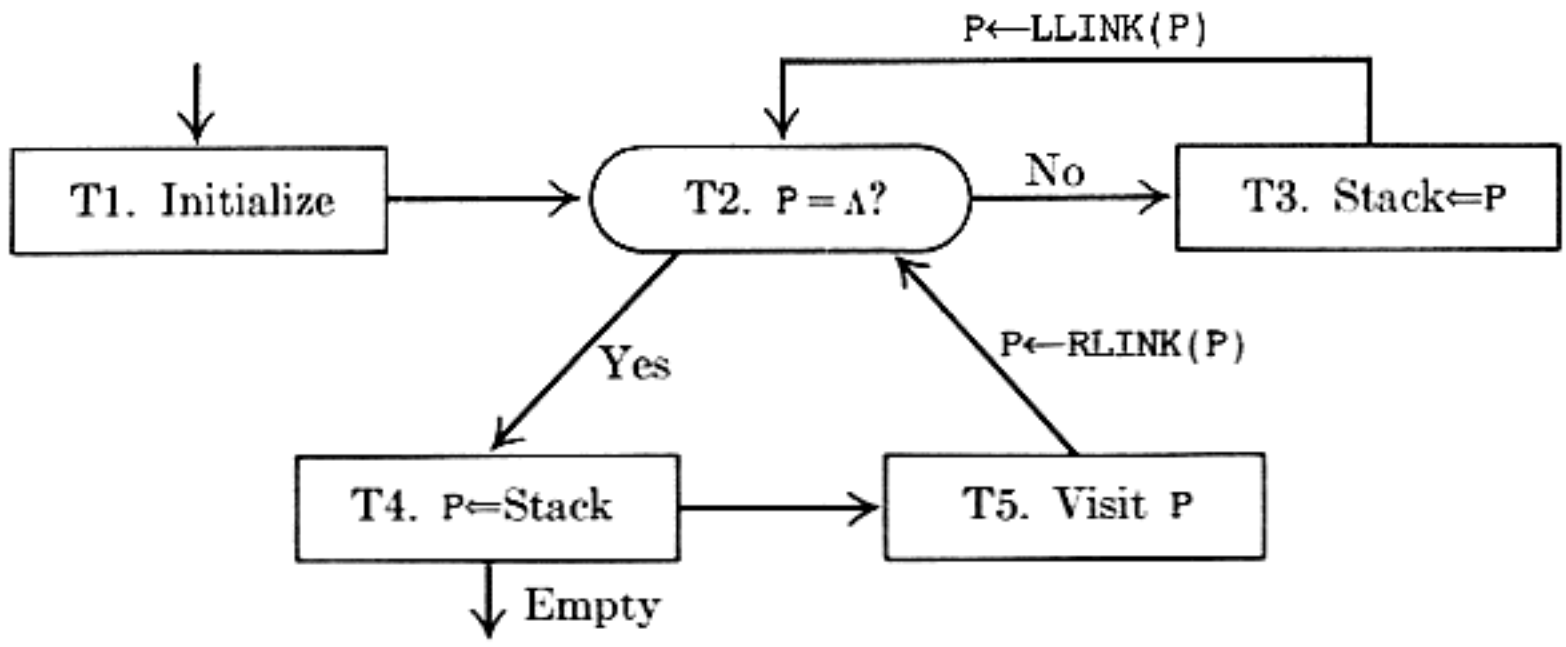


Fig. 23. Flow chart for Algorithm T.

In the final step of this algorithm, the word "visit" means we do whatever activity is intended as the tree is being traversed. Algorithm T runs like a coroutine with respect to this other activity: the main program activates this coroutine whenever it wants P to move from one node to its postorder successor. Of course, since this coroutine calls the main routine in only one place, it is not much different from a subroutine (see Section 1.4.2). Algorithm T assumes that the external activity deletes neither NODE(P) nor any of its ancestors from the tree.

If the reader will now attempt to play through Algorithm T using the tree (2) as a test case, he will easily see the reasons behind the procedure: When we get to step T3, we want to traverse the binary tree whose root is indicated by pointer P. The idea is to save P on a stack and then to traverse the left subtree; when this has been done, we will get to step T4 and will find the old value of P on the stack again. After visiting the root, NODE(P), in step T5, the remaining job is to traverse the right subtree.

Since Algorithm T is typical of so many other algorithms we will see later, it is worth while giving a formal proof of the remarks made in the preceding paragraph. Let us now attempt to *prove* that Algorithm T traverses a binary tree of n nodes in postorder, by using induction on n. Our goal is readily established if we can prove a slightly more general result:

> "Starting at step T2 with P a pointer to a binary tree of n nodes and with the stack A containing A[1] . . . A[m] for some $m \geq 0$, the procedure of steps T2–T5 will traverse the binary tree in question in postorder and will then arrive at step T4 with stack A returned to its original value A[1] . . . A[m]."

This statement is obviously true when $n = 0$, because of step T2. If $n > 0$, let P_0 be the value of P upon entry to step T2. Since $P_0 \neq \Lambda$, we will perform step T3, which means that stack A is changed to A[1] . . . A[m] P_0 and P is set to

LLINK(P_0). Now the left subtree has less than n nodes, so by induction we will traverse the left subtree in postorder and will ultimately arrive at step T4 with A[1] . . . A[m] P_0 on the stack. Step T4 returns the stack to A[1] . . . A[m] and sets P ← P_0. Step T5 now visits NODE(P_0) and sets P ← RLINK(P_0). Now the right subtree has less than n nodes, so by induction we will traverse the right subtree in postorder and arrive at step T4 as required. The tree has been traversed in postorder, by the definition of that order. This completes the proof.

An almost identical algorithm may be formulated which traverses binary trees in preorder (see exercise 12). It is slightly more difficult to achieve the traversal in endorder (see exercise 13) and for this reason endorder is not quite as important as the others.

It is convenient to define a new notation for the successors and predecessors of nodes in these various orders. If P points to a node of a binary tree, let

$$\begin{aligned}
&\texttt{P*} = \text{address of successor of NODE(P) in preorder;}\\
&\texttt{P\$} = \text{address of successor of NODE(P) in postorder;}\\
&\texttt{P\#} = \text{address of successor of NODE(P) in endorder;}\\
&\texttt{*P} = \text{address of predecessor of NODE(P) in preorder;}\\
&\texttt{\$P} = \text{address of predecessor of NODE(P) in postorder;}\\
&\texttt{\#P} = \text{address of predecessor of NODE(P) in endorder.}
\end{aligned} \tag{5}$$

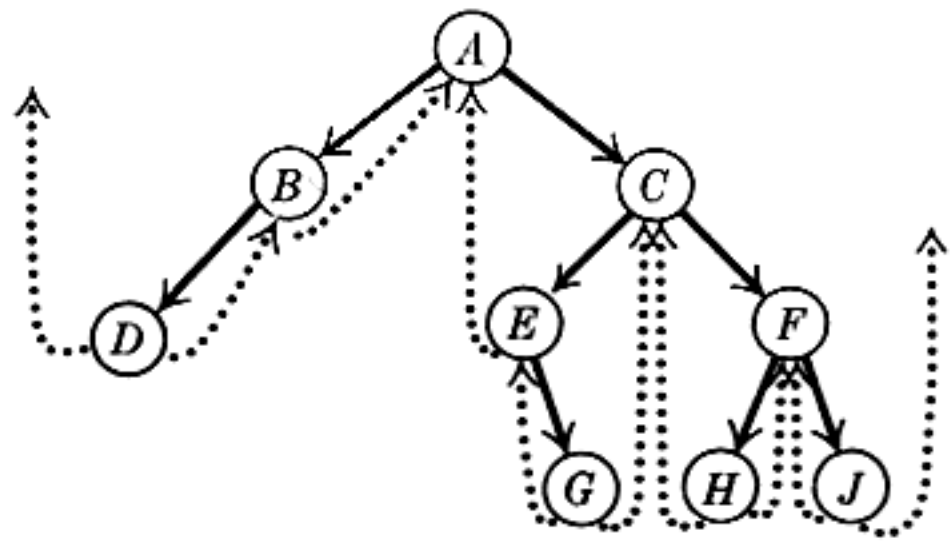
If there is no such successor or predecessor of NODE(P), the value LOC(T) is generally used, where T is a pointer to the tree in question. We have *(P*) = (*P)* = P, $(P$) = ($P)$ = P, etc. As an example of this notation, let INFO(P) be the letter shown in NODE(P) in the tree (2); then if P points to the root, we have INFO(P) = A, INFO(P*) = B, INFO(P$) = E, INFO($P) = B, INFO(#P) = C, and P# = *P = LOC(T).

At this point the reader will perhaps experience a feeling of insecurity about the intuitive meanings of P*, P$, etc. As we proceed further, the ideas will gradually become clearer; exercise 16 at the end of this section may also be of help.

There is an important alternative to the memory representation of binary trees given in (2), which is somewhat analogous to the difference between circular lists and straight one-way lists. Note that there are more null links than other pointers in the tree (2), and indeed this is true of any binary tree represented by that method (see exercise 14). This seems to be wasteful of memory space, and there are various ways to make use of memory more efficiently; for example, we could store two "tag" indicators with each node, which tell in just two bits of memory whether or not the LLINK or RLINK, or both, are null, and the memory space for terminal links can be used for other purposes.

An ingenious use of this extra memory space has been suggested by A. J. Perlis and C. Thornton, who devised the so-called *threaded* tree representation. In this method, terminal links are replaced by "threads" to other parts of the

tree, as an aid to traversing the tree. The threaded tree equivalent to (2) is



(6)

Here dotted lines represent the "threads," which always go to a higher node of the tree. *Every* node now has two links: some nodes, like C, have two ordinary links to left and right subtrees; other nodes, like H, have two thread links, and some nodes have one link of each type. The special threads emanating from D and J will be explained later; they appear in the "leftmost" and "rightmost" nodes.

In the memory representation of a threaded binary tree it is necessary to distinguish between the dotted and solid links; this is done as suggested above by two additional one-bit fields in each node, `LTAG` and `RTAG`. The threaded representation may be precisely defined as follows:

Unthreaded representation	Threaded representation
`LLINK(P)` $= \Lambda$	`LTAG(P)` $=$ "$-$", `LLINK(P)` $=$ `$P`
`LLINK(P)` $= $ `Q` $\neq \Lambda$	`LTAG(P)` $=$ "$+$", `LLINK(P)` $=$ `Q`
`RLINK(P)` $= \Lambda$	`RTAG(P)` $=$ "$-$", `RLINK(P)` $=$ `P$`
`RLINK(P)` $=$ `Q`	`RTAG(P)` $=$ "$+$", `RLINK(P)` $=$ `Q`

In some algorithms it can be guaranteed that the root of any subtree always will appear in a lower memory location than the other nodes of the subtree. Thus `LTAG(P)` will be "$-$" if and only if `LLINK(P)` $<$ `P`, so that `LTAG` (and similarly `RTAG`) is redundant.

According to the definition above, each new thread link points directly to the predecessor or successor of the node in question, in postorder (symmetric order). Figure 24 illustrates the general orientation of thread links in any binary tree.

The great advantage of threaded trees is that the traversal algorithms become simpler. For example, the following algorithm calculates `P$`, given `P`:

Algorithm S (*Symmetric (postorder) successor in a threaded binary tree*). If `P` points to a node of a threaded binary tree, this algorithm sets `Q` $\leftarrow$ `P$`.

S1. [`RLINK(P)` a thread?] Set `Q` $\leftarrow$ `RLINK(P)`. If `RTAG(P)` $=$ "$-$", terminate the algorithm.

S2. [Search to left.] If `LTAG(Q)` $=$ "$+$", set `Q` $\leftarrow$ `LLINK(Q)` and repeat this step. Otherwise the algorithm terminates. ▮

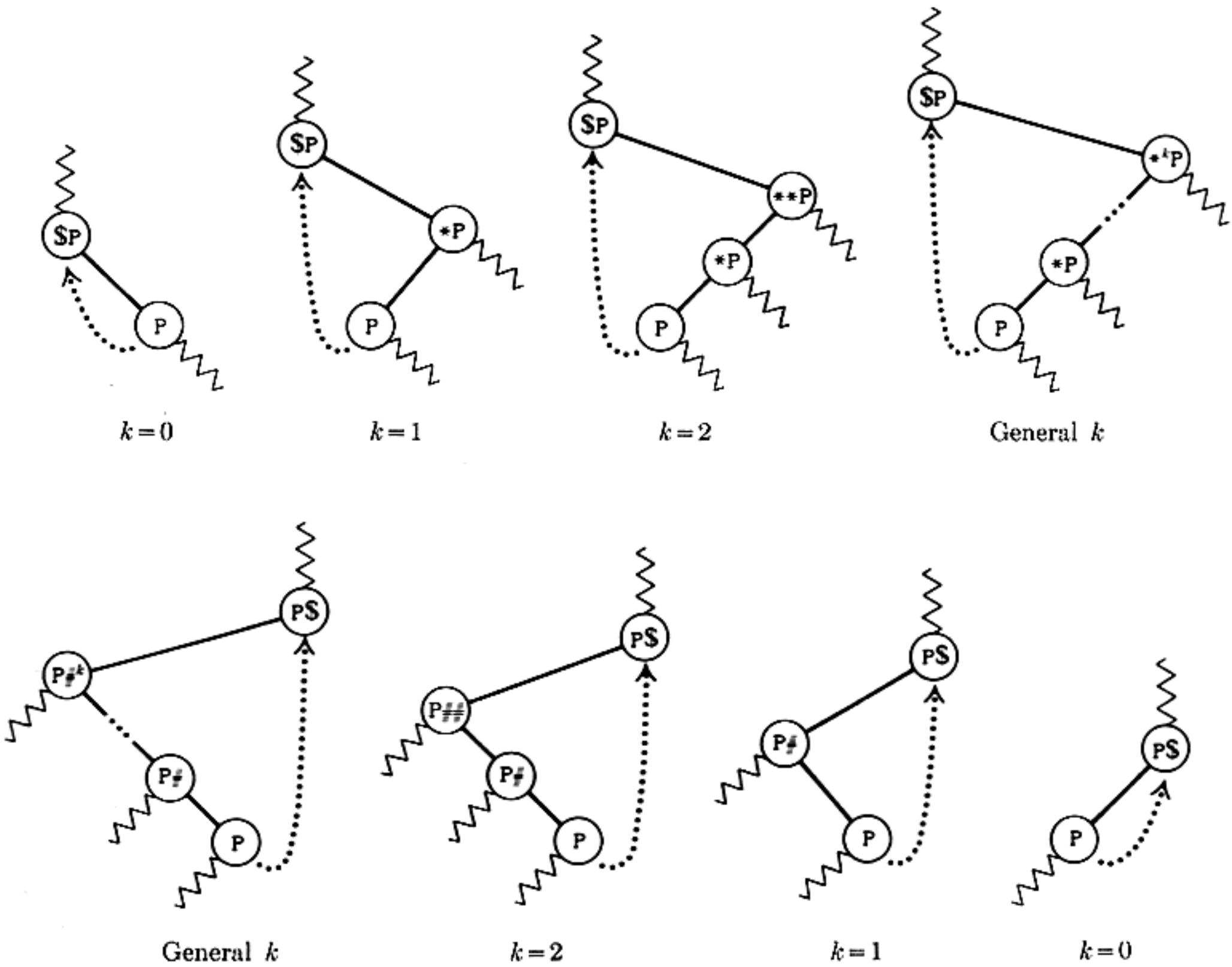


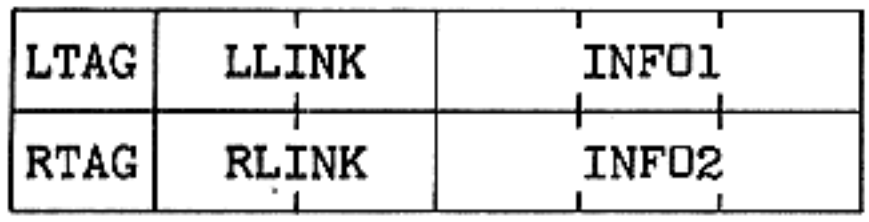
Fig. 24. General orientation of left and right thread links in a binary tree. "∿∿" lines indicate links or threads to other parts of the tree.

Note that no stack is needed here to accomplish what was done using a stack in Algorithm T. In fact, the ordinary representation (2) makes it impossible to find P$ efficiently, given only the address of a random point P in the tree; since no links point upward in the unthreaded representation, there is no clue to what nodes are above a given node, unless we retain a history of how we reached that point (and that is essentially the stack in Algorithm T).

We claim that Algorithm S *is* "efficient," although this property is not immediately obvious, since step S2 can be executed any number of times. In view of the loop in step S2, would it perhaps be faster to use a stack after all, as Algorithm T does? To investigate this question, we will consider the average number of times step S2 must be performed if P is a "random" point in the tree; or what is the same, we will determine the total number of times step S2 is performed if Algorithm S is used repeatedly to traverse an entire tree.

At the same time as this analysis is being carried out, it will be instructive to study MIX programs for both Algorithms S and T. The following programs

assume that the nodes have the two-word form



In an unthreaded tree, LTAG and RTAG will always be "+" and terminal links will be represented by zero. The abbreviations LLINKT and RLINKT will be used to stand for the combined LTAG–LLINK and RTAG–RLINK fields, respectively.

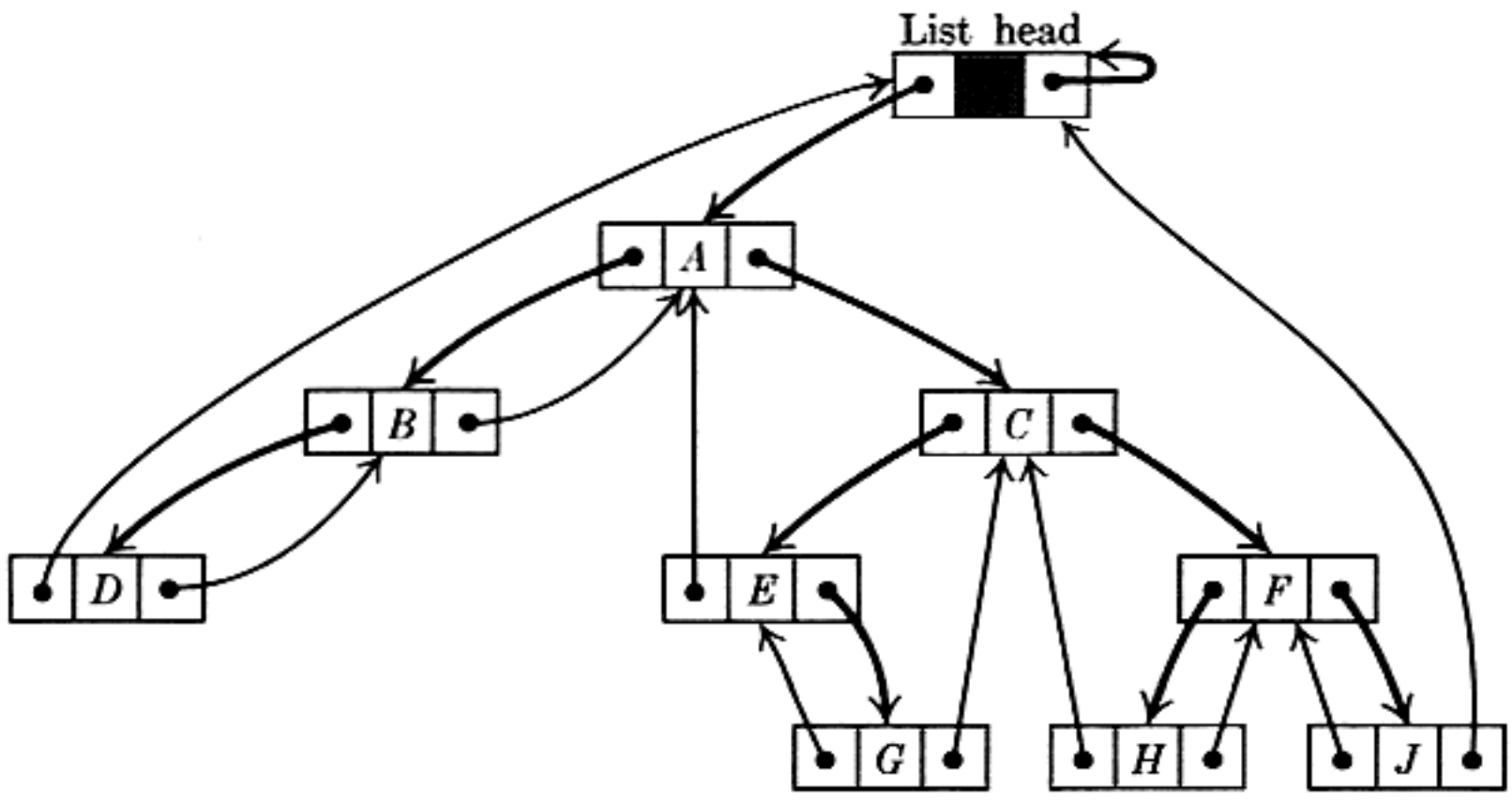
As usual, we should be careful to set up all of our algorithms so that they work properly with empty trees; and if T is the pointer to the tree, we would like to have LOC(T)* and LOC(T)$ be the *first* nodes in prefix or symmetric order, respectively. For threaded trees, it turns out that things will work nicely if NODE(LOC(T)) is made into a "list head" for the tree, with

$$\texttt{LLINKT(HEAD)} \equiv \texttt{T}, \quad \texttt{RLINK(HEAD)} = \texttt{HEAD}, \quad \texttt{RTAG(HEAD)} = \text{"+"}. \tag{7}$$

(HEAD denotes LOC(T), the address of the list head.) An empty threaded tree will satisfy the conditions

$$\texttt{LLINK(HEAD)} = \texttt{HEAD}, \quad \texttt{LTAG(HEAD)} = \text{"}-\text{"}. \tag{8}$$

The tree grows by having nodes inserted to the *left* of the list head. (These initial conditions are primarily dictated by the algorithm to compute P*, which appears in exercise 17.) In accordance with these conventions, the computer representation for tree (1), as a threaded tree, is



(9)

With these preliminaries out of the way, we are now ready to consider MIX programs for Algorithms S and T. The following two programs traverse a binary tree in symmetric order (i.e., postorder), jumping to location VISIT periodically with index register 5 pointing to the node that is currently of interest.

Program T. In this implementation of Algorithm T, the stack is kept in locations A + 1, A + 2, . . . , A + MAX; and OVERFLOW occurs if the stack grows too large. rI6 is the stack pointer and rI5 ≡ P. The program has been rearranged slightly from Algorithm T (step T2 appears twice) so that the test for an empty stack need not be made when going directly from T3 to T2 to T4.

```
01    LLINK   EQU   1:2
02    RLINK   EQU   1:2
03    T1      LD5   HEAD(LLINK)      1    T1. Initialize. Set P ← T.
04            J5Z   DONE             1    Stop if P = Λ.
05            ENT6  0                1
06    T3      DEC6  MAX              n    T3. Stack ⇐ P.
07            J6NN  OVERFLOW         n    Has stack reached capacity?
08            INC6  MAX+1            n    If not, increase stack pointer.
09            ST5   A,6              n    Store P in stack.
10            LD5   0,5(LLINK)       n    P ← LLINK(P).
11    T2A     J5NZ  T3               n    To T3 if P ≠ Λ.
12    T4      LD5   A,6              n    T4. P ⇐ Stack.
13            DEC6  1                n    Decrease stack pointer.
14    T5      JMP   VISIT            n    T5. Visit P.
15            LD5   1,5(RLINK)       n    P ← RLINK(P).
16    T2      J5NZ  T3               n    T2. P = Λ?
17            J6NZ  T4               a    Test if stack is empty.
18    DONE    ...                         ∎
```

Program S. Algorithm S has been augmented with initialization and termination conditions to make this program comparable to Program T.

```
01    LLINKT  EQU   0:2
02    RLINKT  EQU   0:2
03    S0      ENT6  HEAD             1        S0. Initialize. Set Q ← HEAD.
04            JMP   S2               1
05    S3      JMP   VISIT            n        S3. Visit P.
06    S1      LD5N  1,5(RLINKT)      n        S1. RLINK(P) a thread?
07            J5NN  1F               n        Jump if RTAG(P) = "—".
08            ENN6  0,5              n − a    Otherwise set Q ← RLINK(P).
09    S2      ENT5  0,6              n + 1    Set P ← Q.
10            LD6   0,5(LLINKT)      n + 1    Q ← LLINKT(P).
11            J6P   *-2              n + 1    If LTAG(P) = "+", repeat.
12    1H      ENT6  -HEAD,5          n + 1
13            J6NZ  S3               n + 1    Visit unless P = HEAD. ∎
```

An analysis of the running time appears with the above code. These quantities are easy to determine using Kirchhoff's law and the facts that

i) In Program T, the number of insertions onto the stack must equal the number of deletions;

ii) In Program S, the LLINK and RLINK of each node is examined precisely once;
iii) The number of "visits" is the number of nodes in the tree.

The analysis tells us Program T takes $15n + a + 3$ units of time, and Program S takes $11n - a + 8$ units, where n is the number of nodes in the tree, and a is the number of terminal right links (nodes with no right subtree). The quantity a can be as low as 1, assuming that $n \neq 0$, and it can be as high as n; and if left and right are symmetrical, the average value of a is $(n + 1)/2$ as a consequence of facts proved in exercise 14.

The principal conclusions we may reach on the basis of this analysis are that

i) Step S2 of Algorithm S is performed only *once* on the average per execution of that algorithm, if P is a random node of the tree.
ii) Traversal is slightly faster for threaded trees; in the case of MIX the ratio of execution times is about 3 to 2.
iii) Algorithm T needs more memory space than Algorithm S because of the auxiliary stack required. In Program T we kept the stack in consecutive memory locations, and, consequently, it was necessary to put an arbitrary bound on its size. It would be very embarrassing if this bound were exceeded, so it must be set reasonably large (see exercise 10); thus the memory requirement of Program T is significantly more than Program S. Not infrequently a complex computer application will be independently traversing several trees at once, and a separate stack will be needed for each tree under Program T. This suggests that Program T might use linked allocation for its stack (see exercise 20); its execution time then becomes $30n + a + 3$ units, roughly twice as slow as before, although this may not be terribly important when the execution time for the other coroutine is added in. Still another alternative is to keep the stack links within the tree itself in a very tricky way, as discussed in exercise 21.
iv) Algorithm S is, of course, more general than Algorithm T, since it allows us to go from P to P$ when we are not necessarily traversing the entire binary tree.

So a threaded binary tree is decidedly superior to an unthreaded one, with respect to traversal. These advantages are offset in some applications by the slightly increased time needed to insert and delete nodes in a threaded tree. It is also sometimes possible to save memory space by "sharing" common subtrees with an unthreaded representation, while threaded trees require adherence to a strict tree structure with no overlapping of subtrees.

Thread links can also be used to compute P*, $P, and #P with efficiency comparable to that of Algorithm S. The functions *P and P# are slightly harder to compute, just as they are for unthreaded tree representations. The reader is urged to work exercise 17.

Most of the usefulness of threaded trees would disappear if it were hard to set up the thread links in the first place. The fact which makes the idea really work is that threaded trees grow almost as easily as ordinary ones do. We have the following algorithm:

Algorithm I (*Insertion into a threaded binary tree*). This algorithm attaches a single node, NODE(Q), as the right subtree of NODE(P), if the right subtree is empty (i.e., if RTAG(P) = "−"), and it inserts NODE(Q) between NODE(P) and NODE(RLINK(P)) otherwise. The binary tree in which the insertion takes place is assumed to be threaded as in (9); for a modification, see exercise 23.

I1. [Adjust tags and links.] Set RLINK(Q) ← RLINK(P), RTAG(Q) ← RTAG(P), RLINK(P) ← Q, RTAG(P) ← "+", LLINK(Q) ← P, LTAG(Q) ← "−".

I2. [Was RLINK(P) a thread?] If RTAG(Q) = "+", set LLINK(Q$) ← Q. (Here Q$ is determined by Algorithm S, which will work properly even though LLINK(Q$) now points to NODE(P) instead of NODE(Q). This step is necessary only when inserting into the midst of a threaded tree instead of merely inserting a "new leaf.") ∎

By reversing the roles of left and right (in particular, by replacing Q$ by $Q in step I2), we obtain an algorithm which inserts to the left in a similar way.

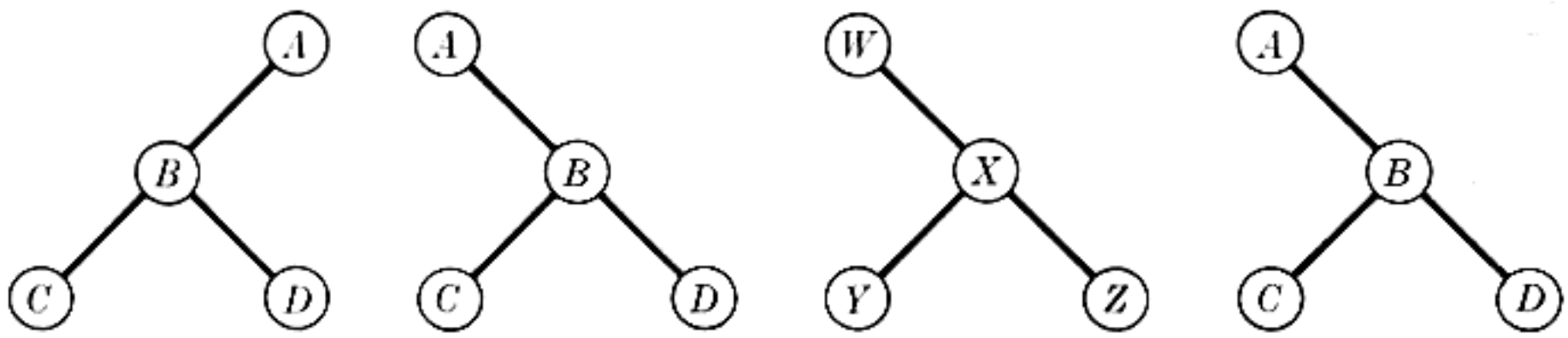
Our discussion of threaded binary trees so far has made use of thread links both to the left and to the right. There is an important middle ground between the completely unthreaded and completely threaded methods of representation: A *right-threaded binary tree* combines the two approaches by making use of threaded RLINKs, but representing empty left subtrees by LLINK = Λ. (Similarly, a left-threaded binary tree threads only the null LLINKs.) Algorithm S does not make essential use of threaded LLINKs; if we change the test "LTAG = +" in step S2 to "LLINK ≠ Λ", we obtain an algorithm for traversing right-threaded binary trees in post-order. Program S works without change in the right-threaded case. A great many applications of binary tree structures require only a left-to-right traversal of trees using the functions P$ and/or P*, and for these applications there is no need to thread the LLINKs. We have described threading in both the left and right directions in order to indicate the symmetry and possibilities of the situation, but in practice one-sided threading is much more common.

Let us now consider an important concept about binary trees, and its connection to traversal. Two binary trees T and T' are said to be *similar* if they have the same structure; formally, this means either they are both empty, or they are both nonempty and their left and right subtrees are respectively similar. Similarity means, informally, that the diagrams of T and T' have the same "shape." Another way to phrase similarity is to say there is a one-to-one correspondence between the nodes of T and T' which preserves the structure: if nodes u_1 and u_2 in T correspond respectively to u_1' and u_2' in T', then u_1 is

in the left subtree of u_2 if and only if u_1' is in the left subtree of u_2', and the same is true for right subtrees.

The binary trees T and T' are said to be *equivalent* if they are similar and, moreover, if corresponding nodes contain the same information. Formally, let info (u) denote the information contained in a node u; the trees are equivalent if and only if they are both empty, or they are both nonempty and info $(\text{root}(T))$ = info $(\text{root}(T'))$ and their left and right subtrees are respectively equivalent.

As examples of these definitions, consider the four binary trees



in which the first two are not similar (although they are equivalent as *trees* instead of as binary trees); the second, third, and fourth are similar and, in fact, the second and fourth are equivalent.

Computer applications involving tree structures often require an algorithm to decide whether two binary trees are similar or equivalent. The following theorem is useful in this regard:

Theorem A. *Let the nodes of binary trees T and T' be respectively*

$$u_1, u_2, \ldots, u_n \qquad \text{and} \qquad u_1', u_2', \ldots, u_n',$$

in preorder. For any node u let

$$\begin{array}{llll} l(u) = 1 & \text{if } u \text{ has a nonempty left subtree,} & l(u) = 0 & \text{otherwise;} \\ r(u) = 1 & \text{if } u \text{ has a nonempty right subtree,} & r(u) = 0 & \text{otherwise.} \end{array} \tag{10}$$

Then T and T' are similar if and only if $n = n'$ and

$$l(u_j) = l(u_j'), \qquad r(u_j) = r(u_j') \qquad \text{for} \qquad 1 \le j \le n. \tag{11}$$

T and T' are equivalent if and only if in addition

$$\text{info } (u_j) = \text{info } (u_j') \qquad \text{for} \qquad 1 \le j \le n. \tag{12}$$

Note that l and r are essentially the contents of the LTAG and RTAG in a threaded tree, with 1 and 0 substituted for "+" and "−". This theorem characterizes any binary tree structure in terms of two sequences of 0's and 1's.

Proof. It is clear that the condition for equivalence of binary trees will follow immediately if we prove the condition for similarity; furthermore $n = n'$ and

(11) is certainly necessary, since corresponding nodes of similar trees must have the same position in preorder. Therefore it suffices to prove that the conditions (11) and $n = n'$ will guarantee the similarity of T and T'. The proof is by induction on n, using the following auxiliary result:

Lemma P. *Let the nodes of a nonempty binary tree be $u_1, u_2, \ldots, u_n$ in preorder, and let $f(u) = l(u) + r(u) - 1$. Then*

$$\begin{aligned} f(u_1) + f(u_2) + \cdots + f(u_n) &= -1, \quad \text{and} \\ f(u_1) + \cdots + f(u_k) &\geq 0 \quad \text{for} \quad 1 \leq k < n. \end{aligned} \tag{13}$$

Proof. The result is clear for $n = 1$. If $n > 1$, the binary tree consists of its root u_1 and further nodes. If $f(u_1) = 0$, then either the left subtree or the right subtree is empty, so the condition is obviously true by induction. If $f(u_1) = 1$, let the left subtree have n_l nodes; by induction we have

$$\begin{aligned} f(u_1) + \cdots + f(u_k) &> 0 \quad \text{for} \quad 1 \leq k \leq n_l, \\ f(u_1) + \cdots + f(u_{n_l+1}) &= 0, \end{aligned} \tag{14}$$

and the condition (13) is again evident. ∎

(For other theorems analogous to Lemma P, see the discussion of "Polish notation" in Chapter 11.)

To complete the proof of Theorem A, note that it is clearly true when $n = 0$. If $n > 0$, the definition of preorder implies that u_1 and u'_1 are the respective roots of their trees, and there are integers n_l, n'_l (the sizes of the left subtrees) such that

$u_2, \ldots, u_{n_l+1}$ and $u'_2, \ldots, u'_{n'_l+1}$ are the left subtrees of T and T';
$u_{n_l+2}, \ldots, u_n$ and $u'_{n'_l+2}, \ldots, u'_n$ are the right subtrees of T and T'.

The proof by induction will be complete if we can show $n_l = n'_l$. There are three cases:

if $l(u_1) = 0$, then $n_l = 0 = n'_l$;
if $l(u_1) = 1$, $r(u_1) = 0$, then $n_l = n - 1 = n'_l$;
if $l(u_1) = r(u_1) = 1$, then by Lemma P we can find the least $k > 0$ such that $f(u_1) + \cdots + f(u_k) = 0$; and $n_l = k - 1 = n'_l$ (cf. Eqs. 14). ∎

As a consequence of Theorem A, we can test two threaded binary trees for equivalence or similarity by simply traversing them in preorder and checking the `INFO` and `TAG` fields.

We conclude this section by giving a typical, yet basic, algorithm for binary trees, one that makes a copy of a binary tree into different memory locations:

Algorithm C (*Copy a binary tree*). Let `HEAD` be the address of the list head of a binary tree T (i.e., T is the left subtree of `HEAD`; `LLINK(HEAD)` is a pointer to the tree). Let `NODE(U)` be a node with an empty left subtree. This algorithm makes a copy of T and the copy becomes the left subtree of `NODE(U)`. In partic-

ular, if NODE(U) is the list head of an empty binary tree, this algorithm changes the empty tree into a copy of T.

C1. [Initialize.] Set P ← HEAD, Q ← U. Go to C4.

C2. [Anything to right?] If NODE(P) has a nonempty right subtree, set R ⇐ AVAIL, and attach NODE(R) to the right of NODE(Q). (At the beginning of step C2, the right subtree of NODE(Q) is empty.)

C3. [Copy INFO.] Set INFO(Q) ← INFO(P). (Here INFO denotes all parts of the node that are to be copied.)

C4. [Anything to left?] If NODE(P) has a nonempty left subtree, set R ⇐ AVAIL, and attach NODE(R) to the left of NODE(Q). (At the beginning of step C4, the left subtree of NODE(Q) is empty.)

C5. [Advance.] Set P ← P*, Q ← Q*.

C6. [Test if complete.] If P = HEAD (or equivalently if Q = RLINK(U), assuming NODE(U) has a nonempty right subtree), the algorithm terminates; otherwise go to step C2. ∎

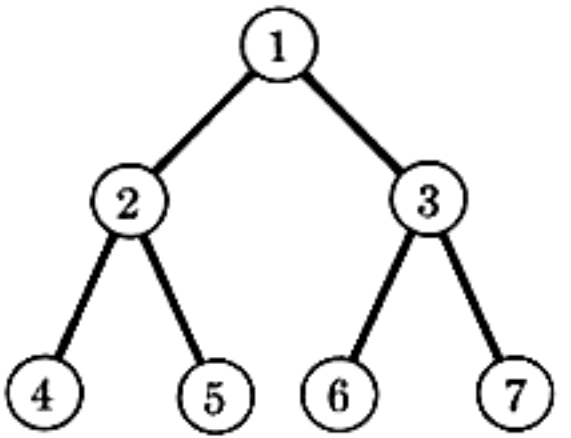
This simple algorithm shows a typical application of tree traversal; the description here applies to threaded, unthreaded, or partially threaded trees. Step C5 requires the calculation of preorder successors P* and Q*; for unthreaded trees, this generally is done with an auxiliary stack. A proof of the validity of Algorithm C appears in exercise 29; a MIX program corresponding to this algorithm in the case of a right-threaded binary tree appears in exercise 2.3.2–13. For threaded trees, the "attaching" in steps C2 and C4 is done using Algorithm I.

The exercises which follow include quite a few topics of interest relating to the material of this section.

EXERCISES

1. [*01*] In the binary tree (2), let `INFO(P)` denote the letter stored in `NODE(P)`. What is `INFO(LLINK(RLINK(RLINK(T))))`?

2. [*11*] List the nodes of the following binary tree in (a) preorder; (b) symmetric order; (c) endorder:



3. [*20*] Is the following statement true or false? "The terminal nodes of a binary tree occur in the same relative position in preorder, postorder, and endorder."

▸ **4.** [*20*] The text defines three basic orders for traversing a binary tree; another alternative would be to proceed in three steps as follows:

a) Visit the root,
b) traverse the right subtree,
c) traverse the left subtree,

using the same rule recursively on all nonempty subtrees. Does this new order bear any simple relation to the three orders already discussed?

5. [*22*] Nodes of a binary tree may be identified by a sequence of zeros and ones, in a notation analogous to "Dewey decimal notation" for trees, as follows: The root (if present) is represented by the sequence "1". Roots (if present) of the left and right subtrees of the node represented by α are respectively represented by $\alpha 0$ and $\alpha 1$. For example, the node H in (1) would have the representation "1110". (Cf. exercise 2.3–15.)

Show that preorder, postorder, and endorder may be conveniently described in terms of this notation.

6. [*M22*] Suppose that a binary tree has n nodes which are $u_1\ u_2 \ldots u_n$ in preorder and $u_{p_1}\ u_{p_2} \ldots u_{p_n}$ in postorder. Show that the permutation $p_1\ p_2 \ldots p_n$ can be obtained by passing $1\ 2 \ldots n$ through a stack, in the sense of exercise 2.2.1–2. Conversely, show that any permutation $p_1\ p_2 \ldots p_n$ obtainable with a stack corresponds to some binary tree in this way.

7. [*22*] Show that if we are given the preorder and the postorder of the nodes of a binary tree, the binary tree structure may be constructed. Does the same result hold true if we are given the preorder and endorder (instead of postorder)? Or if we are given the postorder and endorder?

8. [*20*] Find all binary trees whose nodes appear in exactly the same sequence in both (a) preorder and postorder; (b) preorder and endorder; (c) postorder and endorder.

9. [*M20*] When a binary tree having n nodes is traversed using Algorithm T, state how many times each of the steps T1, T2, T3, T4, and T5 is performed (as a function of n).

▸ **10.** [*20*] What is the largest number of entries that can be in the stack at once, during the execution of Algorithm T, if the binary tree has n nodes? (The answer to this question is very important for storage allocation, if the stack is being stored consecutively.)

11. [*M48*] Analyze the *expected* value of the largest stack size occurring during the execution of Algorithm T as a function of n, given that all binary trees with n nodes are considered equally probable.

12. [*22*] Design an algorithm analogous to Algorithm T which traverses a binary tree in *preorder*, and prove that your algorithm is correct.

▸ **13.** [*24*] Design an algorithm analogous to Algorithm T which traverses a binary tree in *endorder*.

14. [*22*] Show that if a binary tree with n nodes is represented as in (2), the total number of Λ links in the representation can be expressed as a simple function of n; this quantity does not depend on the shape of the tree.

15. [*19*] Note that in a threaded-tree representation like (9), each node except the list head has exactly one link pointing to it from above, namely the link from its "father."

In addition, some of the nodes have further links pointing to them from below; for example, the node containing "*C*" has two pointers coming up from below, node "*E*" has one. Is there any simple connection between the number of links pointing to a node and some other basic property of that node? (It is, of course, necessary to know how many links point to a given node when alterations to the tree structure are being considered.)

▸ **16.** [*22*] The diagrams in Fig. 24 help to give an intuitive characterization of the position of NODE(Q$) in a binary tree, in terms of the structure near NODE(Q): If NODE(Q) has a nonempty right subtree, consider Q = P, Q = P in the upper diagrams; NODE(Q$) is the "leftmost" node of that right subtree. If NODE(Q) has an empty right subtree, consider Q = P in the lower diagrams; NODE(Q$) is located by proceeding upward in the tree until after the first upward step to the right.

Give a similar "intuitive" rule for finding the position of NODE(Q*) in a binary tree in terms of the structure near NODE(Q).

▸ **17.** [*22*] Give an algorithm analogous to Algorithm S for determining P* in a threaded binary tree. Assume that the tree has a list head as in (7), (8), (9).

18. [*24*] Many algorithms dealing with trees like to visit each node *twice* instead of once, using a combination of preorder and postorder called the *dual order*. Traversal of a binary tree in dual order is defined as follows: If the binary tree is empty, do nothing; otherwise

a) visit the root, for the first time;
b) traverse the left subtree, in dual order;
c) visit the root, for the second time;
d) traverse the right subtree, in dual order.

For example, traversal of (1) in dual order gives the sequence

$$A_1B_1D_1D_2B_2A_2C_1E_1E_2G_1G_2C_2F_1H_1H_2F_2J_1J_2$$

where A_1 means "A" is being visited for the first time, etc.

If P points to a node of the tree and if $d = 1$ or 2, define $(\mathtt{P}, d)^{\Delta} = (\mathtt{Q}, e)$ if either the next step in dual order after visiting NODE(P) the dth time is to visit NODE(Q) the eth time, or if (P, d) was the last step in dual order and (Q, e) = (HEAD, 2), where HEAD is the address of the list head. We also define $(\mathtt{HEAD}, 1)^{\Delta}$ as the first step in dual order.

Design an algorithm analogous to Algorithm T which traverses a binary tree in dual order, and also design an algorithm analogous to Algorithm S which computes $(\mathtt{P}, d)^{\Delta}$. Discuss the relation between these algorithms and exercises 12 and 17.

19. [*24*] Design an algorithm analogous to Algorithm S for the calculation of P# in a threaded binary tree.

20. [*23*] Modify Program T so that it keeps the stack in a linked list, not in consecutive memory locations.

21. [*32*] Design an algorithm which traverses an unthreaded binary tree in postorder *without using any auxiliary stack*. It is permissible to alter the LLINK and RLINK fields of the tree nodes during this algorithm in any manner whatever, subject only to the condition that the binary tree has its conventional representation [as in (2), for

example] both before and after your algorithm has traversed the tree. You may also use an RTAG field (one bit only) in each node for temporary storage.

22. [*25*] Write a MIX program for the algorithm given in exercise 21 and compare its execution time to Programs S and T.

23. [*22*] Design algorithms analogous to Algorithm I for insertion to the right and insertion to the left in a *right-threaded* binary tree. Assume that the nodes have the fields LLINK, RLINK, and RTAG.

24. [*M20*] Is Theorem A still valid if the nodes of T and T' are given in postorder instead of preorder?

25. [*M24*] Let $\mathfrak{I}$ be a set of binary trees and let $N(\mathfrak{I})$ be the set $\{\text{info}\,(u) \mid u \text{ is a node of } T \text{ for some } T \text{ in } \mathfrak{I}\}$. Suppose that a linear ordering relation "$\preceq$" (cf. exercise 2.2.3–14) has been defined on $N(\mathfrak{I})$. Given any trees T, T' in $\mathfrak{I}$, let us now define $T \preceq T'$ if and only if

1) T is empty; or
2) T and T' are not empty, and $\text{info}\,(\text{root}\,(T)) \prec \text{info}\,(\text{root}\,(T'))$; or
3) T and T' are not empty, $\text{info}\,(\text{root}\,(T)) = \text{info}\,(\text{root}\,(T'))$, leftsubtree $(T) \preceq$ leftsubtree (T'), and leftsubtree (T) is not equivalent to leftsubtree (T'); or
4) T and T' are not empty, $\text{info}\,(\text{root}\,(T)) = \text{info}\,(\text{root}\,(T'))$, leftsubtree (T) is equivalent to leftsubtree (T'), and rightsubtree $(T) \preceq$ rightsubtree (T').

Prove that (a) $T \preceq T'$ and $T' \preceq T''$ implies $T \preceq T''$; (b) T is equivalent to T' if and only if $T \preceq T'$ and $T' \preceq T$; (c) for any T, T' in $\mathfrak{I}$ we have either $T \preceq T'$ or $T' \preceq T$. [Thus, if equivalent trees in $\mathfrak{I}$ are regarded as equal, the relation $\preceq$ induces a linear ordering on $\mathfrak{I}$. This ordering has many applications (for example, in the simplification of algebraic expressions). When $N(\mathfrak{I})$ has only one element, i.e., if the "info" of each node is the same, we have the special case that equivalence is the same as similarity.]

26. [*M24*] Consider the ordering $T \preceq T'$ defined in the preceding exercise. Prove a theorem analogous to Theorem A, giving a necessary and sufficient condition that $T \preceq T'$, and making use of the dual order as defined in exercise 18.

▸ **27.** [*28*] Design an algorithm which tests two given trees T and T' to see whether $T \prec T'$, $T \succ T'$, or T equivalent to T', in terms of the relation defined in exercise 25, assuming that both binary trees are right-threaded. Assume that each node has the fields LLINK, RLINK, RTAG, INFO; use no auxiliary stack.

28. [*00*] After Algorithm C has been used to make a copy of a tree, is the new binary tree *equivalent* to the original, or *similar* to it?

29. [*M25*] Prove as rigorously as possible that Algorithm C is valid.

▸ **30.** [*22*] Design an algorithm that threads an unthreaded tree [e.g., transforms (2) into (9)]. *Note:* Always use the notation P*, P$, etc. when possible instead of repeating the steps for traversal algorithms like Algorithm T.

31. [*23*] Design an algorithm which "erases" a right-threaded binary tree, i.e., returns all of its nodes except the list head to the AVAIL list and makes the list head signify an empty binary tree. Assume that each node has the fields LLINK, RLINK, RTAG; use no auxiliary stack.

32. [*21*] Suppose that each node of a binary tree has four link fields: LLINK and RLINK, which point to left and right subtrees or Λ, as in an unthreaded tree; and SUC and PRED, which point to the successor and predecessor of the node in symmetric order. (Thus, SUC(P) = P$ and PRED(P) = $P. Such a tree contains more information than a threaded tree.) Design an algorithm like Algorithm I for insertion into such a tree.

▶ **33.** [*30*] There is more than one way to thread a tree! Consider the following representation, using three fields LTAG, LLINK, RLINK in each node:

LTAG(P): defined the same as in a threaded binary tree;
LLINK(P): always equal to P*;
RLINK(P): defined the same as in an unthreaded binary tree.

Discuss insertion algorithms for such a representation, and write out the copying algorithm, Algorithm C, in detail for this representation.

34. [*22*] Let P point to a node in some binary tree, and let HEAD be the address of the list head of an empty binary tree. Give an algorithm which removes NODE(P) and all of its subtrees from whatever tree it was in, and which attaches the subtree having NODE(P) as its root to HEAD. Assume that all the binary trees in question are *right-threaded*, with fields LLINK, RTAG, RLINK in each node.

35. [*40*] Define a *ternary tree* (and, more generally, a t-ary tree for any $t \geq 2$) in a manner analogous to our definition of a binary tree, and explore the topics discussed in this section (including topics found in the exercises above) which can be generalized to t-ary trees in a meaningful way.

36. [*M23*] Exercise 1.2.1–15 shows that lexicographic order extends a well-ordering of a set S to a well-ordering of the n-tuples of elements of S. Exercise 25 above shows that a linear ordering of the information in tree nodes can be extended to a linear ordering of trees, using a similar definition. If the relation $\prec$ well-orders $N(\mathfrak{I})$, is the extended relation of exercise 25 a well-ordering of $\mathfrak{I}$?

▶ **37.** [*24*] (D. Ferguson.) If two computer words are necessary to contain two link fields and an INFO field, representation (2) requires $2n$ words of memory for a tree with n nodes. Design a representation scheme for binary trees which uses less space, assuming that *one* link and an INFO field will fit in a single computer word.

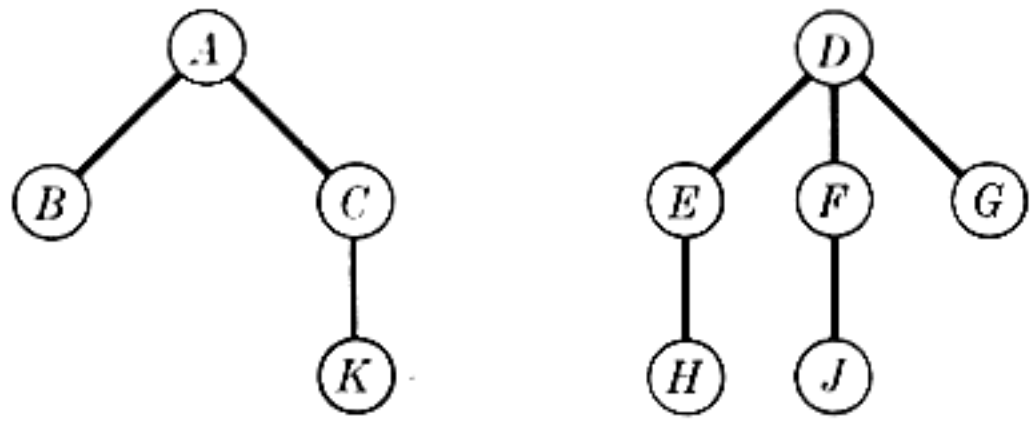
2.3.2. Binary Tree Representation of Trees

We turn now from binary trees to just plain trees. Let us recall the basic differences between trees and binary trees as we have defined them:

1) A tree is never empty, i.e., it always has at least one node; and each node of a tree can have 0, 1, 2, 3, . . . sons.
2) A binary tree can be empty, and each of its nodes can have 0, 1, or 2 sons; in the case of a single son we distinguish between a "left" son and a "right" son.

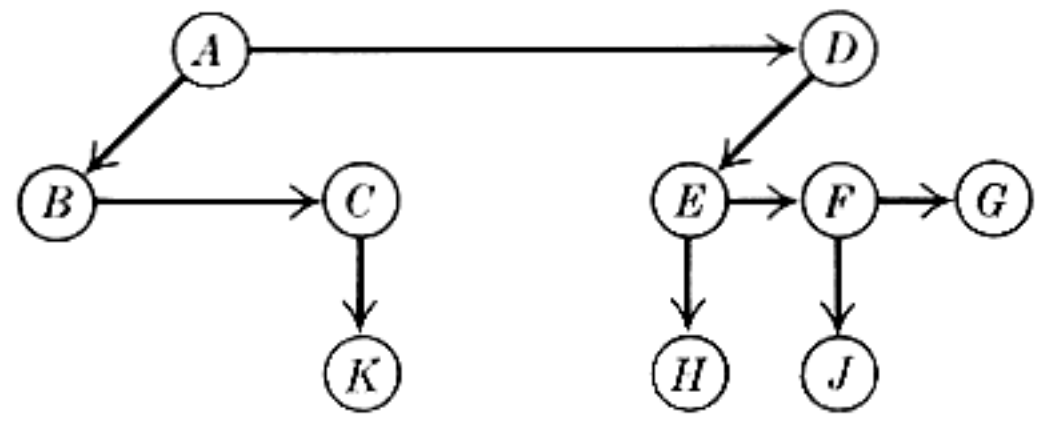
Recall also that a "forest" is an ordered set of zero or more trees. The subtrees immediately below any node of a tree form a forest.

There is a natural way to represent any forest as a binary tree. Consider the following forest of two trees:



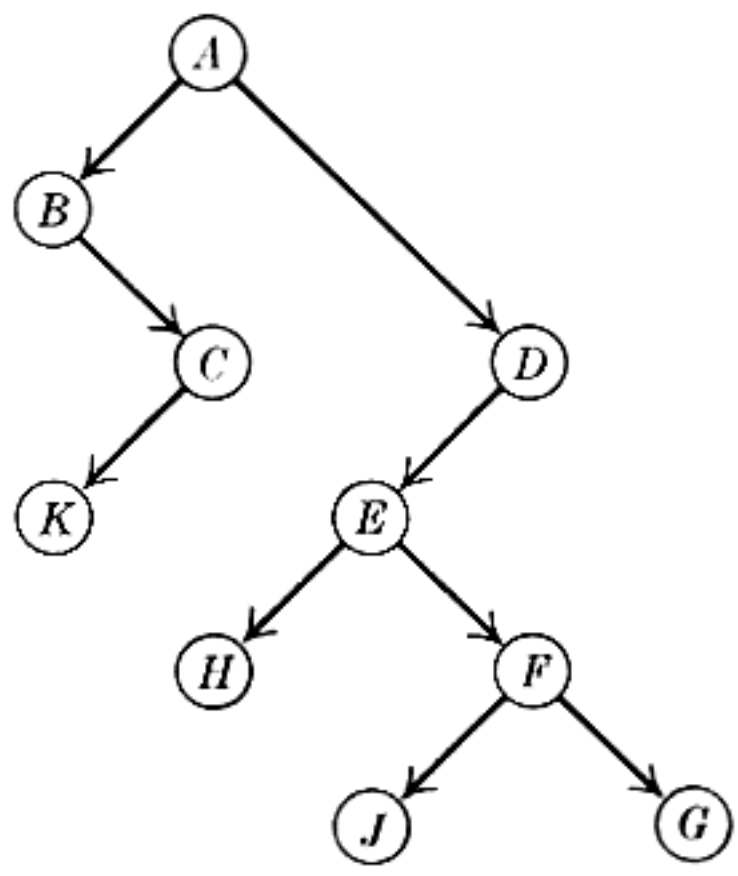
(1)

The corresponding binary tree is obtained by linking together the sons of each family and removing vertical links except from a father to his first son:



(2)

Then, tilt the diagram 45° and we have a binary tree:



(3)

Conversely, it is easy to see that any binary tree corresponds to a unique forest of trees by reversing the process.

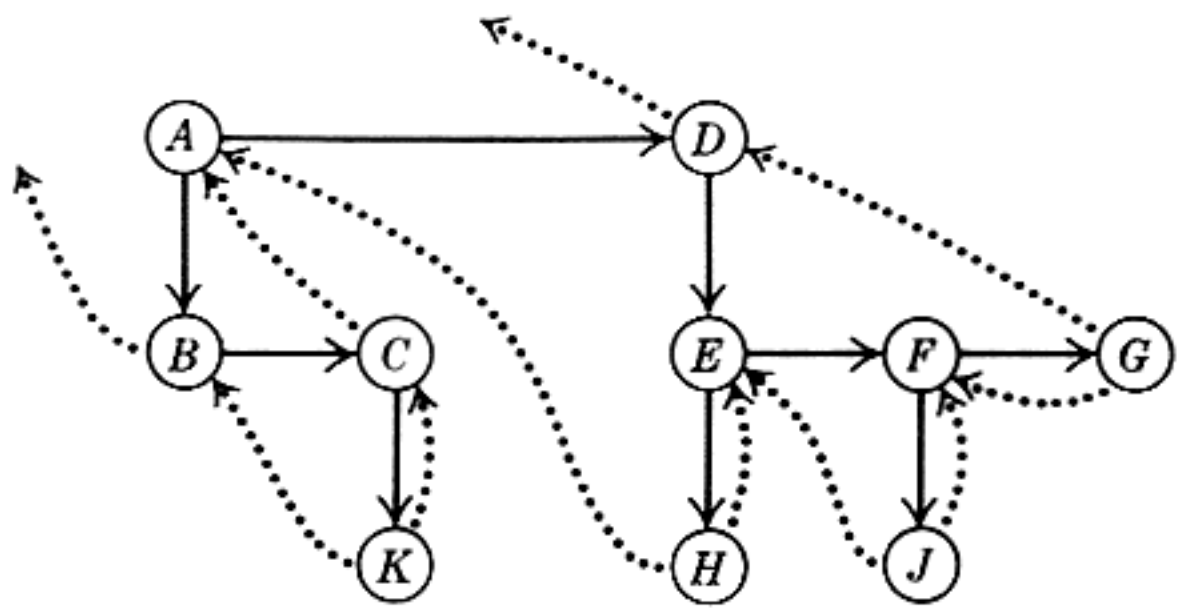
The above transformation is extremely important; it is called the *natural correspondence* between forests and binary trees. (In particular, it gives a correspondence between trees and those binary trees which have a root but no right subtree. We may also change things a little and let the list head of a binary tree correspond to the root of a tree, thus obtaining a one-to-one correspondence between trees and binary trees.)

Let $F = (T_1, T_2, \ldots, T_n)$ be a forest of trees. The binary tree $B(F)$ corresponding to F can be defined rigorously as follows:

a) If $n = 0$, $B(F)$ is empty.
b) If $n > 0$, the root of $B(F)$ is root (T_1); the left subtree of $B(F)$ is $B(T_{11}, T_{12}, \ldots, T_{1m})$, where $T_{11}, T_{12}, \ldots, T_{1m}$ are the subtrees of root (T_1); and the right subtree of $B(F)$ is $B(T_2, \ldots, T_n)$.

These rules specify the transformation from (1) to (3) precisely.

It will occasionally be convenient to draw our binary tree diagrams as in (2), without the 45° rotation. The *threaded* binary tree corresponding to (1) is



(4)

(compare with Fig. 24, giving the latter a 45° change in orientation). Note that *right thread links go from the rightmost son of a family to the father*. Left thread links do not have such a natural interpretation, due to the lack of symmetry between left and right.

The ideas about traversal expressed in the previous section can now be recast in terms of forests (and, therefore, trees) under the natural correspondence. The three orders of traversal take the following form:

Preorder	*Postorder*	*Endorder*
a) Visit the root of the first tree;	a) Traverse the subtrees of the first tree (in postorder);	a) Traverse the subtrees of the first tree (in endorder);
b) traverse the subtrees of the first tree (in preorder);	b) visit the root of the first tree;	b) traverse the remaining trees (in endorder);
c) traverse the remaining trees (in preorder).	c) traverse the remaining trees (in postorder).	c) visit the root of the first tree.

The algorithms developed in the previous section may be applied without change.

In order to understand the significance of these three methods of traversal, consider the following notation for expressing tree structure by nested parentheses:

$$(A(B, C(K)),\ D(E(H), F(J), G)). \tag{5}$$

This notation corresponds to the forest (1): we represent a tree by the information written in its root, followed by a representation of its subtrees; the representation of a nonempty forest is a parenthesized list of the representations of its trees, separated by commas.

If (1) is traversed in preorder, we visit the nodes in the sequence $A\ B\ C\ K\ D\ E\ H\ F\ J\ G$; this is simply (5) with the parentheses and commas removed. Preorder is a natural way to list the nodes of a tree: we list the root first, then the descendants. If a tree structure is represented by indentation as in Fig. 20(c), the rows appear in preorder. The section numbers of this book itself (see Fig. 21) appear in preorder; thus, for example, Section 2.3.2 follows Section 2.3.1, which follows Section 2.3, which follows Section 2.2.6.

Postorder for the nodes in (1) is $B\ K\ C\ A\ H\ E\ J\ F\ G\ D$; this is analogous to preorder, except that it corresponds to the similar parenthesis notation

$$\big((B,\,(K)C)A,\ ((H)E,\,(J)F,\,G)D\big), \tag{6}$$

in which a node appears just *after* its descendants instead of just before. The words "preorder" and "postorder" are meant to imply the difference between (5) and (6).

It is interesting to note that preorder is a time-honored concept which might meaningfully be called *dynastic order*. At the death of a king, duke, or earl, etc., the title passes to his first son, then to descendants of the first son, and finally if these all die out it passes to other sons of the family in the same way. (English custom also includes daughters in a family on the same basis as sons, except they come after all the sons.) In theory, we could take a lineal chart of all the aristocracy and write out the nodes in preorder; then if we consider only the people presently living, we would obtain the *order of succession to the throne* (except as modified by Acts of Abdication).

The nodes of (1) in endorder are

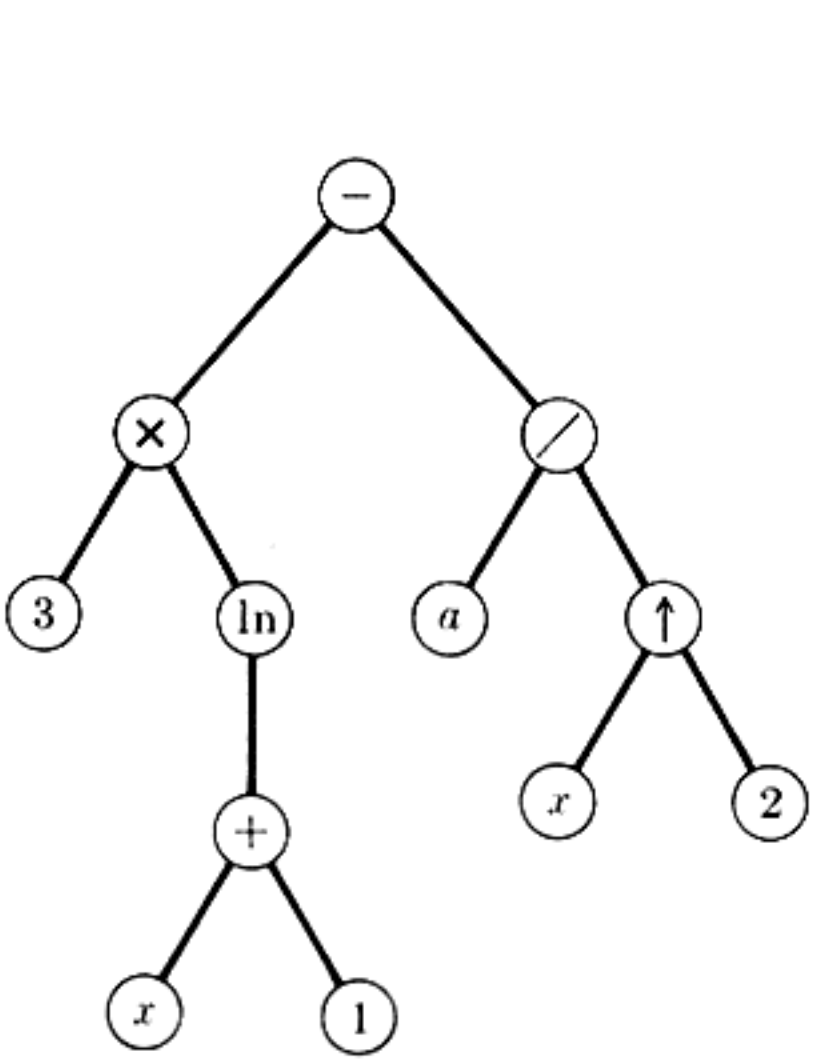
$$K\ C\ B\ H\ J\ G\ F\ E\ D\ A;$$

this may look a little strange at first, but it has a simple interpretation: Skip from left to right in either (5) or (6) until hitting the first right parenthesis; then go to the left, visiting nodes, until encountering a left parentheses; then erase the entire parenthesis group from the expression and go to the next right parenthesis, etc., until the whole expression has been erased. We will see in the next section that the reverse of endorder, for example, $A\ D\ E\ F\ G\ J\ H\ B\ C\ K$, is also important.

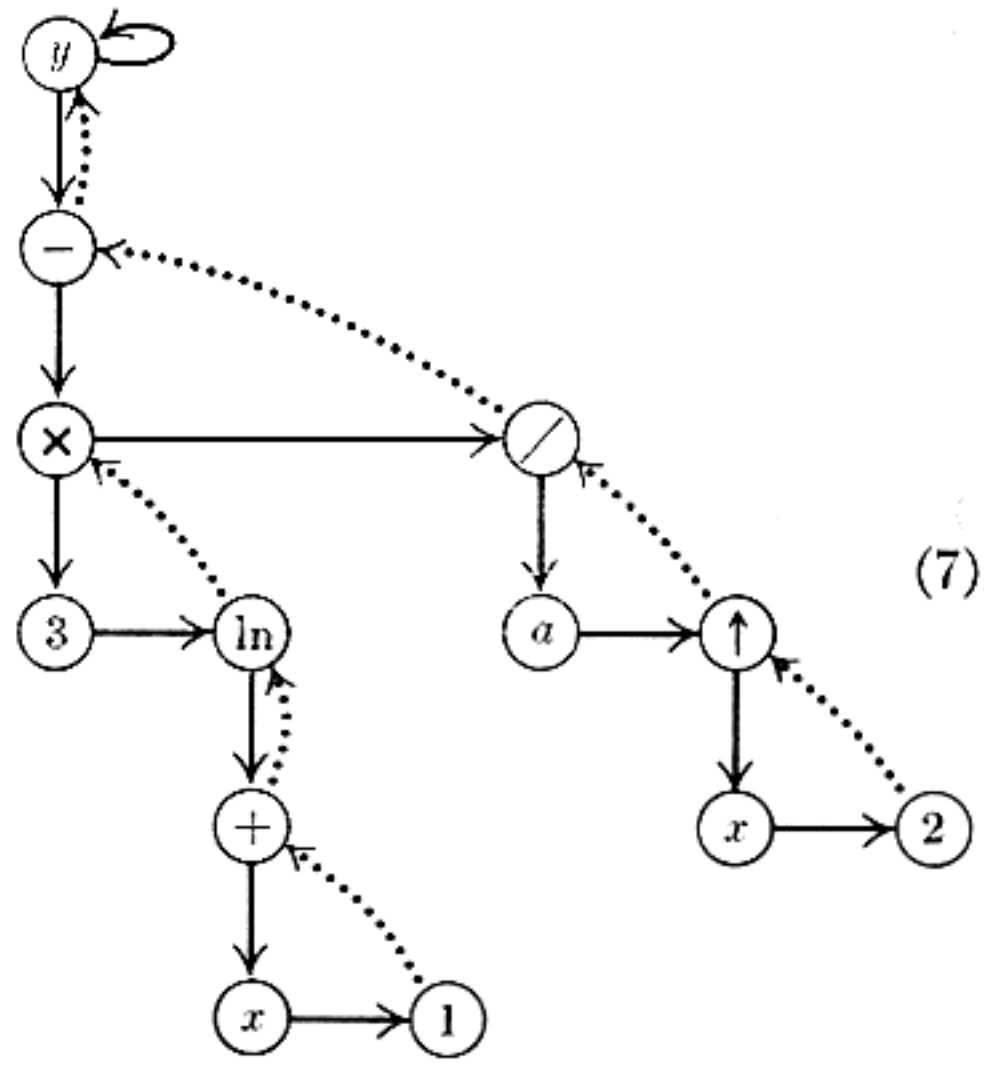
As an example of the application of these methods to a practical problem, let us consider the manipulation of algebraic formulas. Such formulas are most properly regarded as representations of tree structures, not as one- or two-dimensional configurations of symbols. The formula

$$y = 3 \ln (x + 1) - a/x^2$$

has the tree representation



i.e., (7)



Here the left-hand diagram gives the conventional tree representation, like Fig. 22, in which the binary operators $+$, $-$, $\times$, $/$, and $\uparrow$ (the latter denotes exponentiation) have two subtrees corresponding to their operands; the unary operators "ln" and "neg" (the latter does not appear in this tree; it denotes negation as in "$y = -x$") have one subtree, and variables and constants are terminal nodes. In the right-hand diagram, we have shown the equivalent right-threaded binary tree, including an additional node y which is a list head for the tree. The list head has the form described in Eq. 2.3.1-(7).

The nodes of this tree are

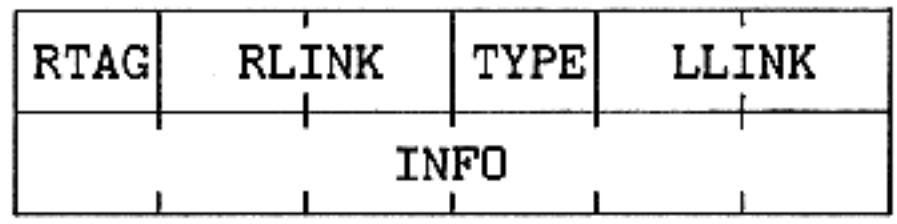
$$- \quad \times \quad 3 \quad \ln \quad + \quad x \quad 1 \quad / \quad a \quad \uparrow \quad x \quad 2 \qquad \text{in preorder;} \tag{8}$$

$$3 \quad x \quad 1 \quad + \quad \ln \quad \times \quad a \quad x \quad 2 \quad \uparrow \quad / \quad - \qquad \text{in postorder.} \tag{9}$$

Algebraic expressions like (8) and (9) are very important, and they are known as "Polish notations" because form (8) was introduced by the Polish logician, Łukasiewicz. Expression (8) is the *prefix notation* for formula (7), and (9) is the corresponding *postfix notation*. We will return to the interesting topic of Polish notation in later chapters; for now let us be content with the knowledge that Polish notation is directly related to the basic orders of tree traversal.

It is important to note that, even though the left-hand tree in (7) bears a superficial resemblance to a binary tree, we are treating it here as a *tree*, and representing it by a quite different binary tree, shown at the right in (7). Although we could develop routines for algebraic manipulations based directly on binary tree structures—these are the so-called "three-address code" representations of algebraic formulas—the author has found that several simplifications occur in practice if we use the general tree representation of algebraic formulas, as in (7), although the tree need not necessarily be threaded.

Let us assume that tree structures for the algebraic formulas with which we will be dealing have nodes of the following form:



(10)

Here RLINK and LLINK have the usual significance, and RTAG is negative for thread links. The TYPE field is used to distinguish different kinds of nodes: TYPE $= 0$ means the node represents a constant, and INFO is the value of the constant. TYPE $= 1$ means the node represents a variable, and INFO is the five-letter alphabetic name of this variable. TYPE ≥ 2 means the node represents an operator; INFO is the alphabetic name of the operator and the value TYPE $= 2, 3, 4, \ldots$ is used to distinguish the different operators $+$, $-$, $\times$, /, etc. We will not concern ourselves here with how the tree structure has been set up inside the computer memory in the first place, since this topic is analyzed in great detail in Chapter 10; let us merely assume that the tree already appears in our computer memory, and questions of input and output will be deferred until later.

We will now discuss the "classical" example of algebraic manipulation, finding the *derivative* of a formula with respect to the variable x. Programs for algebraic differentiation were among the first symbol-manipulation routines ever written for computers; they were written as early as 1952. The process of differentiation illustrates many of the techniques of algebraic manipulation, and it is of significant practical value in scientific applications.

Readers who are not familiar with mathematical calculus may consider this problem as an abstract exercise in formula manipulation, defined by the following rules:

$$D(x) = 1 \tag{11}$$

$$D(a) = 0, \quad \text{if } a \text{ is a constant or a variable} \neq x. \tag{12}$$

$$D(\ln u) = D(u)/u \tag{13}$$

$$D(-u) = -D(u) \tag{14}$$

$$D(u+v) = D(u) + D(v) \tag{15}$$

$$D(u-v) = D(u) - D(v) \tag{16}$$

$$D(u \times v) = D(u) \times v + u \times D(v) \tag{17}$$

$$D(u \,/\, v) = D(u)/v - (u \times D(v))/(v \uparrow 2) \tag{18}$$

$$D(u \uparrow v) = D(u) \times (v \times (u \uparrow (v-1))) + ((\ln u) \times D(v)) \times (u \uparrow v) \tag{19}$$

These rules allow us to evaluate the derivative $D(y)$ for any formula y composed of the above operators.

(Our main interest in this algorithm is, as usual, in the details of how the process is carried out inside a computer. There are many higher-level languages and special routines available at most computer installations which have built-in

facilities that greatly simplify algebraic manipulations like these; but the purpose of the present example is to gain more experience in fundamental tree operations.)

The idea behind the following algorithm is to traverse the tree in postorder, forming the derivative of each node as we go, until eventually the entire derivative has been calculated. Using postorder means that we arrive at an operator node (like "+") *after* its operands have been differentiated. Rules (11) through (19) imply that every subformula of the original formula will have to be differentiated, sooner or later, so we might as well do the differentiations in postorder. By using a right-threaded tree, we avoid the need for a stack during the operation of the algorithm. On the other hand, a threaded tree representation has the disadvantage that it is necessary to make copies of subtrees (for example, in the rule for $D(u \uparrow v)$ we may need to copy u and v each three times), when in many circumstances we could use a List representation instead of a tree and avoid this copying; see Section 2.3.5.

Algorithm D (*Differentiation*). If `Y` is the address of a list head which points to a formula represented as described above, and if `DY` is the address of the list head for an empty tree, this algorithm makes `NODE(DY)` point to a tree representing the analytic derivative of `Y` with respect to the variable "`X`".

D1. [Initialize.] Set `P` ← `Y$` (i.e., the first node of the tree, in postorder).

D2. [Differentiate.] Set `P1` ← `LLINK(P)` and if `P1` ≠ Λ, also set `Q1` ← `RLINK(P1)`. Then perform the routine `DIFF[TYPE(P)]`, described below. (The routines `DIFF[0]`, `DIFF[1]`, etc., will form the derivative of the tree with root `P`, and will set pointer variable `Q` to the address of the root of the derivative. The variables `P1` and `Q1` are set up first, in order to simplify the specification of the `DIFF` routines.)

D3. [Adjust link.] If `TYPE(P)` denotes a binary operator, set `RLINK(P1)` ← `P2`. (See the next step for an explanation.)

D4. [Advance to `P$`.] Set `P2` ← `P`, `P` ← `P$`. Now if `RTAG(P2)` = "+", i.e., if `NODE(P2)` has a brother on his right, set `RLINK(P2)` ← `Q`. (This is the tricky part of the algorithm: we temporarily destroy the structure of tree `Y`, so that a link to the derivative of `P2` is saved for future use. The missing link is reinserted in step D3. See exercise 21 for further discussion of this trick.)

D5. [Done?] If `P` ≠ `Y`, return to step D2. Otherwise set `LLINK(DY)` ← `Q` and `RLINK(Q)` ← `DY`, `RTAG(Q)` ← "−". ∎

The procedure described in Algorithm D is just the background routine for the differentiation operations which are performed by the processing routines `DIFF[0]`, `DIFF[1]`, . . . , called in step D2. In many ways, Algorithm D is like the control routine for an interpretive system or machine simulator, as discussed in Section 1.4.3, but it traverses a tree instead of a simple sequence of instructions.

Let us now consider the routines which do the actual differentiation. In the following discussion, the statement "`P` points to a tree" means that `NODE(P)`

is the root of a tree stored in the conventional manner, and both RLINK(P) and RTAG(P) are meaningless so far as this tree is concerned. We will make use of a *tree construction function* which makes new trees by joining smaller ones together: Let x denote some kind of node, either a constant, variable, or operator, and let U and V denote pointers to trees; then we have

TREE(x, U, V) makes a new tree with x in its root node and with U and V the subtrees of the root: W ⇐ AVAIL, INFO(W) ← x, LLINK(W) ← U, RLINK(U) ← V, RTAG(U) ← "+", RLINK(V) ← W, RTAG(V) ← "−".

TREE(x, U) similarly makes a new tree with only one subtree: W ⇐ AVAIL, INFO(W) ← x, LLINK(W) ← U, RLINK(U) ← W, RTAG(U) ← "−".

TREE(x) makes a new tree with x as a terminal root node: W ⇐ AVAIL, INFO(W) ← x, LLINK(W) ← Λ.

In all cases, the value of TREE is W, that is, a pointer to the tree just constructed. Another function, COPY(U), makes a copy of the tree pointed to by U and has as its value a pointer to the tree thereby created.

The basic functions TREE and COPY make it easy to build up a tree for the derivative of a formula, step by step. Before we look at the DIFF routines, however, let us consider what happens if we blindly apply rules (11) through (19) to a rather simple formula like

$$y = 3 \ln (x + 1) - a/x^2;$$

we get

$$D(y) = (0 \cdot \ln (x + 1) + 3((1 + 0)/(x + 1)) - (0/x^2 - (a(1(2x^{2-1}) + ((\ln x) \cdot 0)x^2))/(x^2)^2), \quad (20)$$

which is completely unsatisfactory. To avoid so many redundant operations in the answer, we must make our routines more complicated, so that they recognize the special cases of adding or multiplying by zero, multiplying by one, or raising to the first power. These simplifications reduce (20) to

$$D(y) = 3(1/(x + 1)) - ((-(a(2x)))/(x^2)^2), \quad (21)$$

and this is acceptable but not wholly satisfactory. The concept of a really satisfactory answer is not well-defined, because different mathematicians will prefer formulas to be expressed in different ways; however, it is clear that (21) is not as simple as it could be. In order to make substantial progress over formula (21), it is necessary to develop algebraic simplification routines (see exercise 17), which would reduce (21) to, for example,

$$D(y) = 3(x + 1)^{-1} + 2ax^{-3}. \quad (22)$$

We will content ourselves here with routines which can produce (21), not (22).

Nullary operators (*constants and variables*). For these operations, NODE(P) is a terminal node, and the values of P1, P2, Q1, and Q before the operation are irrelevant.

DIFF[0]: (NODE(P) is a constant.) Set Q ← TREE("0").

DIFF[1]: (NODE(P) is a variable.) If INFO(P) = "X", set Q ← TREE("1"); otherwise set Q ← TREE("0").

Unary operators (*logarithm and negation*). For these operations, NODE(P) has one son, U, pointed to by P1, and Q points to $D(U)$. The values of P2 and Q1 before the operation are irrelevant.

DIFF[2]: (NODE(P) is "ln".) If INFO(Q) $\neq 0$, set Q ← TREE("/",Q,COPY(P1)).

DIFF[3]: (NODE(P) is "neg".) If INFO(Q) $\neq 0$, set Q ← TREE("neg", Q).

Binary operators (*addition, subtraction, etc.*). For these operations, NODE(P) has two sons, U and V, pointed to respectively by P1 and P2; Q1 and Q point respectively to $D(U)$, $D(V)$.

DIFF[4]: ("+" operation.) If INFO(Q1) $= 0$, set AVAIL ⇐ Q1. Otherwise if INFO(Q) $= 0$, set AVAIL ⇐ Q and Q ← Q1; otherwise set Q ← TREE("+",Q1,Q).

DIFF[5]: ("−" operation.) If INFO(Q) $= 0$, set AVAIL ⇐ Q and Q ← Q1. Otherwise if INFO(Q1) $= 0$, set AVAIL ⇐ Q1 and set Q ← TREE("neg", Q); otherwise set Q ← TREE("−",Q1,Q).

DIFF[6]: ("×" operation.) If INFO(Q1) $\neq 0$, set Q1 ← MULT(Q1,COPY(P2)). Then if INFO(Q) $\neq 0$, set Q ← MULT(COPY(P1),Q). Then go to DIFF[4].

Here MULT(U,V) is a new function which constructs a tree for U × V but also makes a test to see if U or V is equal to "1":

if INFO(U) $= 1$ and TYPE(U) $= 0$, set AVAIL ⇐ U and MULT(U,V) ← V;
if INFO(V) $= 1$ and TYPE(V) $= 0$, set AVAIL ⇐ V and MULT(U,V) ← U;
otherwise set MULT(U;V) ← TREE("×",U,V).

DIFF[7]: ("/" operation.) If INFO(Q1) $\neq 0$, set

Q1 ← TREE("/",Q1,COPY(P2)).

Then if INFO(Q) $\neq 0$, set

Q ← TREE("/",MULT(COPY(P1),Q),TREE("↑",COPY(P2),TREE("2"))).

Then go to DIFF[5].

DIFF[8]: ("↑" operation.) See exercise 12.

We conclude this section by showing how all of the above operations are readily transformed into a computer program, starting "from scratch" with only MIX machine language as a basis.

Program D (*Differentiation*). The following MIXAL program performs Algorithm D, with rI2 ≡ P, rI3 ≡ P2, rI4 ≡ P1, rI5 ≡ Q, rI6 ≡ Q1. The order of computations has been rearranged a little, for convenience.

```
01    * DIFFERENTIATION IN A RIGHT-THREADED TREE
02    LLINK     EQU   4:5          Definition of fields, see (10)
03    RLINK     EQU   1:2
04    RLINKT    EQU   0:2
05    TYPE      EQU   3:3
06    * MAIN CONTROL ROUTINE       D1. Initialize.
07    D1        STJ   9F           Treat whole procedure as a subroutine.
08              LD4   Y(LLINK)     P1 ← LLINK(Y), prepare to find Y$.
09    1H        ENT2  0,4          P ← P1.
10    2H        LD4   0,2(LLINK)   P1 ← LLINK(P).
11              J4NZ  1B           If P1 ≠ Λ, repeat.
12    D2        LD1   0,2(TYPE)    D2. Differentiate.
13              JMP   *+1,1        Jump to DIFF[TYPE(P)].
14              JMP   CONSTANT     Switch table entry for DIFF[0].
15              JMP   VARIABLE                          DIFF[1].
16              JMP   LN                                   .
17              JMP   NEG                                  .
18              JMP   ADD                                  .
19              JMP   SUB                                  .
20              JMP   MUL                                  .
21              JMP   DIV                                  .
22              JMP   PWR                               DIFF[8].
23    D3        ST3   0,4(RLINK)   D3. Adjust link. RLINK(P1) ← P2.
24    D4        ENT3  0,2          D4. Advance to P$. P2 ← P.
25              LD2   0,2(RLINKT)  P ← RLINKT(P).
26              J2N   1F           Jump if RTAG(P) = "—";
27              ST5   0,3(RLINK)   otherwise set RLINK(P2) ← Q.
28              JMP   2B           Note that NODE(P$) will be terminal.
29    1H        ENN2  0,2
30    D5        ENT1  -Y,2         D5. Done?
31              LD4   0,2(LLINK)   P1 ← LLINK(P), prepare for step D2.
32              LD6   0,4(RLINK)   Q1 ← RLINK(P1).
33              J1NZ  D2           Jump to D2 if P ≠ Y;
34              ST5   DY(LLINK)    otherwise set LLINK(DY) ← Q.
34a             ENNA  DY
34b             STA   0,5(RLINKT)  RLINK(Q) ← DY, RTAG(Q) ← "-".
35    9H        JMP   *            Exit from differentiation subroutine. ∎
```

The next part of the program contains the basic subroutines TREE and COPY. The former has three entrances, TREE0, TREE1, and TREE2, according to the number of subtrees of the tree being constructed. Regardless of which entrance to the subroutine is used, rA will contain the address of a special constant indicating what type of node forms the root of the tree being constructed; these special constants appear in lines 103–122.

```
36   * BASIC SUBROUTINES FOR TREE CONSTRUCTION
37   TREE0    STJ   9F          TREE(rA) function
38            JMP   2F
39   TREE1    ST1   3F(0:2)     TREE(rA, rI1) function
40            JSJ   1F
41   TREE2    STX   3F(0:2)     TREE(rA, rX, rI1) function
42   3H       ST1   *(RLINKT)   RLINK(rX) ← rI1, RTAG(rX) ← "+".
43   1H       STJ   9F
44            LDXN  AVAIL
45            JXZ   OVERFLOW
46            STX   0,1(RLINKT) RLINK(rI1) ← AVAIL, RTAG(rI1) ← "−".
47            LDX   3B(0:2)
48            STA   *+1(0:2)
49            STX   *(LLINK)    Set LLINK of next root node.
50   2H       LD1   AVAIL       rI1 ⇐ AVAIL.
51            J1Z   OVERFLOW
52            LDX   0,1(LLINK)
53            STX   AVAIL
54            STA   *+1(0:2)    Move root node to available space.
55            MOVE  *(2)
56            DEC1  2           Reset rI1 to point to root node.
57   9H       JMP   *           Exit from TREE, result in rI1
58   COPYP1   ENT1  0,4         COPY(P1), special entrance to COPY
59            JSJ   COPY
60   COPYP2   ENT1  0,3         COPY(P2), special entrance to COPY
61   COPY     STJ   9F          COPY(rI1) function
 ⋮            ⋮                (see exercise 13)
102  9H       JMP   *           Exit from COPY, rI1 points to new tree.
103  CON0     CON   0           Node representing constant "0"
104           CON   0
105  CON1     CON   0           Node representing "1"
106           CON   1
107  CON2     CON   0           Node representing "2"
108           CON   2
109  LOG      CON   2(TYPE)     Node representing "ln"
110           ALF     LN
111  NEGOP    CON   3(TYPE)     Node representing "neg"
112           ALF     NEG
113  PLUS     CON   4(TYPE)     Node representing "+"
114           ALF     +
115  MINUS    CON   5(TYPE)     Node representing "−"
116           ALF     -
117  TIMES    CON   6(TYPE)     Node representing "×"
118           ALF     *
119  SLASH    CON   7(TYPE)     Node representing "/"
120           ALF     /
121  UPARROW  CON   8(TYPE)     Node representing "↑"
122           ALF     **
```

The remaining portion of the program corresponds to the differentiation routines DIFF[0], DIFF[1], . . . ; these routines are written to return control to step D3 after processing a binary operator, otherwise control is to return to step D4.

```
123    * DIFFERENTIATION ROUTINES
124 VARIABLE  LDX   1,2
125           ENTA  CON1
126           CMPX  2F          Is INFO(P) = "X"?
127           JE    *+2         If so, call TREE("1").
128 CONSTANT  ENTA  CON0        Call TREE("0").
129           JMP   TREE0
130 1H        ENT5  0,1         Q ← location of new tree.
131           JMP   D4          Return to control routine.
132 2H        ALF     X
133 LN        LDA   1,5
134           JAZ   D4          Return to control routine if INFO(Q) = 0;
135           JMP   COPYP1      otherwise set rI1 ← COPY(P1).
136           ENTX  0,5
137           ENTA  SLASH
138           JMP   TREE2       rI1 ← TREE("/", Q, rI1).
139           JMP   1B          Q ← rI1, return to control.
140 NEG       LDA   1,5
141           JAZ   D4          Return if INFO(Q) = 0.
142           ENTA  NEGOP
143           ENT1  0,5
144           JMP   TREE1       TREE("neg", Q)
145           JMP   1B          → Q, return to control.
146 ADD       LDA   1,6
147           JANZ  1F          Jump unless INFO(Q1) = 0.
148 3H        LDA   AVAIL       AVAIL ⇐ Q1.
149           STA   0,6(LLINK)
150           ST6   AVAIL
151           JMP   D3          Return to control, binary operator.
152 1H        LDA   1,5
153           JANZ  1F          Jump unless INFO(Q) = 0.
154 2H        LDA   AVAIL       AVAIL ⇐ Q.
155           STA   0,5(LLINK)
156           ST5   AVAIL
157           ENT5  0,6         Q ← Q1.
158           JMP   D3          Return to control.
159 1H        ENTA  PLUS        Prepare to call TREE("+", Q1, Q).
160 4H        ENTX  0,6
161           ENT1  0,5
162           JMP   TREE2
163           ENT5  0,1         Q ← TREE("±", Q1, Q).
164           JMP   D3          Return to control.
```

```
165  SUB     LDA   1,5
166          JAZ   2B          Jump if INFO(Q) = 0.
167          LDA   1,6
168          JANZ  1F          Jump unless INFO(Q1) = 0.
169          ENTA  NEGOP
170          ENT1  0,5
171          JMP   TREE1
172  .       ENT5  0,1         Q ← TREE("neg", Q).
173          JMP   3B          AVAIL ⇐ Q1 and return.
174  1H      ENTA  MINUS       Prepare to call TREE("—", Q1, Q).
175          JMP   4B
176  MUL     LDA   1,6
177          JAZ   1F          Jump if INFO(Q1) = 0;
178          JMP   COPYP2      otherwise set rI1 ← COPY(P2).
179          ENTA  0,6
180          JMP   MULT        MULT(Q1, COPY(P2))
181          ENT6  0,1         → Q1.
182  1H      LDA   1,5
183          JAZ   ADD         Jump if INFO(Q) = 0;
184          JMP   COPYP1      otherwise set rI1 ← COPY(P1).
185          ENTA  0,1
186          ENT1  0,5
187          JMP   MULT        MULT(COPY(P1), Q)
188          ENT5  0,1         → Q.
189          JMP   ADD
190  MULT    STJ   9F          MULT(rA, rI1) subroutine
191          STA   1F(0:2)     Let rA ≡ U, rI1 ≡ V.
192          ST2   8F(0:2)     Save rI2.
193  1H      ENT2  *           rI2 ← U.
194          LDA   1,2         Test if INFO(U) = 1
195          DECA  1
196          JANZ  1F
197          LDA   0,2(TYPE)   and if TYPE(U) = 0.
198          JAZ   2F
199  1H      LDA   1,1         If not, test if INFO(V) = 1
200          DECA  1
201          JANZ  1F
202          LDA   0,1(TYPE)   and if TYPE(V) = 0.
203          JANZ  1F
204          ST1   *+2(0:2)    If so, interchange U ↔ V.
205          ENT1  0,2
206          ENT2  *
207  2H      LDA   AVAIL       AVAIL ⇐ U.
208          STA   0,2(LLINK)
209          ST2   AVAIL
210          JMP   8F          Result is V.
211  1H      ENTA  TIMES
212          ENTX  0,2
```

```
213          JMP    TREE2      Result is TREE("×", U, V).
214    8H    ENT2   *          Restore rI2 setting.
215    9H    JMP    *          Exit MULT with result in rI1. ▌
```

The other two routines DIV and PWR are similar and they have been left as exercises (see exercises 15 and 16).

EXERCISES

▸ **1.** [*20*] The text gives a formal definition of $B(F)$, the binary tree corresponding to a forest F. Give a formal definition which reverses the process, i.e., define $F(B)$, the forest corresponding to a binary tree B.

▸ **2.** [*20*] We defined Dewey decimal notation for forests in Section 2.3, and for binary trees in exercise 2.3.1–5. Thus the node "*J*" in (1) is represented by "2.2.1", and in the equivalent binary tree (3) it is represented by "11010". If possible, give a rule that directly expresses the natural correspondence between trees and binary trees as a correspondence between the Dewey decimal notations.

3. [*22*] What is the relation between Dewey decimal notation for the nodes of a forest and the preorder and postorder of these nodes?

4. [*21*] Is the following statement true or false? "The terminal nodes of a tree occur in the same relative position in preorder, postorder, and endorder."

5. [*24*] The text gives various "intuitive" interpretations of preorder, postorder, and endorder. Give an "intuitive" interpretation of *reverse* endorder [for example, *A D E F G J H B C K* in the forest (1)].

▸ **6.** [*25*] Let T be a nonempty binary tree in which each node has 0 or 2 sons. If we regard T as an ordinary tree, it corresponds (via the natural correspondence) to *another* binary tree T'. Is there any simple relation between preorder, postorder, and endorder of the nodes of T (as defined for binary trees) and the same three orders for the nodes of T'?

7. [*M20*] A forest may be regarded as a partial ordering, if we say that each node precedes its descendants in the tree. Are the nodes topologically sorted (as defined in Section 2.2.3) when they are listed in (a) preorder? (b) postorder? (c) endorder? (d) reverse preorder? (e) reverse postorder? (f) reverse endorder?

8. [*M20*] Exercise 2.3.1–25 shows how an ordering between the information stored in the individual nodes of a binary tree may be extended to a linear ordering of all binary trees. The same construction leads to an ordering of all trees, under the natural correspondence. Reformulate the definition of that exercise, in terms of trees.

9. [*M21*] Show that the total number of nonterminal nodes in a forest has a simple relation to the total number of right links equal to Λ in the corresponding binary tree.

10. [*M23*] Let F be a forest of trees whose nodes in preorder are $u_1, u_2, \ldots, u_n$, and let F' be a forest whose nodes in preorder are $u'_1, u'_2, \ldots, u'_{n'}$. Let $S(u)$ denote the degree (the number of sons) of node u. In terms of these ideas, formulate and prove a theorem analogous to Theorem 2.3.1A.

11. [*20*] Draw trees analogous to those shown in (7), corresponding to the formula $y = e^{-x^2}$.

12. [*M21*] Give specifications for the routine DIFF[8] (the "↑" operation), which was omitted from the algorithm in the text.

▶ **13.** [*26*] Write a MIX program for the COPY subroutine (which fits in the program of the text between lines 61–102). [*Hint:* Adapt Algorithm 2.3.1C to the case of right-threaded binary trees, with suitable initial conditions.]

▶ **14.** [*M21*] How long does it take the program of exercise 13 to copy a tree with n nodes?

15. [*23*] Write a MIX program for the DIV routine, corresponding to DIFF[7] as specified in the text. (This routine should be added to the program in the text after line 215.)

16. [*24*] Write a MIX program for the PWR routine, corresponding to DIFF[8] as specified in exercise 12. (This routine should be added to the program in the text after the solution to exercise 15.)

17. [*M40*] Write a program to do algebraic simplification capable of reducing, for example, (20) or (21) to (22). [*Hints:* Include a new field with each node, representing its coefficient (for summands) or its exponent (for factors in a product). Apply algebraic identities, like replacing $\ln(u \uparrow v)$ by $v \ln u$, and remove the operations $-$, $/$, $\uparrow$, and neg when possible by using equivalent addition or multiplication operations. Make $+$ and $\times$ into n-ary instead of binary operators; collect like terms by sorting their operands in tree order (exercise 8); some sums and products will now reduce to zero or unity, presenting perhaps further simplifications. Other adjustments, like replacing a sum of logarithms by the logarithm of a product, also suggest themselves.] For references, see the survey article by J. Sammet, *CACM* **9** (1966), 555–569.

18. [*M40*] Consider algebraic formulas which are composed of the operators of symbolic logic (AND, OR, and NOT, say). Try several different algorithms for deciding whether or not such a formula is a *tautology*, i.e., is true for all combinations of truth values of its variables. [For example,

((X AND Y) OR NOT X) OR NOT(Y AND Z)

is a tautology.] For each algorithm considered, analyze its computational efficiency. For examples of algorithms to try, see H. Wang, *IBM J. Res. Dev.* **4** (1960), 2–22, and *CACM* **3** (1960), 220–234; see also Dunham, Fridshal and Sward, *Information Processing* (Proceedings of the International Congress of Information Processing, UNESCO, Paris, 1959), 282–285. Another idea is to use the Boolean operations of a binary computer on 2^n-bit quantities, where n is the number of variables.

19. [*M35*] A *free lattice* is a mathematical system, which (for the purposes of this exercise) can be simply defined as the set of all formulas composed of variables and two abstract binary operators "$\cup$" and "$\cap$". A relation "$X \supseteq Y$" is defined between certain formulas X and Y in the free lattice by the following rules:

a) $X \cup Y \supseteq Z$ if and only if $X \supseteq Z$ or $Y \supseteq Z$

b) $X \cap Y \supseteq Z$ if and only if $X \supseteq Z$ and $Y \supseteq Z$

c) $X \supseteq Y \cup Z$ if and only if $X \supseteq Y$ and $X \supseteq Z$

d) $X \supseteq Y \cap Z$ if and only if $X \supseteq Y$ or $X \supseteq Z$

e) $v_1 \supseteq v_2$ if and only if $v_1 = v_2$, for variables v_1, v_2.

For example, we find $a \cap (b \cup c) \supseteq (a \cap b) \cup (a \cap c) \not\supseteq a \cap (b \cup c)$.

Design an algorithm which tests whether or not $X \supseteq Y$, given two formulas X and Y in the free lattice.

▸ **20.** [*M22*] Prove that if u and v are nodes of a tree, u is an ancestor of v if and only if u precedes v in preorder and u follows v in postorder.

21. [*25*] Algorithm D controls the differentiation activity for binary operators, unary operators, and "nullary" operators, i.e., for trees whose nodes have degree 2, 1, or 0; but it does not indicate explicitly how the control would be handled for ternary operators and nodes of higher degree. (For example, exercise 17 suggests making addition and multiplication into operators with any number of operands.) Is it possible to extend Algorithm D in a simple way so that it will handle operators of degree more than 2?

▸ **22.** [*M26*] If T and T' are trees, let us say T *can be embedded in* T', written $T \subseteq T'$, if there is a one-to-one function f from the nodes of T into the nodes of T' such that f preserves both preorder and postorder. (In other words, u precedes v in preorder for T if and only if $f(u)$ precedes $f(v)$ in preorder for T', and the same holds for postorder. See Fig. 25.)

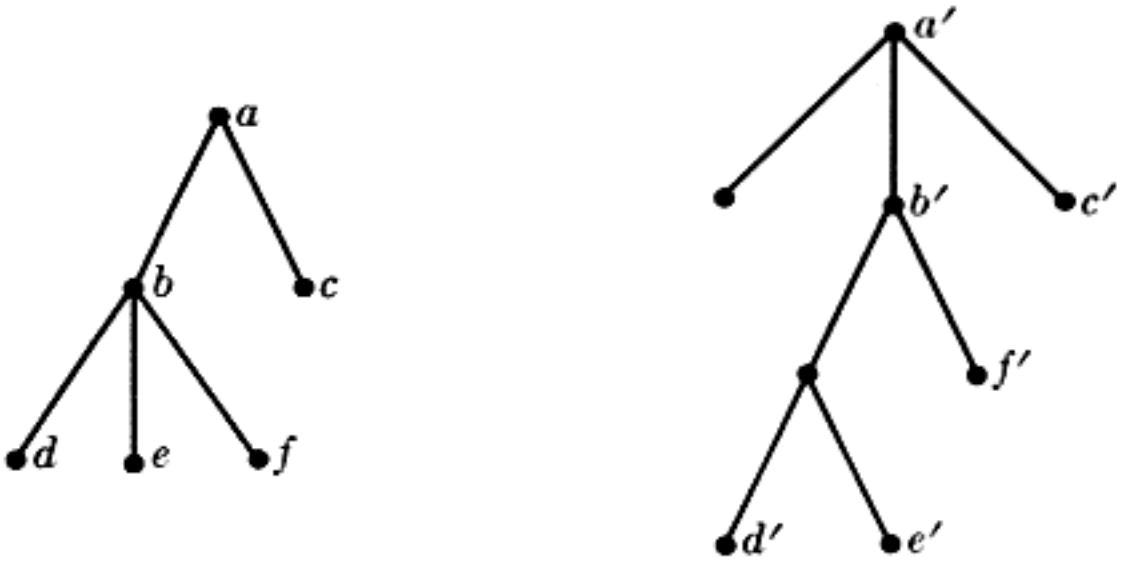


Fig. 25. One tree embedded in another (see exercise 22).

If T has more than one node, let $l(T)$ be the leftmost subtree of root (T) and let $r(T)$ be the rest of T, that is, T with $l(T)$ deleted. Prove that T can be embedded in T' if either T has just one node, or both T and T' have more than one node and either $T \subseteq l(T')$, or $T \subseteq r(T')$, or $l(T) \subseteq l(T')$ and $r(T) \subseteq r(T')$. Does the converse hold?

2.3.3. Other Representations of Trees

There are many ways to represent tree structures inside a computer besides the LLINK–RLINK (left son–right sibling) method given in the previous section. As usual, the proper choice of representation depends heavily on what kind of operations we want to perform on the trees. In this section we will consider a few of the possible tree representation methods that have proved to be useful.

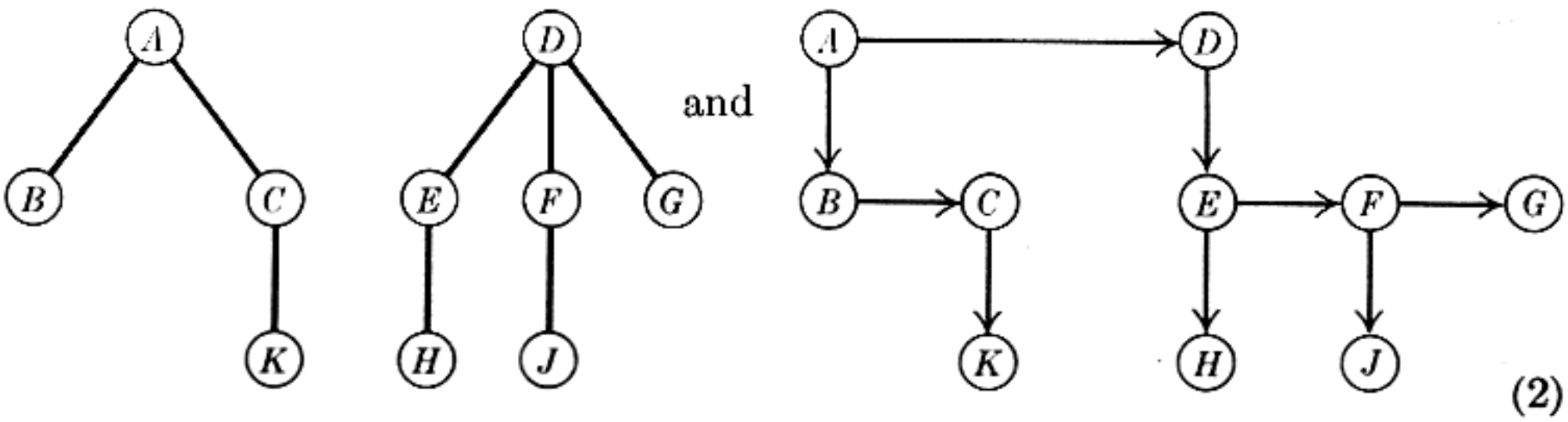
First we can use *sequential* memory techniques. As in the case of linear lists, this mode of allocation is most suitable when we want a compact representation

of a tree structure that is not going to be subject to radical dynamic changes in size or shape during program execution. There are many situations in which we need essentially constant tables of tree structures for reference within a program, and the desired form of these trees in memory depends on the way in which these tables are to be examined.

The two most common sequential representations of trees (and forests) correspond essentially to omission of `LLINK` or of `RLINK`; consecutive storage of the nodes takes the place of one type of link. Let us consider first the case that `LLINK`s are omitted, and look at the forest

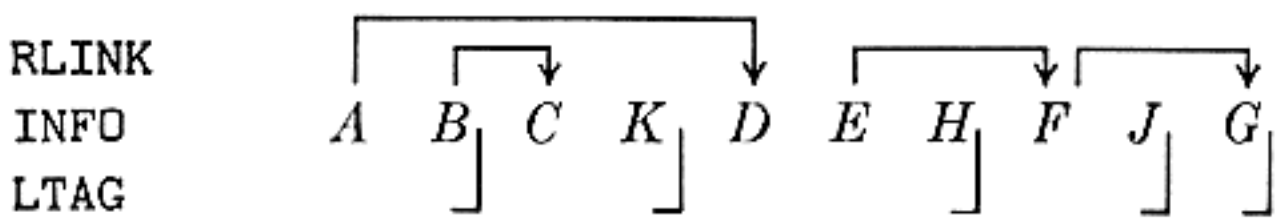
$$(A(B, C(K)),\ D(E(H), F(J), G)) \tag{1}$$

considered in the previous section, which has the tree diagrams



(2)

The *preorder sequential representation* has the nodes appearing in preorder, with the fields `INFO`, `RLINK`, and `LTAG` in each node:



(3)

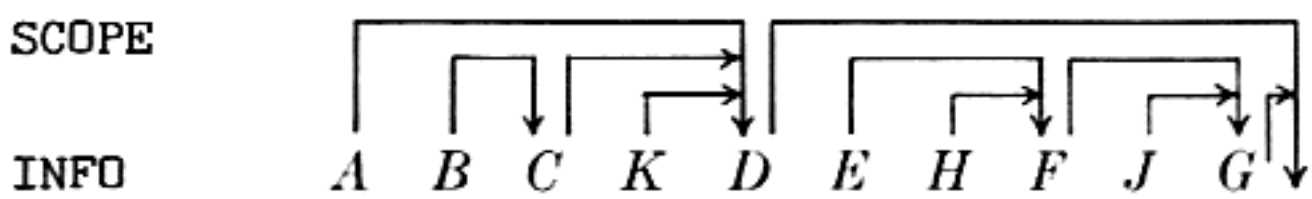
Here nonnull `RLINK`s have been indicated by arrows, and `LTAG` = "—" (for terminal nodes) is indicated by "⌋". `LLINK` is unnecessary, since it would either be null or it would point to the next item in sequence.

This representation has several interesting properties. In the first place, all subtrees of a node appear immediately after that node, so that all subtrees within the original forest appear in consecutive blocks. [Compare this with the "nested parentheses" in (1) and in Fig. 20(b).] In the second place, note that the `RLINK` arrows never cross each other in (3); this will be true in general, for in a binary tree all nodes between `X` and `RLINK(X)` in preorder lie in the left subtree of `X`, and so no outward arrows will emerge from that part of the tree. In the third place, we may observe that the `LTAG` field, which indicates whether a node is terminal or not, is redundant, since "⌋" occurs only at the end of the forest and just *preceding* every downward pointing arrow. Thus if we would proceed from left to right in (3), maintaining a suitable stack to remember the locations of `RLINK`s that have not yet been encountered, we could tell what the

value of LTAG must be. (This observation is interesting from a theoretical point of view, but in practice it is usually advantageous to include LTAG or some equivalent, since it takes only one bit of a node anyway.) The reader will find it interesting to compare (1) with (3).

Note that representation (3) involves some wasted space, since over half of the RLINK fields are equal to Λ for this particular forest. There are two common ways to make use of the wasted space:

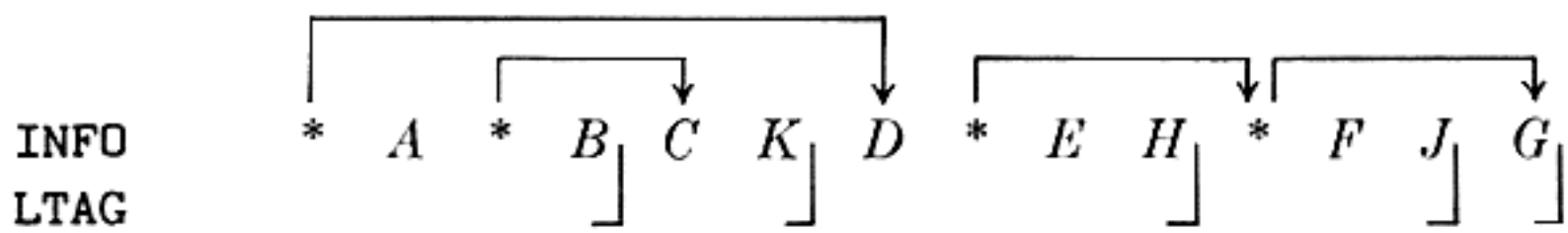
1) Fill in RLINK of each node to the address following the subtree below that node. The field is now often called "SCOPE" instead of RLINK, since it indicates the right boundary of the "influence" (descendants) of each node. Instead of (3), we would have



(4)

The arrows still do not cross each other. Furthermore, LTAG(X) = "−" is characterized by the condition SCOPE(X) = X + c, given that c is the number of words per node. One example of the use of this SCOPE idea appears in exercise 2.4–12.

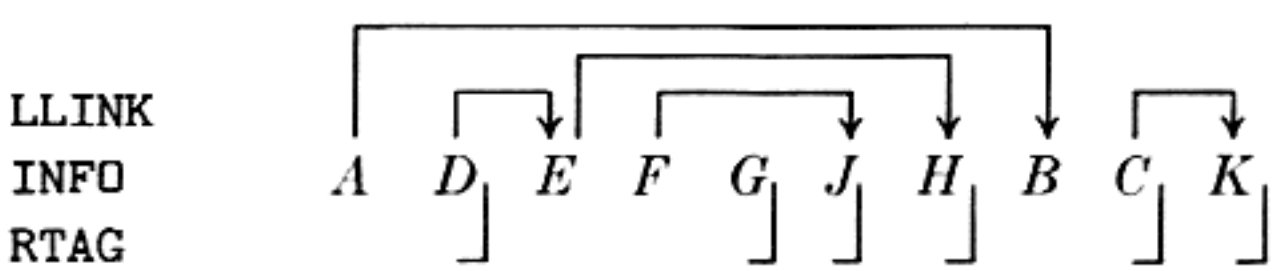
2) Decrease the size of each node by removing the RLINK field, and add special "link" nodes just before nodes that formerly had a nonnull RLINK:



(5)

Here "*" indicates the special link nodes, whose INFO somehow characterizes them as links pointing as shown by the arrows. If the INFO and RLINK fields of (3) occupy roughly the same amount of space, the net effect of the change to (5) is to consume less memory, since the number of "*" nodes is always less than the number of non-"*" nodes. Representation (5) is somewhat analogous to a sequence of instructions in a one-address computer like MIX, with the "*" nodes corresponding to conditional jump instructions.

The *family-order sequential representation* of a tree differs from the preorder representation in that it omits RLINK but retains LLINK. It has the nodes appearing in the *reverse* of endorder, which in the case of trees (2) would be



(6)

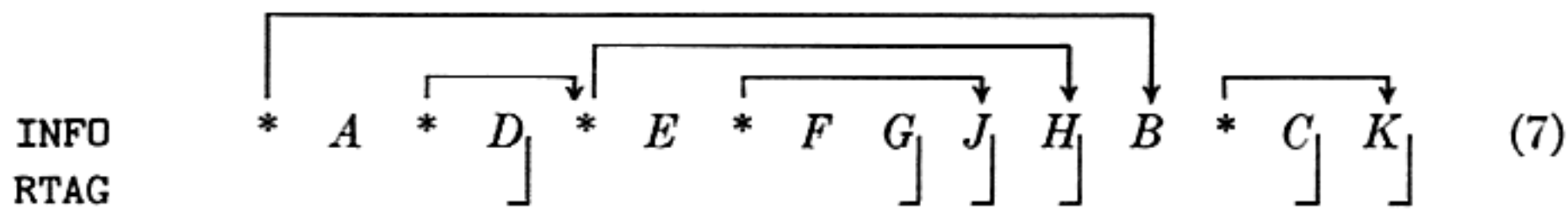
Reverse endorder may be called family-order because all sons of the same family are adjacent; thus, for example, E F G appear together in (6). (A similar

but not identical ordering has been called *level order* by G. Salton; see *CACM* **5** (1962), 103–114.) Family-order for any forest may be recursively defined as follows:

a) visit the root of the first tree;
b) traverse the remaining trees, in family-order;
c) traverse the subtrees of the root of the first tree, in family-order.

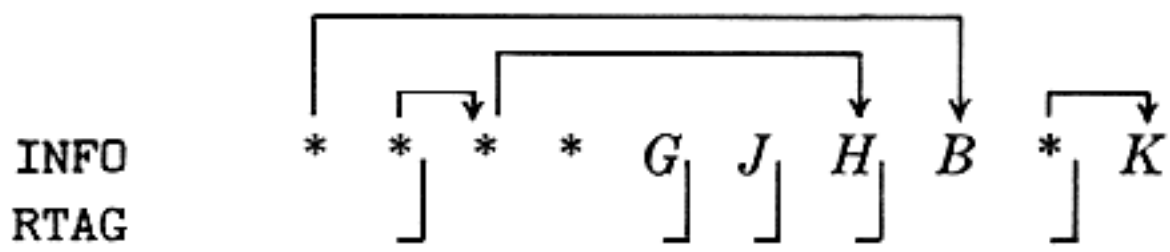
(Compare with the definition of endorder in the previous section.)

Representation (6) is in a sense the "natural" analog, for trees, of the sequential representation of linear lists, if we think of a forest as a list of trees, and if we think of each tree as one item of INFO plus a list of subtrees. Each family is a sequentially-stored list. The family-order representation has interesting properties analogous to those of the preorder representation: (a) The arrows never cross each other (see exercise 1). (b) The RTAG field, which marks the end of each family, is redundant, just as LTAG was redundant in the case of preorder. (c) To compress memory in certain circumstances, a representation analogous to (5) may be used:



(7)

In this case there are often situations where nonterminal nodes carry no information, so the node following each "*" node may be deleted. Thus, if the names A, D, E, F, and C do not need to be stored, (7) becomes



(8)

which is like (6) with only INFO and RTAG fields in each node.

The reader will easily see how to design algorithms that traverse and analyze trees represented sequentially as above, since the LLINK and RLINK information is essentially available just as though we had a fully linked tree structure.

A third method, *postorder sequential representation*, is somewhat different from the above two. We list the nodes in postorder and give the *degree* of each node instead of links:

DEGREE	0	0	1	2	0	1	0	1	0	3
INFO	*B*	*K*	*C*	*A*	*H*	*E*	*J*	*F*	*G*	*D*

(9)

For a proof that this is sufficient to characterize the tree structure, see exercise 2.3.2–10. This order is useful for the evaluation of certain functions defined on the nodes of a tree.

Algorithm F (*Evaluate a locally defined function in a tree*). Suppose f is a function of the nodes of a tree, such that the value of f at a node x depends only on x and the values of f on the sons of x. The following algorithm, using an auxiliary stack, evaluates f at each node of a nonempty forest.

F1. [Initialize.] Set the stack empty, and let `P` point to the first node of the forest in postorder.

F2. [Evaluate f.] Set $d \leftarrow$ `DEGREE(P)`. (The first time this step is reached, d will be zero. In general, when we get to this point, it will always be true that the top d items of the stack are $f(x_d), \ldots, f(x_1)$—from the top of the stack downward—where $x_1, \ldots, x_d$ are the sons of `NODE(P)` from left to right.) Evaluate f(`NODE(P)`), using the values of $f(x_d), \ldots, f(x_1)$ found on the stack.

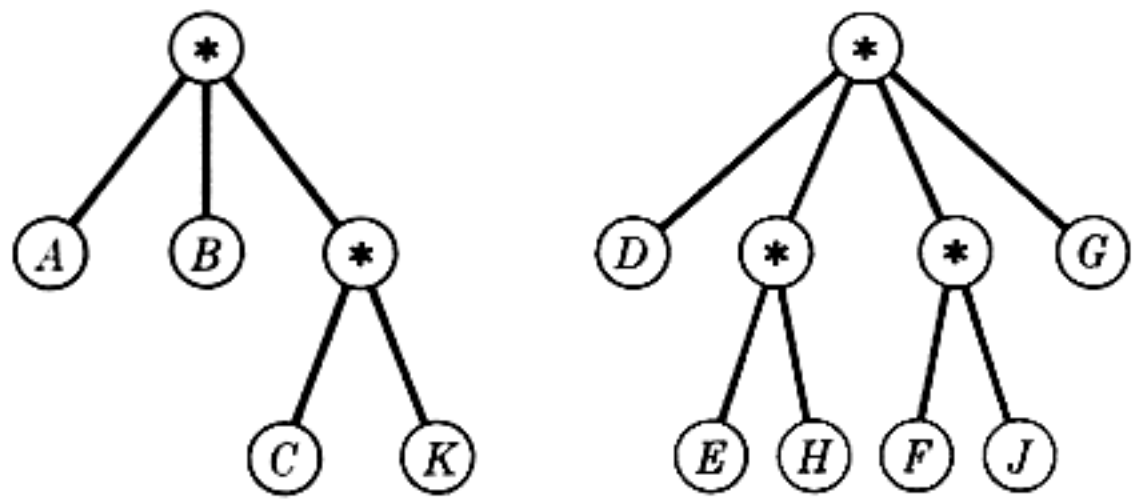
F3. [Update the stack.] Remove the top d items of the stack, and then put the value f(`NODE(P)`) on top of the stack.

F4. [Advance.] If `P` is the last node in postorder, terminate the algorithm. (Then the stack contains $f(\text{root}\,(T_m)), \ldots, f(\text{root}\,(T_1))$, from top to bottom, where $T_1, \ldots, T_m$ are the trees of the given forest.) Otherwise set `P` $\leftarrow$ `P$` [this would be simply `P` $\leftarrow$ `P` $+ 1$ in the representation (9)], and return to step F2. ∎

The validity of Algorithm F follows by induction on the size of the trees processed (see exercise 17). This algorithm bears a striking similarity to the differentiation algorithm (2.3.2D) of the previous section, which evaluates a function of a closely related type. (See exercise 3.) The same idea is used in many interpretive routines in connection with the evaluation of arithmetic expressions in postfix notation; we will return to this topic in Chapter 8. See also exercise 18, which gives another important procedure similar to Algorithm F.

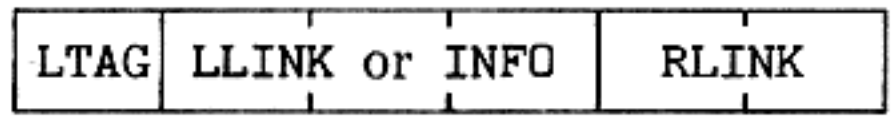
We have now seen that sequential representation of trees and forests is possible and meaningful in any of the three basic orders for the tree nodes: preorder, postorder, and (in reverse) endorder. There are also a number of linked forms of representation, which we will now consider.

The first idea is related to the transformation that takes (6) into (7): we remove the `INFO` fields from all nonterminal nodes and put this information as a new terminal node below the previous node. For example, the trees (2) would become



(10)

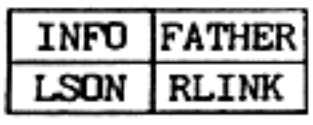
This new form shows that we may assume (without loss of generality) that all INFO in a tree structure appears in its terminal nodes. Therefore in the natural binary tree representation of Section 2.3.2, the LLINK and INFO fields are mutually exclusive and they can share the same field in each node. A node might have the fields

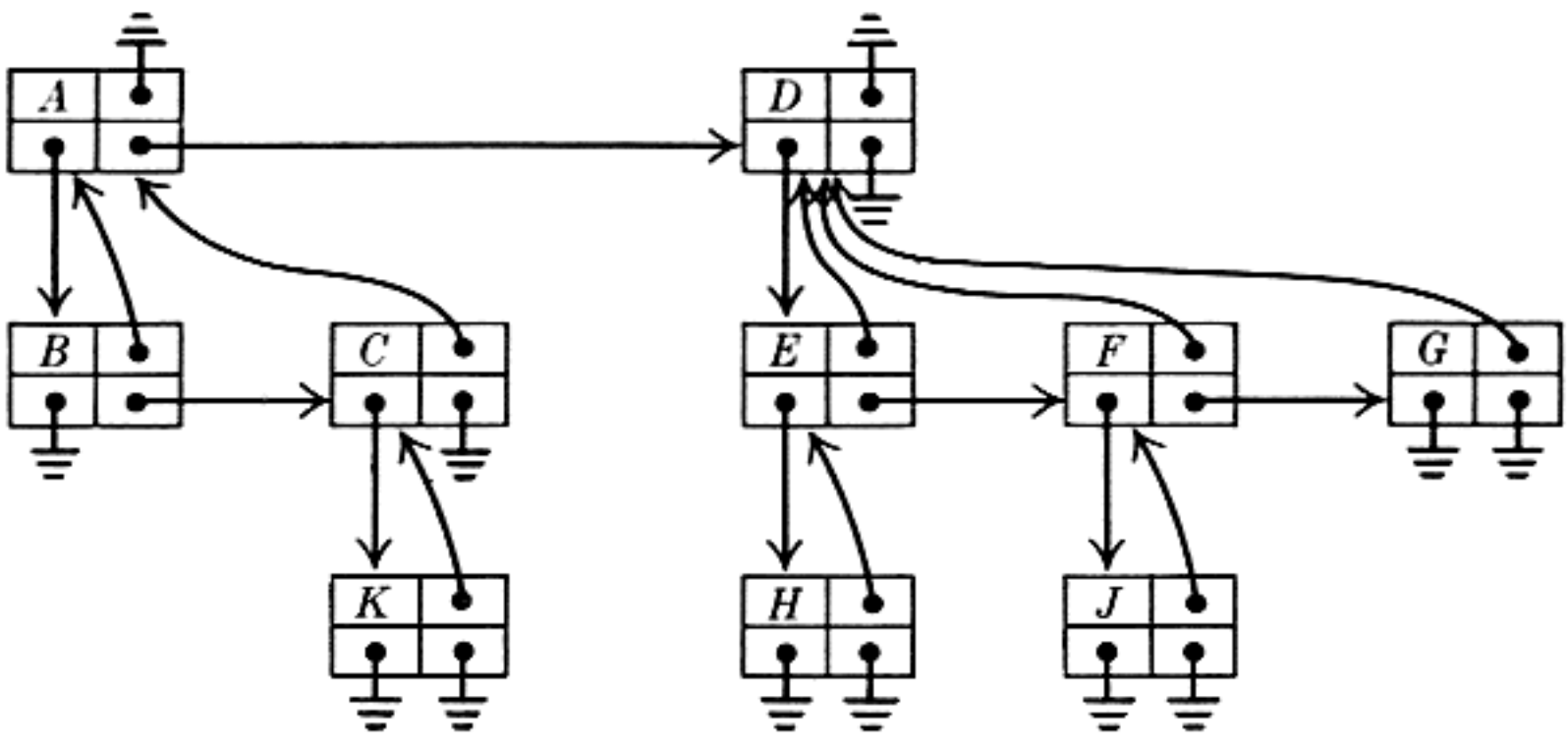


where the sign LTAG tells whether the second field is a link or not. [Compare this representation with, for example, the two-word format of (10) in Section 2.3.2.] By cutting INFO down from 6 bytes to 3, we can fit each node into one word. However, note that there are now 15 nodes instead of 10; the forest (10) takes 15 words of memory while (2) takes 20, yet the latter has 60 bytes of INFO compared to 30 in the other. There is no real gain in memory space in (10) unless the excess INFO space was going to waste; the LLINKs replaced in (10) are removed at the expense of about the same number of new RLINKs in the added nodes. Precise details of the difference between the two representations are discussed in exercise 4.

In the standard binary tree representation of a tree, the LLINK field might be more accurately called the "LSON" field, since it points from a father node to his leftmost son. The leftmost son is usually the "youngest" of the sons in the tree, since it is easier to insert a node at the left of a family than at the right; so the abbreviation "LSON" may also be thought of as the "last son" or "least son."

Many applications of tree structures require rather frequent references upward in the tree as well as downward. A threaded tree gives us the ability to go upward, but not with great speed; occasionally, it is preferable to have a third link, FATHER, in each node. This leads to a *triply linked tree*, where each node has LSON, RLINK, and FATHER links. Figure 26 shows a triply linked tree representation of (2). For an example of the use of triply linked trees, see Section 2.4.





It is clear that the FATHER link all by itself is enough to specify any *oriented* tree (or forest) completely. For we can draw the diagram of the tree if we know all the upward links. Every node except the root has just one father, but there may be several sons; so it is simpler to give upward links than downward ones. Why then haven't we considered upward links much earlier in our discussion? The answer, of course, is that upward links by themselves are hardly adequate in most situations, since it is very difficult to tell quickly if a node is terminal or not, or to locate any of its sons, etc. There is, however, a very important application in which only upward links are sufficient: We now turn to a brief study of an elegant algorithm for dealing with equivalence relations, which is due to M. J. Fischer and B. A. Galler.

An *equivalence relation* "$\equiv$" is a relation between the elements of a set of objects S satisfying the following three properties for any objects x, y, and z (not necessarily distinct) in S:

i) If $x \equiv y$ and $y \equiv z$, then $x \equiv z$. (Transitivity.)
ii) If $x \equiv y$, then $y \equiv x$. (Symmetry.)
iii) $x \equiv x$. (Reflexivity.)

(Compare this with the definition of a "partial ordering" relation in Section 2.2.3; equivalence relations are quite different from partial orderings, in spite of the fact that two of the three defining properties are the same.) Examples of equivalence relations are the relation "$=$", the relation of congruence (modulo m) for integers, the relation of similarity between trees, as defined in Section 2.3.1, etc.

The equivalence problem is to read in pairs of equivalences and to determine later whether two particular elements can be proved equivalent or not on the basis of the given pairs. For example, suppose that S is the set $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and suppose that we are given the pairs

$$1 \equiv 5, \quad 6 \equiv 8, \quad 7 \equiv 2, \quad 9 \equiv 8, \quad 3 \equiv 7, \quad 4 \equiv 2, \quad 9 \equiv 3. \tag{11}$$

It follows that, for example, $2 \equiv 6$, since $2 \equiv 7 \equiv 3 \equiv 9 \equiv 8 \equiv 6$. But we cannot show that $1 \equiv 6$. In fact, the pairs (11) divide S into two classes

$$\{1, 5\} \quad \text{and} \quad \{2, 3, 4, 6, 7, 8, 9\}, \tag{12}$$

such that two elements are equivalent if and only if they belong to the same class. It is not difficult to prove that *any* equivalence relation partitions its set S into disjoint classes (called the "equivalence classes"), such that two elements are equivalent if and only if they belong to the same class.

Therefore to solve the equivalence problem it is a matter of keeping track of equivalence classes like (12). We may start with each element alone in its class, thus:

$$\{1\} \ \{2\} \ \{3\} \ \{4\} \ \{5\} \ \{6\} \ \{7\} \ \{8\} \ \{9\} \tag{13}$$

Now if we are given the relation $1 \equiv 5$, we put $\{1, 5\}$ together in a class. After processing the first three relations $1 \equiv 5$, $6 \equiv 8$, and $7 \equiv 2$, we will have changed (13) to

$$\{1, 5\} \quad \{2, 7\} \quad \{3\} \quad \{4\} \quad \{6, 8\} \quad \{9\}. \tag{14}$$

Now the pair $9 \equiv 8$ puts $\{6, 8, 9\}$ together, etc.

The problem is to find a good way to represent situations like (12), (13), and (14) within a computer so that we can efficiently perform the operations of merging classes together and of testing whether two given elements are in the same class. The algorithm below uses tree structures for this purpose: The elements of S become nodes of a forest, and two nodes are equivalent as a consequence of the pairs of equivalences read so far *if and only if they belong to the same tree.* This test is easy to make, since two elements are in the same tree if and only if they are below the same root element. Furthermore, it is easy to merge two trees together into one, by simply attaching one as a new subtree of the root of the other.

Algorithm E (*Process equivalence relations*). Let S be the set of numbers $\{1, 2, \ldots, n\}$, and let FATHER[1], FATHER[2], ..., FATHER[n] be integer variables. This algorithm inputs a set of relations such as (11) and adjusts the FATHER table to represent a set of trees, so that two elements are equivalent as a consequence of the given relations if and only if they belong to the same tree. (*Note:* In a more general situation, the elements of S would be symbolic names instead of simply the numbers from 1 to n; then a search routine, as in Chapter 6, would locate nodes corresponding to the elements of S, and FATHER would be a field in each node. The modifications for this more general case are straightforward.)

E1. [Initialize.] Set FATHER[k] $\leftarrow 0$ for $1 \leq k \leq n$. [This means all trees initially consist of a root alone, as in (13).]

E2. [Input new pair.] Get the next pair of equivalent elements "$j \equiv k$" from the input. If the input is exhausted, the algorithm terminates.

E3. [Find roots.] If FATHER[j] $\neq 0$, set $j \leftarrow$ FATHER[j] and repeat this step. If FATHER[k] $\neq 0$, set $k \leftarrow$ FATHER[k] and repeat this step. (After this operation, j and k have moved up to the roots of two trees which are to be made equivalent. The input relation $j \equiv k$ was redundant if and only if we now have $j = k$.)

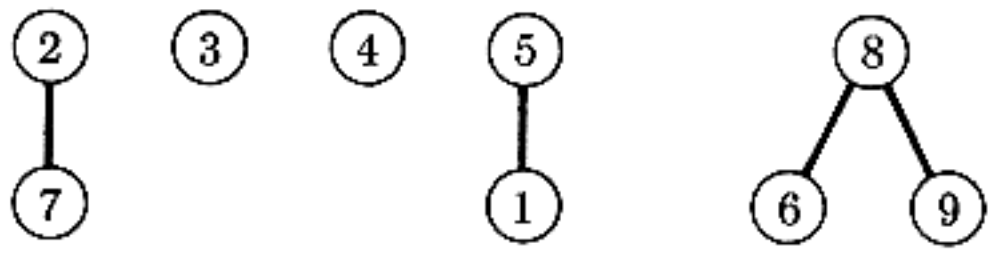
E4. [Merge trees.] If $j \neq k$, set FATHER[j] $\leftarrow k$. Go back to step E2. ∎

The reader should try this algorithm on the input (11). After processing $1 \equiv 5$, $6 \equiv 8$, $7 \equiv 2$, and $9 \equiv 8$, we will have

FATHER[k]:	5	0	0	0	0	8	2	0	8
k :	1	2	3	4	5	6	7	8	9

(15)

which represents the trees



(16)

After this point, the remaining relations of (11) cause some more interesting things to happen; see exercise 9.

This equivalence problem occurs in many applications. A more general version of the problem which arises when a compiler processes "equivalence declarations" in languages like FORTRAN is discussed in exercise 11.

There are still more ways to represent trees in computer memory. Recall that we discussed three principal methods for representing linear lists in Section 2.2: the "straight" representation with terminal link Λ, the "circularly" linked lists, and the "doubly" linked lists. The representation of unthreaded binary trees described in Section 2.3.1 corresponds to a "straight" representation in both LLINKs and RLINKs. It is possible to get eight other binary tree representations by independently using any of these three methods in the LLINK and RLINK directions. For example, Fig. 27 shows what we get if circular linking is used in both directions. If circular links are used throughout as in the figure, we have what is called a *ring structure;* ring structures have proved to be quite flexible in a number of applications. The proper choice of representation depends, as always, on the type of insertions, deletions, and traversals that are needed in the algorithms that manipulate these structures. A reader who has looked over the examples given so far in this chapter should have no difficulty understanding when to use and how to deal with any of these memory representations.

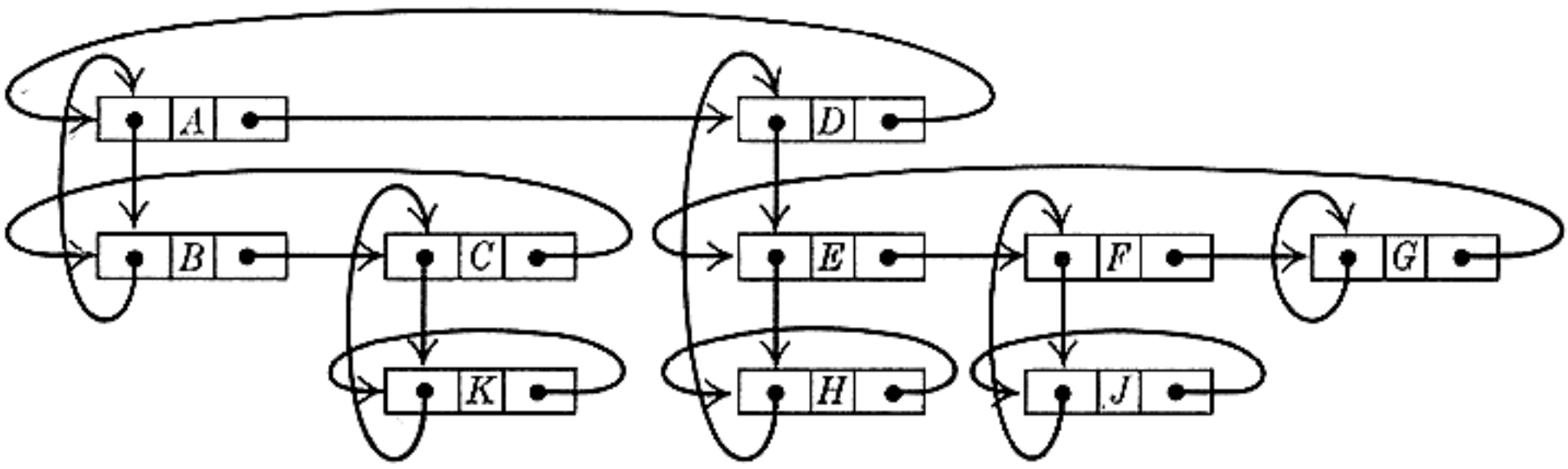


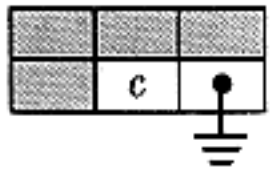
Fig. 27. A ring structure.

We close this section with an example of modified doubly linked ring structures applied to a problem we have considered before: arithmetic on polynomials. Algorithm 2.2.4A performs the addition of one polynomial to another, given that the two polynomials are expressed as circular lists, and various other algorithms in that section give other operations on polynomials; however, the polynomials are restricted to at most three variables. When multi-variable polynomials are

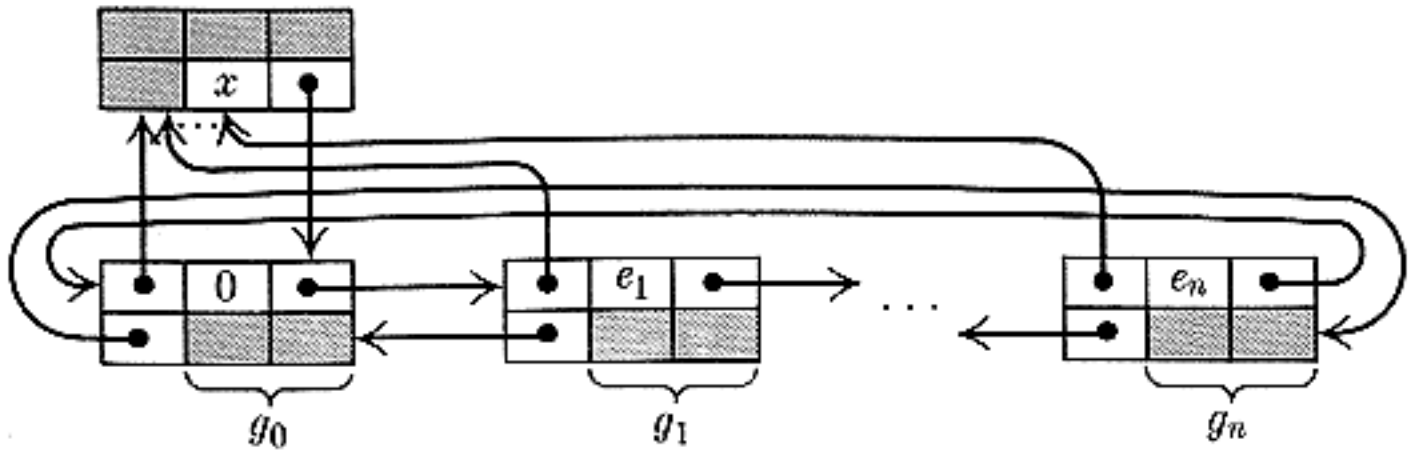
(a) Fields

UP	EXP	RIGHT
LEFT	CV	DOWN

(b) Polynomial $= c$ (constant)



(c) Polynomial $= g_0 + g_1 x^{e_1} + g_2 x^{e_2} + \cdots + g_n x^{e_n}$



(d) Example: $3 + x^2 + xyz + z^3 - 3xz^3$

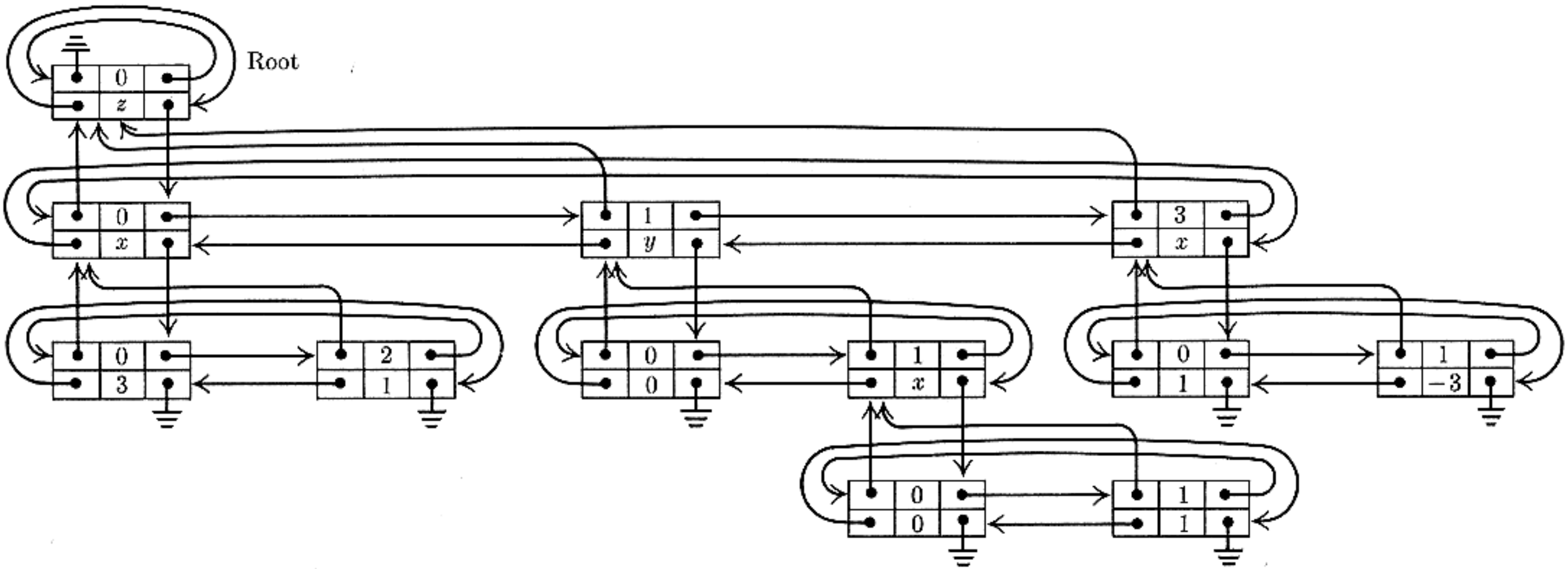
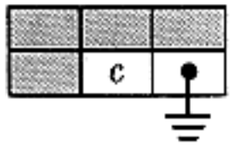


Fig. 28. Representation of polynomials using four-directional links. Shaded areas of nodes indicate information irrelevant in the context considered.

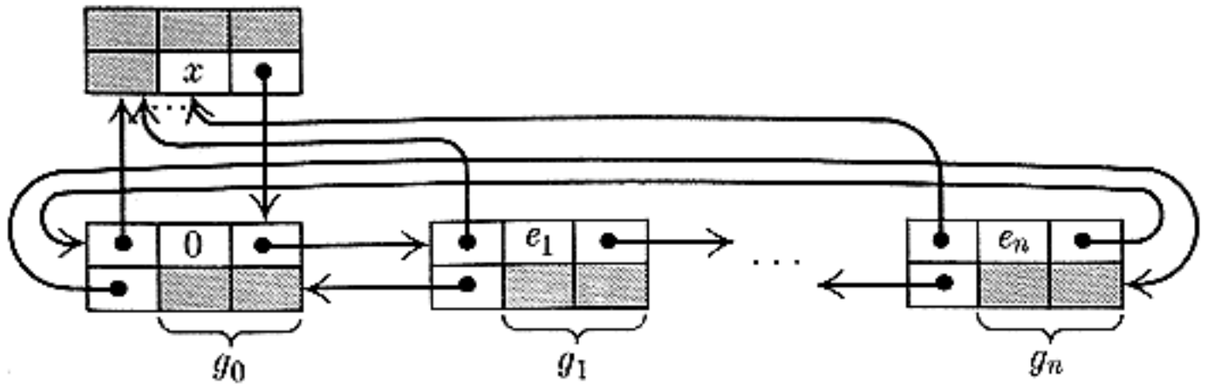
(a) Fields

UP	EXP	RIGHT
LEFT	CV	DOWN

(b) Polynomial $= c$ (constant)



(c) Polynomial $= g_0 + g_1 x^{e_1} + g_2 x^{e_2} + \cdots + g_n x^{e_n}$



(d) Example: $3 + x^2 + xyz + z^3 - 3xz^3$

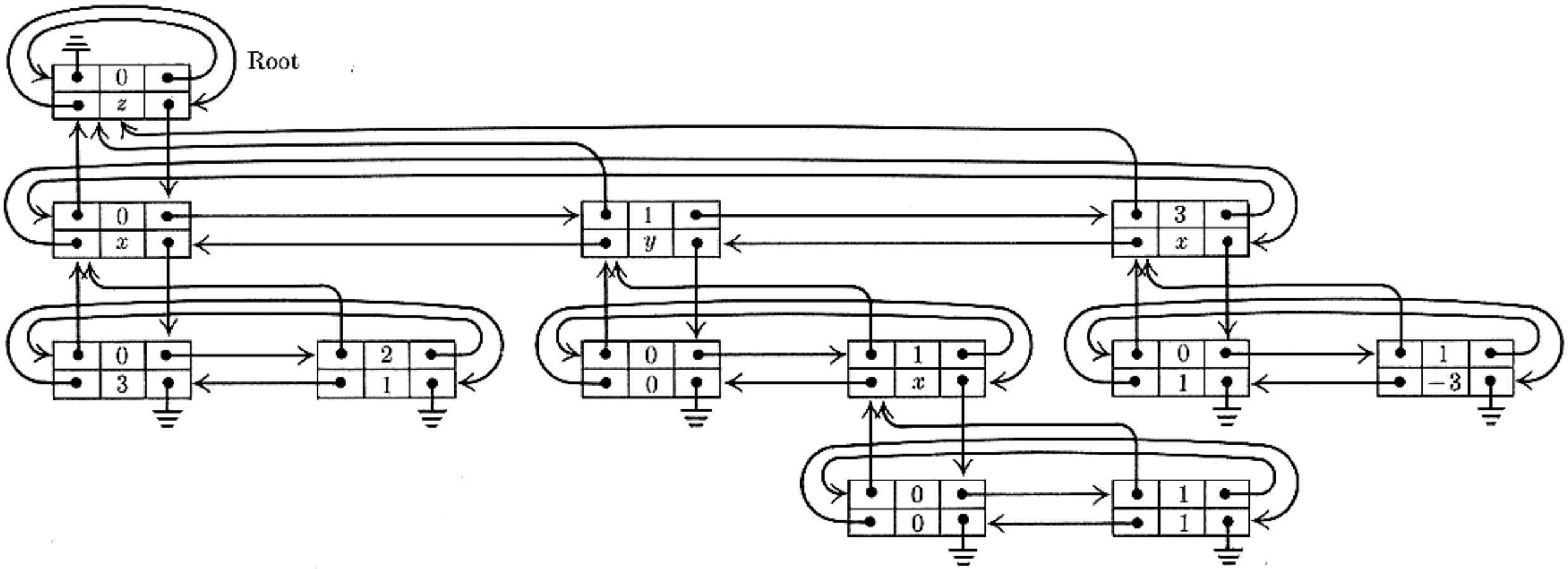


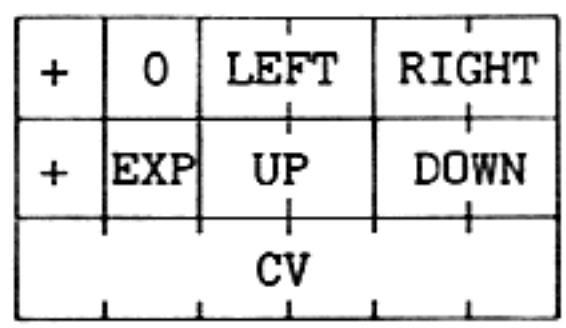
Fig. 28. Representation of polynomials using four-directional links. Shaded areas of nodes indicate information irrelevant in the context considered.

involved, it is often more appropriate to use a tree structure instead of a linear list.

A polynomial either is a constant or has the form

$$\sum_{0\le j\le n} g_j x^{e_j},$$

where x is a variable, $n > 0$, $0 = e_0 < e_1 < \cdots < e_n$, and $g_0, \ldots, g_n$ are polynomials involving only variables alphabetically less than x; $g_1, \ldots, g_n$ are not zero. This definition of polynomials lends itself to tree representation as indicated in Fig. 28. Nodes have six fields, which in the case of MIX might fit in three words:



(17)

Here LEFT, RIGHT, UP, and DOWN are links, EXP is an integer representing an exponent, and CV is either a constant (coefficient) or the alphabetic name of a variable.

The following algorithm illustrates traversal, insertion, and deletion in such a four-way-linked tree, so it bears careful study.

Algorithm A (*Addition of polynomials*). This algorithm adds polynomial(P) to polynomial(Q), assuming that P and Q are pointer variables which link to the roots of polynomial trees having the form shown in Fig. 28. At the conclusion of the algorithm, polynomial(P) will be unchanged, and polynomial(Q) will contain the sum.

A1. [Test type of polynomial.] If DOWN(P) = Λ (i.e., if P points to a constant), then set Q $\leftarrow$ DOWN(Q) zero or more times until DOWN(Q) = Λ and go to A3. If DOWN(P) $\neq \Lambda$, then if DOWN(Q) = Λ or if CV(Q) < CV(P), go to A2. Otherwise if CV(Q) = CV(P), set P $\leftarrow$ DOWN(P), Q $\leftarrow$ DOWN(Q) and repeat this step; if CV(Q) > CV(P), set Q $\leftarrow$ DOWN(Q) and repeat this step. (Step A1 either finds two matching terms of the polynomials or else determines that an insertion of a new variable must be made into this part of polynomial(Q).)

A2. [Downward insertion.] Set R $\Leftarrow$ AVAIL, S $\leftarrow$ DOWN(Q). If S $\neq \Lambda$, set UP(S) $\leftarrow$ R, S $\leftarrow$ RIGHT(S), and if EXP(S) $\neq 0$, repeat this operation until ultimately EXP(S) = 0. Set UP(R) $\leftarrow$ Q, DOWN(R) $\leftarrow$ DOWN(Q), LEFT(R) $\leftarrow$ R, RIGHT(R) $\leftarrow$ R, CV(R) $\leftarrow$ CV(Q), and EXP(R) $\leftarrow$ 0. Finally, set CV(Q) $\leftarrow$ CV(P) and DOWN(Q) $\leftarrow$ R, and return to A1. (We have inserted a zero polynomial just below NODE(Q), to obtain a match with a corresponding polynomial found within P's tree. The link manipulations done in this step

are straightforward and may be derived easily using "before-and-after" diagrams, as explained in Section 2.2.3.)

A3. [Match found.] (At this point, P and Q point to corresponding terms of the given polynomials, so addition is ready to proceed.) Set CV(Q) ← CV(Q) + CV(P). If this sum is zero and if EXP(Q) ≠ 0, go to step A8.

A4. [Advance to left.] (After successfully adding a term, we look for the next term to add.) Set P ← LEFT(P). If EXP(P) = 0, go to A6. Otherwise set Q ← LEFT(Q) one or more times until EXP(Q) ≤ EXP(P). If then EXP(Q) = EXP(P), return to step A1.

A5. [Insert to right.] Set R ⇐ AVAIL. Set UP(R) ← UP(Q), DOWN(R) ← Λ, LEFT(R) ← Q, RIGHT(R) ← RIGHT(Q), LEFT(RIGHT(R)) ← R, RIGHT(Q) ← R, EXP(R) ← EXP(P), CV(R) ← 0, and Q ← R. Return to step A1. (It was found necessary to insert a new term in the current row, just to the right of NODE(Q), in order to match a corresponding exponent in polynomial(P). As in step A2, a "before-and-after" diagram makes the above operations clear.)

A6. [Return upward.] (A row of polynomial(P) has now been completely traversed.) If UP(P) = Λ, the algorithm terminates; otherwise set P ← UP(P).

A7. [Move Q up to right level.] If UP(P) = Λ, the algorithm terminates; otherwise set Q ← UP(Q) zero or more times until CV(UP(Q)) = CV(UP(P)). Return to step A4.

A8. [Delete zero term.] Set R ← Q, Q ← RIGHT(Q), LEFT(Q) ← LEFT(R), and AVAIL ⇐ R. (Cancellation occurred, so a row element of polynomial(Q) is deleted.) If now EXP(LEFT(P)) = 0 and Q = LEFT(Q), go to A9; otherwise return to A4.

A9. [Delete constant polynomial.] (Cancellation has caused a polynomial to reduce to a constant, so a row of polynomial(Q) is deleted.) Set R ← Q, Q ← UP(Q), DOWN(Q) ← DOWN(R), CV(Q) ← CV(R), and AVAIL ⇐ R. Set S ← DOWN(Q); if S ≠ Λ, set UP(S) ← Q, S ← RIGHT(S), and if EXP(S) ≠ 0, repeat this operation until ultimately EXP(S) = 0.

A10. [Zero detected?] If DOWN(Q) = Λ, CV(Q) = 0, and EXP(Q) ≠ 0, set P ← UP(P) and go to A8; otherwise go to A6. ∎

This algorithm will actually run much faster than Algorithm 2.2.3A if polynomial(P) has few terms and polynomial(Q) has many, since it is not necessary to pass over all of polynomial(Q) during the addition process. The reader will find it instructive to simulate Algorithm A by hand, adding the polynomial $xy - x^2 - xyz - z^3 + 3xz^3$ to the polynomial shown in Fig. 28. (This case does not demonstrate the efficiency of the algorithm, but it makes the algorithm go through all of its paces by showing the difficult situations which must be handled.) For further commentary on Algorithm A, see exercises 12 and 13.

No claim is being made here that the representation shown in Fig. 28 is the "best" for polynomials in several variables; in Chapter 8 we will consider another format for polynomial representation, together with arithmetic algorithms using an auxiliary stack, which have significant advantages of conceptual simplicity when compared to Algorithm A. Our main interest in Algorithm A is the way it typifies manipulations on trees with many links.

EXERCISES

▶ **1.** [*20*] Show that for any family-order sequential representation, as in (6), the LLINK arrows never cross each other.

2. [*22*] (Burks, Warren, and Wright, *Math. Comp.* **8** (1954), 46–50.) The trees (2) stored in *preorder* with degrees would be

DEGREE	2	0	1	0	3	1	0	1	0	0
INFO	*A*	*B*	*C*	*K*	*D*	*E*	*H*	*F*	*J*	*G*

[cf. (9) where postorder was used]. Design an algorithm analogous to Algorithm F to evaluate a locally defined function of the nodes by going from right to left in this representation.

▶ **3.** [*24*] Modify Algorithm 2.3.2D so that it follows the ideas of Algorithm F, placing the derivatives it computes as intermediate results on a stack, instead of recording their locations in an anomalous fashion as is done in step D3. (Cf. exercise 2.3.2–21.) The stack may be maintained by using the RLINK field in the root of each derivative.

4. [*18*] The trees (2) contain 10 nodes, five of which are terminal. Representation of these trees in the normal binary-tree fashion involves 10 LLINK fields and 10 RLINK fields (one for each node). Representation of these trees in the form (10), where LLINK and INFO share the same space in a node, requires 5 LLINKs and 15 RLINKs. There are 10 INFO fields in each case.

Given a forest with n nodes, m of which are terminal, compare the total number of LLINKs and RLINKs that must be stored using these two methods of tree representation.

5. [*16*] A triply linked tree, as shown in Fig. 26, contains FATHER, LSON, and RLINK fields in each node, with liberal use of Λ-links when there is no appropriate node to mention in the FATHER, LSON, or RLINK field. Would it be a good idea to extend this representation to a *threaded* tree, by putting "thread" links in place of the null LSON and RLINK entries, as we did in Section 2.3.1?

▶ **6.** [*24*] Suppose that the nodes of an *oriented* forest have three link fields, FATHER, LSON, and RLINK, but only the FATHER link has been set up to indicate the tree structure. The LSON field of each node is Λ and the RLINK fields are set as a linear list which simply links the nodes together in some order. The link variable FIRST points to the first node, and the last node has RLINK = Λ.

Design an algorithm which goes through these nodes and fills in the LSON and RLINK fields compatible with the FATHER links, so that a triply linked tree representation like that in Fig. 26 is obtained. Also, reset FIRST so that it now points to the root of the first tree in this representation.

7. [*15*] What classes would appear in (12) if the relation $9 \equiv 3$ had not been given in (11)?

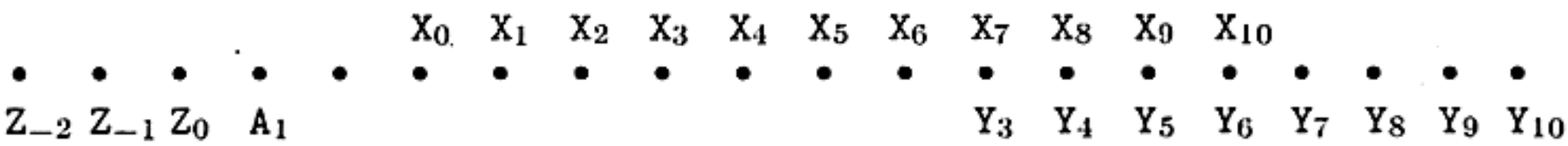
8. [*20*] Algorithm E sets up a tree structure which represents the given pairs of equivalences, but the text does not mention explicitly how the result of Algorithm E can be used. Design an algorithm which answers the question, "Is $j \equiv k$?", assuming that $1 \le j \le n$, $1 \le k \le n$, and that Algorithm E has set up the FATHER table for some set of equivalences.

9. [*21*] Give a table analogous to (15) and a diagram analogous to (16) which shows the trees present after Algorithm E has processed all of the pairs of equivalences in (11) from left to right.

10. [*M48*] Perform an analysis of the efficiency of Algorithm E when it is given random pairs of equivalences in random order. In particular, what is the average level of the nodes in the trees, after Algorithm E has been in operation?

▸ **11.** [*24*] ("Equivalence declarations.") Several compiler languages provide a facility for overlapping the memory locations assigned to sequentially stored tables. The programmer gives the compiler pairs of relations of the form "X[j] $\equiv$ Y[k]" which means variable X[$j+s$] is to be assigned to the same location as variable Y[$k+s$] for all s. Each variable is further given a range of allowable subscripts: "ARRAY X[ℓ:u]" means that space is to be set aside in memory for the table entries X[ℓ], X[$\ell+1$], ..., X[u]. For each equivalence class of variables, the compiler reserves as small a block of consecutive memory locations as possible, to contain all the table entries for the allowable subscript values of these variables.

For example, suppose we have ARRAY X[0:10], ARRAY Y[3:10], ARRAY A[1:1], and ARRAY Z[−2:0], plus the equivalences X[7] $\equiv$ Y[3], Z[0] $\equiv$ A[0], and Y[1] $\equiv$ A[8]. We must set aside 20 consecutive locations



for these variables. Note that the location following A[1] is not an "allowable" subscript value for any of the arrays.

The object of this exercise is to modify Algorithm E so that it applies to the more general situation just described. Assume that we are writing a compiler for such a language, and the tables inside our compiler program itself have one node for each array, containing the fields NAME, FATHER, DELTA, LBD, and UBD. Assume that the compiler program has previously processed all the ARRAY declarations, so that if "ARRAY X[ℓ:u]" has appeared and if P points to the node for X, then

$$\texttt{NAME(P)} = \text{"X"}, \quad \texttt{FATHER(P)} = \Lambda, \quad \texttt{DELTA(P)} = 0,$$
$$\texttt{LBD(P)} = \ell, \quad \texttt{UBD(P)} = u.$$

The problem is to design an algorithm which processes the equivalence declarations, so that after this algorithm has been performed

FATHER(P) = Λ means that locations X[LBD(P)], ..., X[UBD(P)] are to be reserved in memory for this equivalence class;

FATHER(P) = Q $\neq \Lambda$ means location X[k] equals location Y[k + DELTA(P)], where NAME(Q) = "Y".

For example, before the equivalences listed above we might have the nodes

P	NAME(P)	FATHER(P)	DELTA(P)	LBD(P)	UBD(P)
α	X	Λ	0	0	10
β	Y	Λ	0	3	10
γ	A	Λ	0	1	1
δ	Z	Λ	0	−2	0

After the equivalences are processed, the nodes might appear thus:

α	X	Λ	*	−5	14
β	Y	α	4	*	*
γ	A	δ	0	*	*
δ	Z	α	−3	*	*

("*" denotes irrelevant information).

Design an algorithm which makes this transformation. Assume that inputs to your algorithm have the form (P, j, Q, k), denoting "X[j] $\equiv$ Y[k]", where NAME(P) = "X" and NAME(Q) = "Y". Be sure to check whether the equivalences are contradictory; e.g., X[1] $\equiv$ Y[2] contradicts X[2] $\equiv$ Y[1].

12. [*21*] At the beginning of Algorithm A, the variables P and Q point to the roots of two trees. Let P_0 and Q_0 denote the values of P and Q before execution of Algorithm A. (a) After the algorithm terminates, is Q_0 always the address of the root of the sum of the two given polynomials? (b) After the algorithm terminates, have P and Q returned to their original values P_0, Q_0?

▸ **13.** [*M29*] Give an informal proof that at the beginning of step A8 of Algorithm A we always have EXP(P) = EXP(Q) and CV(UP(P)) = CV(UP(Q)). (This fact is important to the proper understanding of that algorithm.)

14. [*M32*] Give a formal proof (or disproof) of the validity of Algorithm A.

15. [*40*] Design an algorithm to compute the product of two polynomials represented as in Fig. 28.

▸ **16.** [*28*] Design an algorithm which, given tables INFO1[j], RLINK[j] for $1 \leq j \leq n$, corresponding to preorder sequential representation, forms tables INFO2[j], DEGREE[j] for $1 \leq j \leq n$, corresponding to postorder sequential representation. For example, your algorithm should transform

j	1	2	3	4	5	6	7	8	9	10
INFO1[j]	A	B	C	K	D	E	H	F	J	G
RLINK[j]	5	3	0	0	0	8	0	10	0	0

[cf. (3)] into

INFO2[j]	B	K	C	A	H	E	J	F	G	D
DEGREE[j]	0	0	1	2	0	1	0	1	0	3

[cf. (9)].

17. [*M24*] Prove the validity of Algorithm F.

18. [*25*] Algorithm F evaluates a "bottom-up" locally-defined function, namely, one which should be evaluated at the sons of a node before it is evaluated at the node. A "top-down" locally-defined function f is one in which the value of f at a node x depends only on x and the value of f at the *father* of x. Using an auxiliary stack, design an algorithm analogous to Algorithm F which evaluates a "top-down" function f at each node of a tree. (Like Algorithm F, your algorithm should work efficiently on trees which have been stored in *postorder* with degrees, as in (9).)

*2.3.4. Basic Mathematical Properties of Trees

Tree structures have been the object of extensive mathematical investigations for many years, long before the advent of computers, and many interesting facts have been discovered about them. In this section we will survey the mathematical theory of trees, which not only gives us more insight into the nature of tree structures but also has important applications to computer algorithms.

The material which follows comes mostly from a larger area of mathematics known as the theory of graphs. Unfortunately, there is as yet no standard terminology in this field, and so the author has followed the usual practice of contemporary books on graph theory, namely to use words that are similar but not identical to the terms used in all *other* books on graph theory. An attempt has been made in the following subsections (and, indeed, throughout this book) to choose short, descriptive words for the important concepts, selected from those which are in reasonably common use and which do not sharply conflict with other common terminology. The nomenclature used here is also biased towards computer applications; thus, an electrical engineer may prefer to call a "tree" what we call a "free tree," but we want the shorter term "tree" to stand for the concept which is generally used in the computer literature and which is so much more important in computer applications. If we were to follow the terminology of some authors on graph theory, we would have to say "finite labeled rooted ordered tree" instead of just "tree," and "topological bifurcating arborescence" instead of "binary tree"!

2.3.4.1. Free trees. A *graph* is generally defined to be a set of points (called *vertices*) together with a set of lines (called *edges*) joining certain pairs of distinct vertices. There is at most one edge joining any pair of vertices. Two vertices are called *adjacent* if there is an edge joining them. If V and V' are vertices and if $n \geq 0$, we say that $(V_0, V_1, \ldots, V_n)$ is a *path* of length n from V to V' if $V = V_0$, V_k is adjacent to V_{k+1} for $0 \leq k < n$, and $V_n = V'$. The path is *simple* if $V_0, V_1, \ldots, V_{n-1}$ are distinct and if $V_1, \ldots, V_{n-1}, V_n$ are distinct. A graph is *connected* if there is a path between any two vertices of the graph. A *cycle* is a simple path of length three or more from a vertex to itself.

These definitions are illustrated in Fig. 29, which shows a connected graph with five vertices and six edges. Vertex C is adjacent to A but not to B; there are two paths of length two from B to C, namely (B, A, C) and (B, D, C). There are several cycles, including (B, D, E, B).

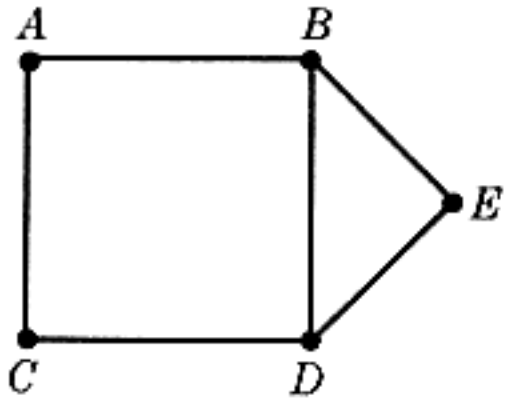


Fig. 29. A graph.

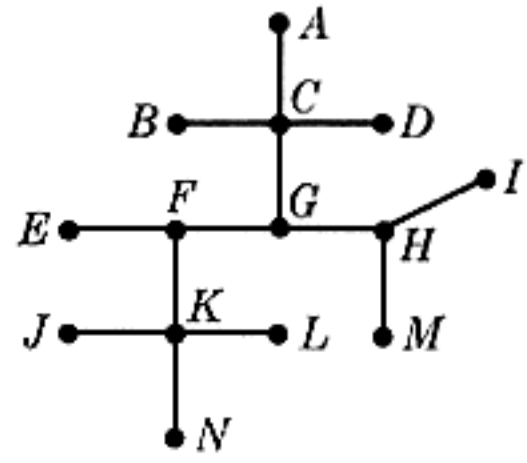


Fig. 30. A free tree.

A *free tree* or "unrooted tree" (Fig. 30) is defined to be a connected graph with no cycles. This definition applies to infinite graphs as well as finite ones, although for computer applications we naturally are most concerned with finite trees. There are many equivalent ways to define a free tree; some of these appear in the following well-known theorem:

Theorem A. *If G is a graph, the following statements are equivalent:*

a) *G is a free tree;*

b) *G is connected, but if any edge is deleted, the resulting graph is no longer connected.*

c) *If V and V' are distinct vertices of G, there is exactly one simple path from V to V'.*

Furthermore, if G is finite, containing exactly $n > 0$ vertices, the following properties are also equivalent to (a), (b), and (c):

d) *G contains no cycles and has $n - 1$ edges.*

e) *G is connected and has $n - 1$ edges.*

Proof. (a) implies (b), for if the edge VV' is deleted but G is still connected, there must be a simple path $(V, V_1, \ldots, V')$—see exercise 2—and then $(V, V_1, \ldots, V', V)$ would be a cycle in G.

(b) implies (c), for there is at least one simple path from V to V'. And if there were two such paths $(V, V_1, \ldots, V')$ and $(V, V_1', \ldots, V')$, we could find the smallest k for which $V_k \neq V_k'$; deleting the edge $V_{k-1}V_k$ would not disconnect the graph, since there would still be a path $(V_{k-1}, \ldots, V, V_1', \ldots, V', \ldots, V_k)$ from V_{k-1} to V_k which does not use the deleted edge.

(c) implies (a), for if G contains a cycle $(V, V_1, \ldots, V)$, there are two simple paths from V to V_1.

To show that (d) and (e) are also equivalent to (a), (b), and (c), let us first prove an auxiliary result: If G is any finite graph which has no cycles and at least one edge, then there is at least one vertex which is adjacent to exactly one other vertex. For we take an arbitrary vertex V_1 and an adjacent vertex V_2; for $k \geq 2$ either V_k is adjacent to V_{k-1} and no other, or it is adjacent to a vertex which we may call $V_{k+1} \neq V_{k-1}$. Since there are no cycles, $V_1, V_2, \ldots, V_{k+1}$ must be distinct vertices, so this process must ultimately terminate.

Now assume G is a tree with $n > 1$ vertices, and let V_n be a vertex which is adjacent to only one other vertex, namely V_{n-1}. If we delete V_n and the edge $V_{n-1}V_n$, the remaining graph G' is a tree, since V_n appears in no simple path of G except as the first or the last element. This argument proves (by induction on n) that G has $n - 1$ edges, i.e., (a) implies (d).

Assume that G satisfies (d) and let V_n, V_{n-1}, G' be as in the preceding paragraph. Then the graph G is connected, since V_n is connected to V_{n-1} which (by induction on n) is connected to all other vertices of G'. Thus (d) implies (e).

Finally assume that G satisfies (e). If G contains a cycle, we can delete any edge appearing in that cycle and G would still be connected. We can therefore continue deleting edges in this way until we obtain a connected graph G' with $n - 1 - k$ edges and no cycles. But since (a) implies (d), we must have $k = 0$, that is, $G = G'$. ∎

The idea of a free tree can be applied directly to the analysis of computer algorithms. In Section 1.3.3, we discussed the application of Kirchhoff's law to the problem of counting the number of times each step of an algorithm is performed; we found that Kirchhoff's law does not completely determine the number of times each step is executed, but it reduces the number of unknowns that must be specially interpreted. The theory of trees tells us how many independent unknowns will remain, and it gives us a systematic way to find them.

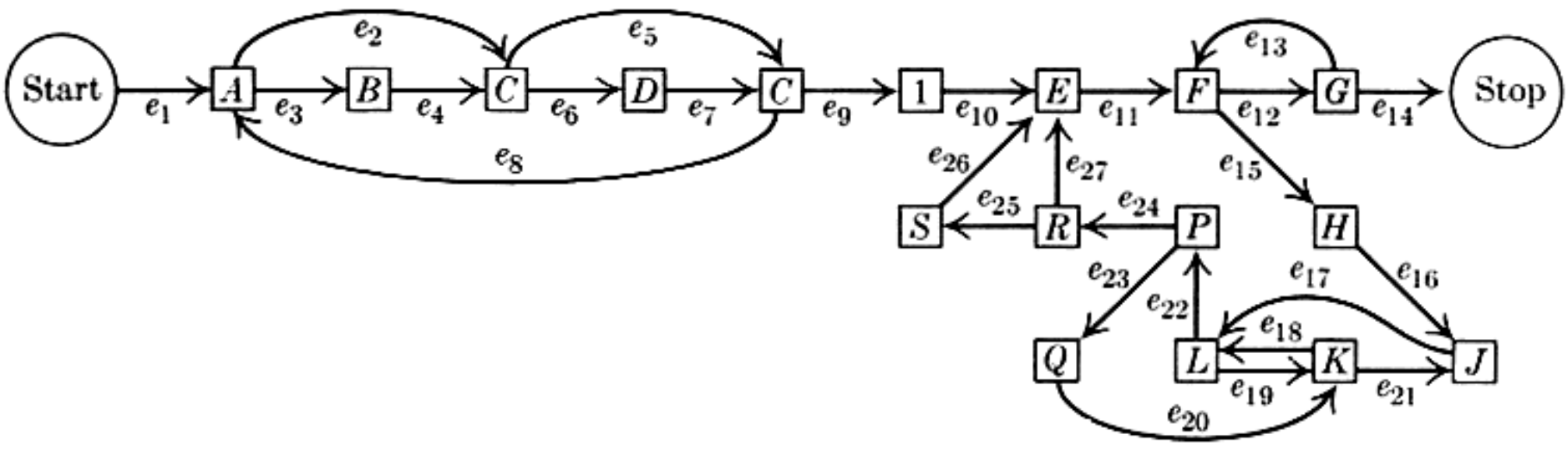


Fig. 31. Abstracted flow chart of Program 1.3.3A.

It is easier to understand the method which follows if an example is studied, so we will work an example as the theory is being developed. Figure 31 shows an abstracted flow chart for Program 1.3.3A, which was subjected to a "Kirchhoff's law" analysis in Section 1.3.3. Each box in Fig. 31 represents part of the computation, and the letter or number inside the box denotes the number of times that computation will be performed during one run of the program, using the notation of Section 1.3.3. An arrow between boxes represents a possible jump in the program. The arrows have been labeled $e_1, e_2, \ldots, e_{27}$. Our goal is to find all relations between the quantities A, B, C, D, E, F, G, H, J, K, L, P, Q, R, and S that are implied by Kirchhoff's law, and at the same time we hope to gain some insight into the general problem. (*Note:* Some simplifications have

already been made in Fig. 31, e.g., the box between C and E has been labeled "1", and this is in fact a consequence of Kirchhoff's law.)

Let E_j denote the number of times branch e_j is taken during the execution of the program being studied; Kirchhoff's law is

"sum of E's into box = value in box = sum of E's leaving box"; (1)

e.g., in the case of the box marked K we have

$$E_{19} + E_{20} \quad = \quad K \quad = \quad E_{18} + E_{21}. \tag{2}$$

In the discussion which follows, we will regard $E_1, E_2, \ldots, E_{27}$ as the unknowns, instead of $A, B, \ldots, S$.

The flow chart in Fig. 31 may be further abstracted so that it becomes a graph G as in Fig. 32. The boxes have shrunk to vertices, and the arrows $e_1, e_2, \ldots$ now represent edges of the graph. (A graph, strictly speaking, has no implied direction in its edges, and when we refer to graph-theoretical properties of G, the direction of the arrows should be ignored. The application to Kirchhoff's law, however, makes use of the arrows, as we will see later.) For convenience an extra edge e_0 has been drawn from the "stop" vertex to the "start" vertex, so that Kirchhoff's law applies uniformly to all parts of the graph. Figure 32 also includes some other minor changes from Fig. 31: an extra vertex and edge have been added to divide e_{13} into two parts e'_{13} and e''_{13}, so that the basic definition of a graph (no two edges join the same two vertices) is valid; e_{19} has also been split up in this way. A similar modification would have been made if we had any vertex with an arrow leading back to itself.

Some of the edges in Fig. 32 have been drawn much heavier than the others. These edges form a *free subtree* of the graph, connecting all the vertices. It is

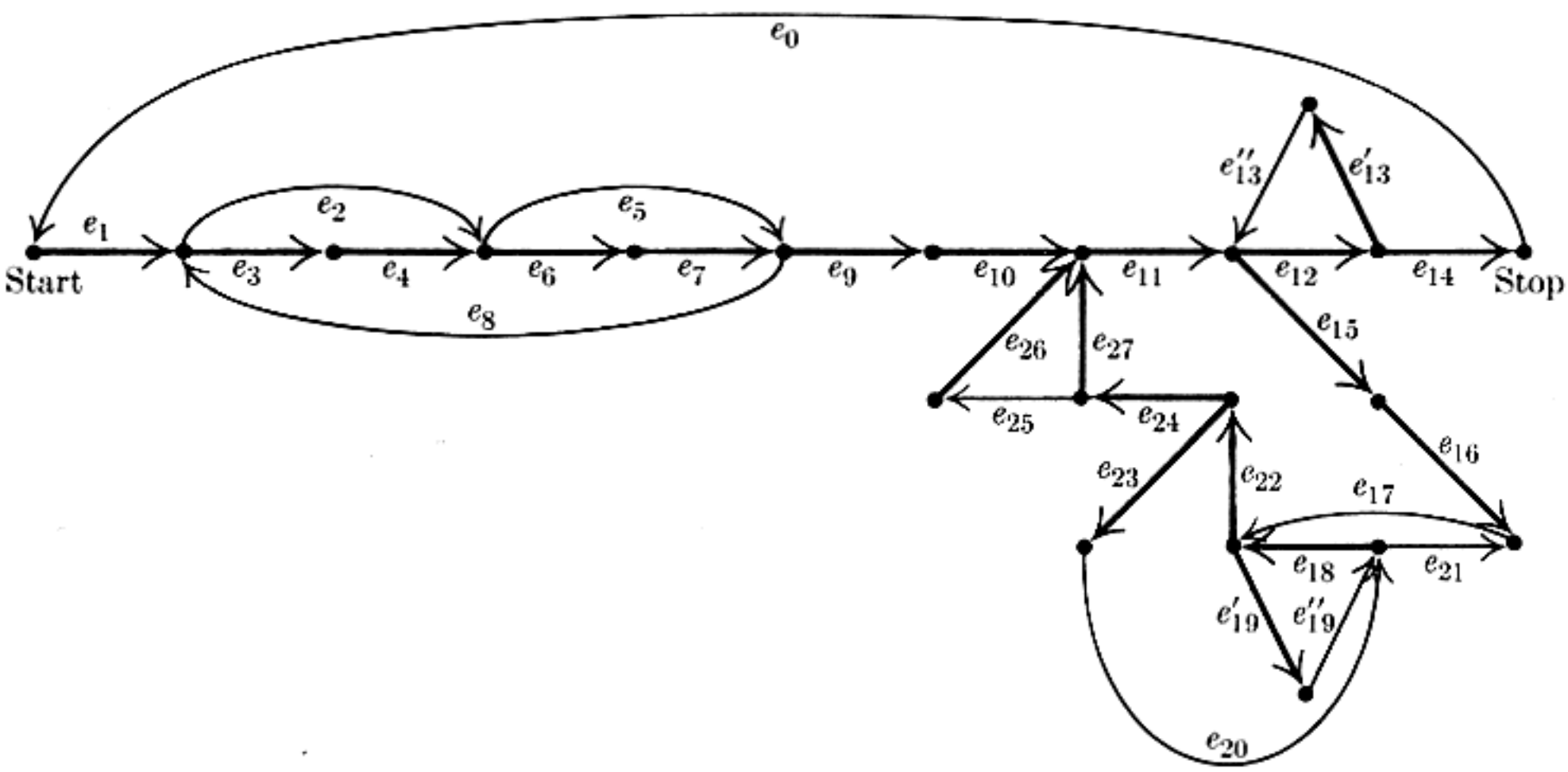


Fig. 32. Graph corresponding to Fig. 31, including a free subtree.

always possible to find a free subtree of the graphs arising from flow charts, because the graphs must be connected and, by part (b) of Theorem A, if G is connected and not a free tree, we can delete some edge and still have the resulting graph connected; this process can be iterated until we reach a subtree. Another algorithm for finding a free subtree appears in exercise 9. We can in fact always discard the edge e_0 (which went from the "stop" to the "start" vertex) first, so that we may assume e_0 does not appear in the subtree chosen.

Let G' be a free subtree of the graph G found in this way, and consider any edge VV' of G that is *not* in G'. We may now note an important consequence of Theorem A: G' plus this new edge VV' contains a cycle, and in fact there is *exactly one* cycle having the form $(V, V', \ldots, V)$, since there is a unique simple path from V' to V in G'. For example, if G' is the free subtree shown in Fig. 32, and if we add the edge e_2, we obtain a cycle which goes along e_2 and then (in the direction opposite to the arrows) along e_4 and e_3. This cycle may be written algebraically as "$e_2 - e_3 - e_4$", using plus signs and minus signs to indicate whether the cycle goes in the direction of the arrows or not.

If we carry out this process for each edge not in the free subtree, we obtain the so-called *fundamental cycles*, which in the case of Fig. 32 are

$$
\begin{aligned}
C_0&: \quad e_0 + e_1 + e_3 + e_4 + e_6 + e_7 + e_9 + e_{10} + e_{11} + e_{12} + e_{14},\\
C_2&: \quad e_2 - e_3 - e_4,\\
C_5&: \quad e_5 - e_6 - e_7,\\
C_8&: \quad e_8 + e_3 + e_4 + e_6 + e_7,\\
C''_{13}&: \quad e''_{13} + e_{12} + e'_{13},\\
C_{17}&: \quad e_{17} + e_{22} + e_{24} + e_{27} + e_{11} + e_{15} + e_{16},\\
C''_{19}&: \quad e''_{19} + e_{18} + e'_{19},\\
C_{20}&: \quad e_{20} + e_{18} + e_{22} + e_{23},\\
C_{21}&: \quad e_{21} - e_{16} - e_{15} - e_{11} - e_{27} - e_{24} - e_{22} - e_{18},\\
C_{25}&: \quad e_{25} + e_{26} - e_{27}.
\end{aligned}
\tag{3}
$$

Obviously an edge e_j which is not in the free subtree will appear in only one of the fundamental cycles, namely C_j.

We are now approaching the climax of this construction. Each fundamental cycle represents a solution to Kirchhoff's equations; for example, the solution corresponding to C_2 is to let $E_2 = +1$, $E_3 = -1$, $E_4 = -1$, and all other E's $= 0$. It is clear that flow around a cycle in a graph always satisfies the condition (1) of Kirchhoff's law. Moreover, Kirchhoff's equations are "homogeneous," so the sum or difference of solutions to (1) yields another solution. Therefore we may conclude that the values of $E_0, E_2, E_5, \ldots, E_{25}$ are *independent* in the following sense:

If $x_0, x_2, \ldots, x_{25}$ are any real numbers (one x_j for each e_j not in the free subtree G'), *there is a solution to Kirchhoff's equations* (1) *such that $E_0 = x_0$, $E_2 = x_2$, $\ldots$, $E_{25} = x_{25}$.* (4)

Such a solution is found by going x_0 times around cycle C_0, x_2 times around cycle C_2, etc. Furthermore, we find that the values of the remaining variables $E_1, E_3, E_4, \ldots$ are completely *dependent* on the values $E_0, E_2, \ldots, E_{25}$:

The solution mentioned in statement (4) *is unique.* (5)

For if there are two solutions to Kirchhoff's equations such that $E_0 = x_0, \ldots, E_{25} = x_{25}$, we can subtract one from the other and we thereby obtain a solution in which $E_0 = E_2 = E_5 = \cdots = E_{25} = 0$. But now *all* E_j must be zero, for it is easy to see that a nonzero solution to Kirchhoff's equations is impossible when the graph is a free tree (see exercise 4). Therefore the two assumed solutions must be identical. We have now proved that all solutions of Kirchhoff's equations may be obtained as sums of multiples of the fundamental cycles.

When these remarks are applied to the graph in Fig. 32, we obtain the following general solution of Kirchhoff's equations in terms of the independent variables $E_0, E_2, \ldots, E_{25}$:

$$
\begin{aligned}
E_1 &= E_0, & E_{14} &= E_0,\\
E_3 &= E_0 - E_2 + E_8, & E_{15} &= E_{17} - E_{21},\\
E_4 &= E_0 - E_2 + E_8, & E_{16} &= E_{17} - E_{21},\\
E_6 &= E_0 - E_5 + E_8, & E_{18} &= E''_{19} + E_{20} - E_{21},\\
E_7 &= E_0 - E_5 + E_8, & E'_{19} &= E''_{19},\\
E_9 &= E_0, & E_{22} &= E_{17} + E_{20} - E_{21},\\
E_{10} &= E_0, & E_{23} &= E_{20},\\
E_{11} &= E_0 + E_{17} - E_{21}, & E_{24} &= E_{17} - E_{21},\\
E_{12} &= E_0 + E''_{13}, & E_{26} &= E_{25},\\
E'_{13} &= E''_{13}, & E_{27} &= E_{17} - E_{21} - E_{25}.
\end{aligned}
\qquad (6)
$$

To obtain these equations, we merely list, for each edge e_j in the subtree, all E_k for which e_j appears in cycle C_k, with the appropriate sign. [Thus, the matrix of coefficients in (6) is just the transpose of the matrix of coefficients in (3).]

Strictly speaking, C_0 should not be called a fundamental cycle, since it involves the special edge e_0. We may call C_0 minus the edge e_0 a *fundamental path from "start" to "stop."* Our boundary condition, that the "start" and "stop" boxes in the flow chart are performed exactly once, is equivalent to the relation

$$E_0 = 1. \qquad (7)$$

The preceding discussion shows how to obtain all solutions to Kirchhoff's law; the same method may be applied (as Kirchhoff himself applied it) to electrical circuits instead of program flow charts. It is natural to ask at this point whether Kirchhoff's law is the strongest possible set of equations that can be given for the case of program flow charts, or whether more can be said: Any execution of a computer program that goes from "start" to "stop" gives us a set of values $E_1, E_2, \ldots, E_{27}$ for the number of times each edge is traversed, and these values obey Kirchoff's law; but are there solutions to Kirchhoff's

equations which do not correspond to any computer program execution? (In this question, we do not assume any information about the computer program as given to us, except its flow chart.) If there are solutions which meet Kirchhoff's conditions but do not correspond to actual program execution, we can give stronger conditions than Kirchhoff's law. For the case of electrical circuits Kirchhoff himself gave a second law: the sum of the voltage drops around a fundamental cycle must be zero. This second law does not apply to our problem.

There is indeed an obvious further condition that the E's must satisfy, if they are to correspond to some actual path in the flow chart from "start" to "stop"; they must be integers, and in fact they must be *nonnegative integers.* This is not a trivial condition, for we may not simply assign arbitrary nonnegative integer values to the independent variables $E_2, E_5, \ldots, E_{25}$; for example, if we take $E_2 = 2$ and $E_8 = 0$, we find from (6), (7) that $E_3 = -1$. (Thus, no execution of the flow chart in Fig. 31 will take branch e_2 twice without taking branch e_8 at least once.) The condition that all the E's be nonnegative integers is not enough either; for example, consider the solution in which $E''_{19} = 1$, $E_2 = E_5 = \cdots = E_{17} = E_{20} = E_{21} = E_{25} = 0$; there is no way to get to e_{18} except via e_{15}. The following condition is a necessary and sufficient condition which answers the problem raised in the previous paragraph: Let $E_2, E_5, \ldots, E_{25}$ be any given values, and determine $E_1, E_3, \ldots, E_{27}$ according to (6), (7). Assume that all the E's are nonnegative integers, and assume that the graph whose edges are those e_j for which $E_j > 0$, and whose vertices are those which touch such e_j, is *connected.* Then there is a path from "start" to "stop" in which edge e_j is traversed exactly E_j times. This fact is proved in the next section (see exercise 2.3.4.2–24).

Let us now summarize the preceding discussion:

Theorem K. *If a flow chart (such as Fig. 31) contains n boxes (including "start" and "stop") and m arrows, it is possible to find $m - n + 1$ fundamental cycles and a fundamental path from "start" to "stop", such that any path from "start" to "stop" is equivalent (in terms of the number of times each edge is traversed) to one traversal of the fundamental path plus a uniquely determined number of traversals of each of these fundamental cycles.* (The fundamental path and fundamental cycles may include some edges which are to be traversed in a direction *opposite* that shown by the arrow on the edge; we conventionally say that such edges are being traversed -1 times.)

Conversely, for any traversal of the fundamental path and the fundamental cycles in which the total number of times each edge is traversed is nonnegative, and in which the vertices and edges corresponding to a positive number of traversals form a connected graph, there is at least one equivalent path from "start" to "stop." ∎

The fundamental cycles are found by picking a free subtree as in Fig. 32; if we choose a different subtree we get, in general, a different set of fundamental cycles. The fact that there are $m - n + 1$ fundamental cycles follows from Theorem A. The modifications we made to get from Fig. 31 to Fig. 32 do not

change the value of $m - n + 1$, although they may increase both m and n; the construction could have been generalized so as to avoid these trivial modifications entirely (see exercise 8).

Theorem K is encouraging because it says that Kirchhoff's law (which consists of n equations in the m unknowns $E_1, E_2, \ldots, E_m$) has just one "redundancy," i.e., these n equations allow us to eliminate $n - 1$ unknowns. Note however that throughout this discussion the unknown variables have been the number of times the *edges* have been traversed, not the number of times each *box* of the flow chart has been entered.

Algorithms for finding all cycles in a graph are discussed by J. T. Welch, Jr., in *JACM* **13** (1966), 205–210.

EXERCISES

1. [*14*] List all cycles from B to B which are present in the graph of Fig. 29.

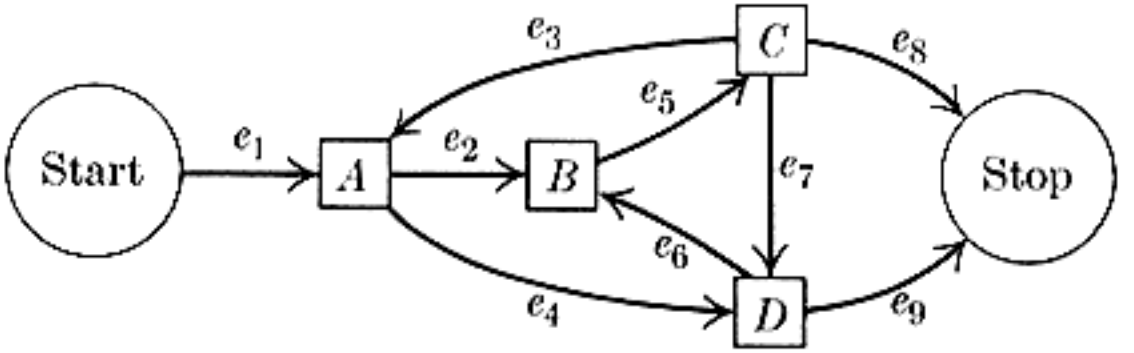
2. [*M20*] Prove that if V and V' are vertices of a graph and if there is a path from V to V', then there is a simple path from V to V'.

3. [*15*] What path from "start" to "stop" is equivalent (in the sense of Theorem K) to one traversal of the fundamental path plus one traversal of cycle C_2 in Fig. 32?

▶ **4.** [*M20*] Let G' be a finite free tree in which arrows have been drawn on its edges $e_1, \ldots, e_{n-1}$; let $E_1, \ldots, E_{n-1}$ be numbers satisfying Kirchhoff's law (1) in G'. Show that $E_1 = \cdots = E_{n-1} = 0$.

5. [*20*] Using Eqs. (6), express the quantities $A, B, \ldots, S$ which appear inside the boxes of Fig. 31 in terms of the independent variables $E_2, E_5, \ldots, E_{25}$.

6. [*22*] Carry out the construction in the text for the flow chart



using the free subtree consisting of edges e_1, e_2, e_3, e_4, e_9. What are the fundamental cycles? Express E_1, E_2, E_3, E_4, E_9 in terms of E_5, E_6, E_7, and E_8.

▶ **7.** [*M23*] The text shows that if we apply Kirchhoff's laws to the number of times each edge is traversed, $n - 1$ of the unknowns are eliminated (if there are n boxes in the flow chart). Discuss the effect of Kirchhoff's law on the number of independent unknowns if we let the unknowns be the number of times each *box* is entered instead of the number of times each edge is traversed. (Thus, we might take $A, B, \ldots, S$ to be the unknowns in the text's example, instead of $E_1, E_2, \ldots, E_{27}$.)

8. [*M22*] Edges e_{13} and e_{19} were split into two parts in Fig. 32, since a graph is not supposed to have two edges joining the same two vertices. However, if we look at the final result of the construction, this splitting into two parts seems quite artificial

since $E'_{13} = E''_{13}$, $E'_{19} = E''_{19}$ are two of the relations found in (6), and E''_{13}, E''_{19} are two of the independent variables. Explain how the construction could be generalized so that an artificial splitting of edges may be avoided.

▶ **9.** [*M27*] Suppose a graph has n vertices $V_1, \ldots, V_n$ and m edges $e_1, \ldots, e_m$. Each edge e_k is represented by a pair of integers (a_k, b_k) giving the numbers of the vertices which it makes adjacent. Design an algorithm which takes the input pairs $(a_1, b_1), \ldots, (a_m, b_m)$ and prints out a subset of these which forms a free tree; the algorithm reports failure if this is impossible. Try to make the algorithm as efficient as possible.

10. [*16*] An electrical engineer, designing the circuitry for a computer, finds that he has n terminals $T_1, T_2, \ldots, T_n$ which he wants to have at essentially the same voltage at all times. To achieve this, he can solder wires between any pairs of terminals; the idea is to make enough wire connections so that there is a path through the wires from any terminal to any other. Show that the minimum number of wires needed to connect all the terminals is $n - 1$, and $n - 1$ wires achieve the desired connection if and only if they form a free tree (with terminals and wires standing for vertices and edges).

11. [*M27*] (R. C. Prim, *Bell System Tech. J.* **36** (1957), 1389–1401.) Consider the wire connection problem of exercise 10 with the additional proviso that a cost $c(i, j)$ is given for each $i < j$, denoting the expense of wiring terminal T_i to terminal T_j. Show that the following algorithm gives a connection tree of minimum cost: "If $n = 1$, do nothing. Otherwise, renumber the terminals and costs so that

$$c(n-1, n) = \min_{1 \le i < n} c(i, n);$$

connect terminal T_{n-1} to T_n; then change $c(j, n-1)$ to $\min\big(c(j, n-1), c(j, n)\big)$ for $1 \le j < n - 1$, and repeat the algorithm for $n - 1$ terminals $T_1, \ldots . T_{n-1}$ using these new costs. (The algorithm is to be repeated with the understanding that whenever a connection is subsequently requested between the terminals now called T_j and T_{n-1}, the connection is actually made between terminals now called T_j and T_n if it is cheaper; thus T_{n-1} and T_n are being regarded as though they were one terminal in the remainder of the algorithm.)" This algorithm may also be stated as follows: "Choose a particular terminal to start with; then repeatedly make the cheapest possible connection from an unchosen terminal to a chosen one, until all have been chosen."

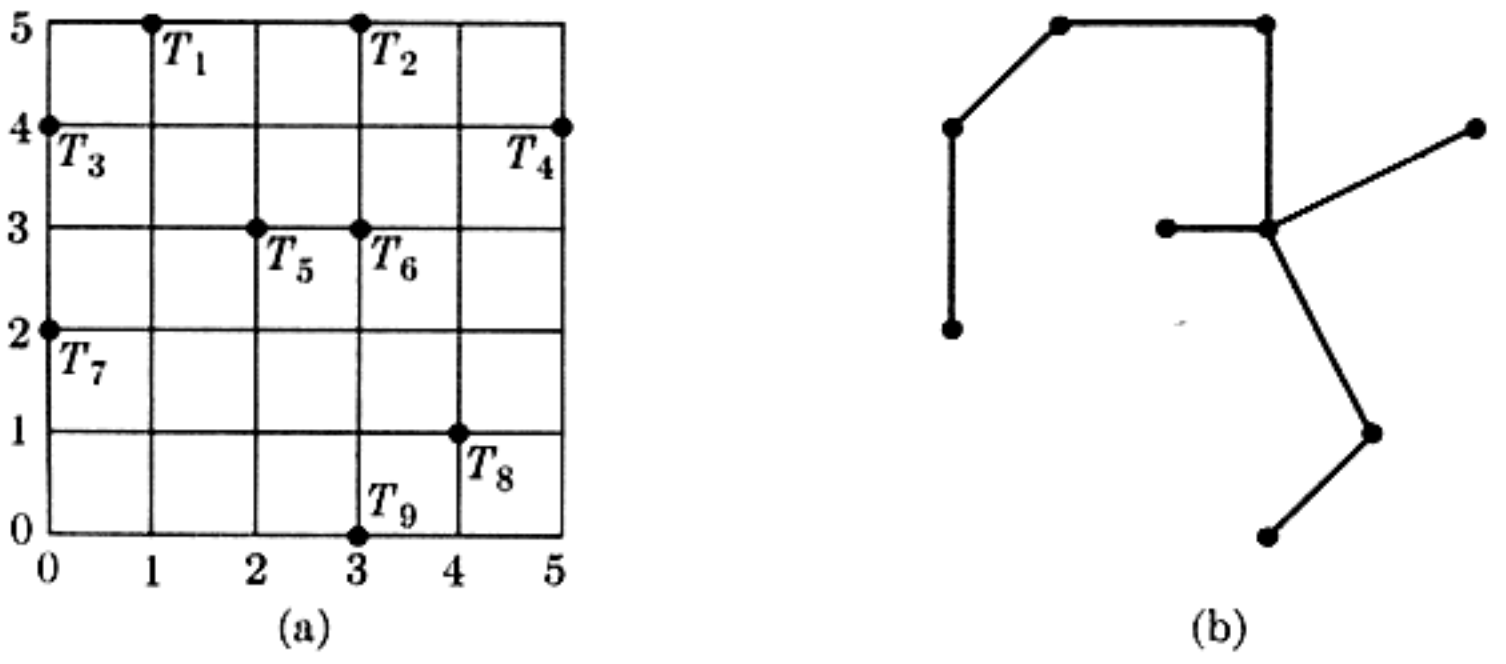


Fig. 33. Free tree of minimum cost. (See exercise 11.)

For example, consider Fig. 33(a), which shows nine terminals on a grid; let the cost of connecting two terminals be the wire length, i.e., the distance between them. (The reader may wish to try to find a minimal cost tree by hand, using intuition instead of the above algorithm.) The algorithm would first connect T_8 to T_9, then T_6 to T_8, T_5 to T_6, T_2 to T_6, T_1 to T_2, T_3 to T_1, T_7 to T_3, and finally T_4 to either T_2 or T_6. A minimum cost tree (with total wire length $7 + 2\sqrt{2} + 2\sqrt{3}$) is shown in Fig. 33(b).

▸ **12.** [*29*] The algorithm of exercise 11 is not stated in a fashion suitable for direct computer implementation. Reformulate that algorithm, specifying in more detail the operations that are to be done, in such a way that a computer program can carry out the process with reasonable efficiency.

13. [*M20*] Let G be a graph (possibly infinite) which contains no cycles. Assume that if any edge not already present in G is added to G, a cycle is formed. Prove that G is a free tree.

14. [*M24*] Consider a graph with n vertices and m edges, in the notation of exercise 9. Show that it is possible to write any permutation of the integers $\{1, 2, \ldots, n\}$ as a product of transpositions $(a_{k_1}b_{k_1})(a_{k_2}b_{k_2}) \cdots (a_{k_t}b_{k_t})$ if and only if the graph is connected. (Hence there are sets of $n - 1$ transpositions which generate all permutations on n elements, but no set of $n - 2$ will do so.)

2.3.4.2. Oriented trees. In the previous section, we saw that an abstracted flow chart may be regarded as a graph, if we ignore the direction of the arrows on its edges; the graph-theoretic ideas of "cycle" and "free subtree," etc., were shown to be relevant in the study of flow charts. There is a good deal more that can be said when the direction of each edge is given more significance, and in this case we have what is called a "directed graph" or "digraph."

Let us define a *directed graph* formally as a set of vertices and a set of *arcs*, each arc leading from a vertex V to a vertex V'. If e is an arc from V to V' we say V is the *initial* vertex of e, and V' is the *final* vertex, and we write $V = \text{init}\ (e)$, $V' = \text{fin}\ (e)$. The case that $\text{init}\ (e) = \text{fin}\ (e)$ is not excluded (although it was excluded from the definition of edge in an ordinary graph), and several different arcs may have the same initial and final vertices. The *out-degree* of a vertex V is the number of arcs leading out from it, i.e., the number of arcs e such that $\text{init}\ (e) = V$; similarly, the *in-degree* of V is the number of arcs with $\text{fin}\ (e) = V$.

The concepts of paths, cycles, etc. are defined for directed graphs in a manner similar to the corresponding definitions for ordinary graphs, but there are some important new technicalities that must be considered. If $e_1, e_2, \ldots, e_n$ are arcs (with $n \geq 1$), we say $(e_1, e_2, \ldots, e_n)$ is an *oriented path* of length n from V to V' if $V = \text{init}\ (e_1)$, $V' = \text{fin}\ (e_n)$, and $\text{fin}\ (e_k) = \text{init}\ (e_{k+1})$ for $1 \leq k < n$. An oriented path $(e_1, e_2, \ldots, e_n)$ is called *simple* if $\text{init}\ (e_1), \ldots, \text{init}\ (e_n)$ are distinct and $\text{fin}\ (e_1), \ldots, \text{fin}\ (e_n)$ are distinct. An *oriented cycle* is a simple oriented path from a vertex to itself. (Note that an oriented cycle can have length 1 or 2, but this was excluded from our definition of "cycle" in the previous section. Can the reader see why this makes sense?)

As examples of these straightforward definitions, we may refer to Fig. 31 in the previous section. The box labeled "J" is a vertex with in-degree 2 (because of the arcs e_{16}, e_{21}) and out-degree 1. The sequence $(e_{17}, e_{19}, e_{18}, e_{22})$ is an oriented path of length 4 from J to P; this path is not simple since, for example, init (e_{19}) = L = init (e_{22}). The diagram contains no oriented cycles of length 1, but (e_{18}, e_{19}) is an oriented cycle of length 2.

A directed graph is said to be *strongly connected* if there is an oriented path from V to V' for any two vertices $V \neq V'$. It is said to be *rooted* if there is at least one "root," i.e., at least one vertex R such that there is an oriented path from V to R for all $V \neq R$. ["Strongly connected" always implies "rooted," but the converse does not hold. A flow chart such as Fig. 31 in the previous section is an example of a rooted digraph, with R the "stop" vertex; with the additional arc from "stop" to "start" (Fig. 32) it becomes strongly connected.]

Every directed graph G corresponds in an obvious manner to an ordinary graph G_0, where G_0 has an edge from V to V' if and only if $V \neq V'$ and G has an arc from V to V' or from V' to V. We can speak of (unoriented) *paths* and *cycles* in G with the understanding that these are paths and cycles of G_0; we can say that G is *connected* (this is a much weaker property than "strongly connected," even weaker than "rooted") if the corresponding graph G_0 is connected, etc.

An *oriented tree* (see Fig. 34), sometimes called a "rooted tree" by other authors, is a directed graph with a specified vertex R such that:

a) Each vertex $V \neq R$ is the initial vertex of exactly one arc, denoted by $e[V]$;

b) R is the initial vertex of no arc;

c) R is a root in the sense defined above (i.e., for each vertex $V \neq R$ there is an oriented path from V to R).

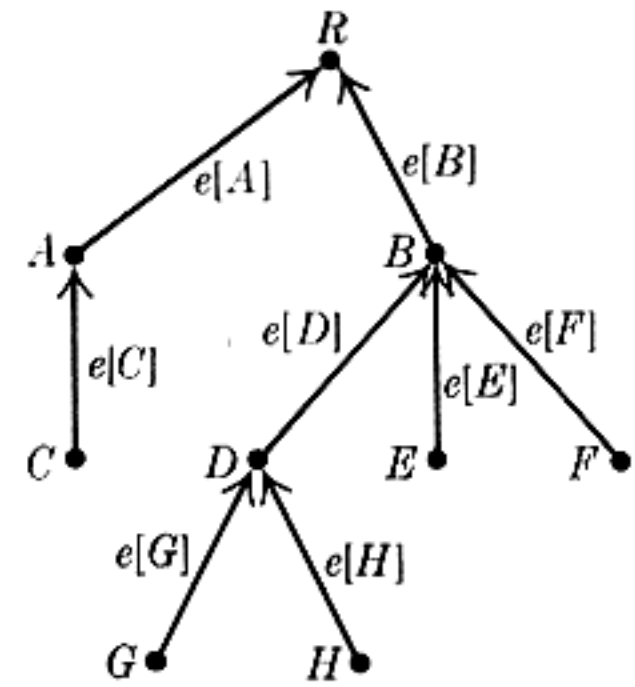


Fig. 34. An oriented tree.

It follows immediately that for each vertex $V \neq R$ there is a *unique* oriented path from V to R; and hence there are no oriented cycles.

Our previous definition of "oriented tree" (at the beginning of Section 2.3) is easily seen to be compatible with the new definition just given, when there are finitely many vertices; the vertices correspond to nodes, and the arc $e[V]$ is the link from V to FATHER(V).

The (undirected) graph corresponding to an oriented tree is connected [because of property (c)], and contains no cycles [since by property (a) if $(V_0, V_1, \ldots, V_n)$ is a cycle with $n \geq 3$, either

$$(e[V_0], e[V_1], \ldots, e[V_{n-1}]) \qquad \text{or} \qquad (e[V_n], \ldots, e[V_2], e[V_1])$$

is an oriented cycle]; therefore *an oriented tree is a free tree* when the direction of the arcs is neglected.

Conversely, it is important to note that we can reverse the process just described. If we start with any nonempty free tree, such as that in Fig. 30, we can choose *any* vertex as the root R, and assign directions to the edges. The intuitive idea is to "pick up" the graph at vertex R and shake it; then assign upward-pointing arrows. More formally, the rule is this:

> Change the edge VV' to an arc from V to V' if and only if the simple path from V to R leads through V', that is, if it has the form $(V_0, V_1, \ldots, V_n)$, where $n > 0$, $V_0 = V$, $V_1 = V'$, $V_n = R$.

To verify that such a construction is valid, we need to prove that each edge VV' is assigned the direction $V \rightarrow V'$ or the direction $V' \leftarrow V$; and this is easy to prove, for if $(V, V_1, \ldots, R)$ and $(V', V_1', \ldots, R)$ are paths, there is a cycle unless $V = V_1'$ or $V_1 = V'$. It is a consequence of this construction that the directions of the arcs in an oriented tree are completely determined by knowing which vertex is the root, so they need not be shown in diagrams when the root is explicitly indicated.

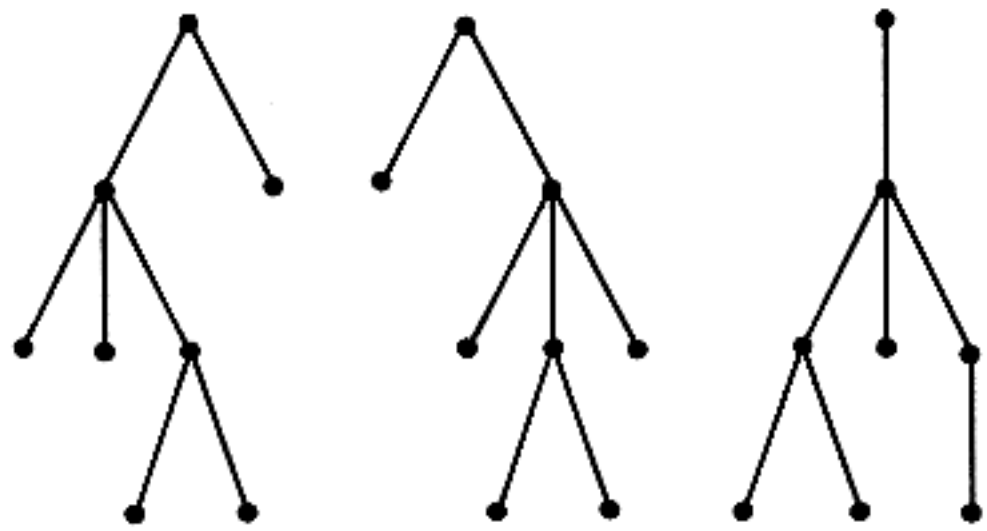


Fig. 35. Three tree structures.

We now see the relation between three types of trees: the (ordered) tree which is of principal importance in computer programs, as defined at the beginning of Section 2.3; the oriented tree (i.e., unordered tree); and the free tree. Both of the latter two types of trees arise in the study of computer algorithms, but not as often as the first type. *The essential distinction between these types of tree structure is merely the amount of information that is taken to be relevant.* For example, Fig. 35 shows three distinct trees if they are considered as ordered trees (with root at the top). As oriented trees, the first and second are identical, since the left-to-right order of subtrees is immaterial; as free trees, all three graphs in Fig. 35 are identical, since the root is immaterial.

An *Eulerian circuit* in a directed graph is an oriented path $(e_1, e_2, \ldots, e_m)$ such that *every* arc in the directed graph occurs exactly once, and fin (e_m) = init (e_1). This is a "complete traversal" of the arcs of the directed graph. (Eulerian circuits get their name from Leonhard Euler's famous discussion in 1736 of the impossibility of traversing each of the seven bridges in the city of Königsberg exactly once during a Sunday stroll. He treated the analogous problem for

undirected graphs. Eulerian circuits should be distinguished from "Hamiltonian circuits," which are oriented cycles that encounter each *vertex* exactly once; see Chapter 7.)

A directed graph is said to be *balanced* (see Fig. 36) if every vertex V has the same in-degree as its out-degree, i.e., there are just as many edges with V as their initial vertex as there are with V as their final vertex. This condition is closely related to Kirchhoff's law (see exercise 24). It is obviously possible to find an Eulerian circuit in a directed graph only if the graph is connected and balanced, provided that there are no *isolated vertices*, i.e., vertices with in-degree and out-degree both equal to zero.

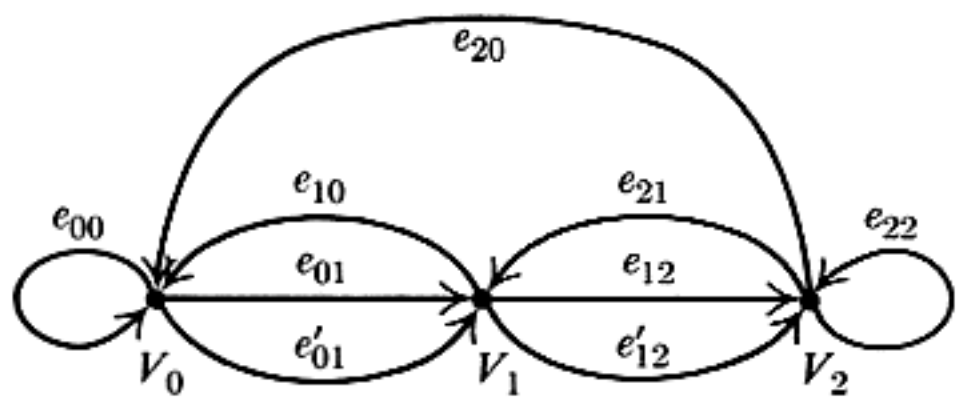


Fig. 36. A balanced directed graph.

So far in this section there have been quite a few definitions (e.g., directed graph, arc, initial vertex, final vertex, out-degree, in-degree, oriented path, simple oriented path, oriented cycle, oriented tree, Eulerian circuit, isolated vertex, and the properties of being strongly connected, rooted, and balanced), but there has been a scarcity of important results connecting these concepts. Now we are ready for meatier material. The first basic result is a theorem due to I. J. Good [*J. London Math. Soc.* **21** (1947), 167–169], who showed that Eulerian circuits are always possible unless they are obviously impossible:

Theorem G. *A finite, directed graph with no isolated vertices possesses an Eulerian circuit if and only if it is connected and balanced.*

Proof. Assume G is balanced and let

$$P = (e_1, \ldots, e_m)$$

be an oriented path of longest possible length that uses no arc twice. Then if $V = \text{fin}\ (e_m)$, and if k is the out-degree of V, all k arcs e with $\text{init}\ (e) = V$ must already appear in P, otherwise we could add e and get a longer path. But if $\text{init}\ (e_j) = V$ and $j > 1$, then $\text{fin}\ (e_{j-1}) = V$; hence, since G is balanced, we must have

$$\text{init}\ (e_1) = V = \text{fin}\ (e_m),$$

otherwise the in-degree of V would be at least $k + 1$.

Now by cyclic permutation of P it follows that any arc e not in the path has neither initial nor final vertex in common with any arc in the path; so if P is not an Eulerian circuit, G is not connected. ∎

There is an important connection between Eulerian circuits and oriented trees:

Lemma E. *Let $(e_1, \ldots, e_m)$ be an Eulerian circuit of a directed graph G having no isolated vertices. Let $R = \text{fin}\,(e_m) = \text{init}\,(e_1)$. For each vertex $V \neq R$ let $e[V]$ be the "last exit" from V in the circuit, i.e.,*

$$e[V] = e_j \quad \text{if} \quad \text{init}\,(e_j) = V \quad \text{and} \quad \text{init}\,(e_k) \neq V \quad \text{for} \quad j < k \leq m. \tag{1}$$

Then the vertices of G with the arcs $e[V]$ form an oriented tree with root R.

Proof. Properties (a) and (b) of the definition of oriented tree are evidently satisfied. By exercise 7 we need only show there are no oriented cycles among the $e[V]$; but this is immediate, since if $\text{fin}\,(e[V]) = V' = \text{init}\,(e[V'])$, where $e[V] = e_j$ and $e[V'] = e_{j'}$, then $j < j'$. ∎

This lemma can perhaps be better understood if we turn things around and consider the "first entrances" to each vertex; the first entrances form an unordered tree with all arcs pointing *away* from R. Lemma E has a surprising and important converse, proved by T. van Aardenne-Ehrenfest and N. G. de Bruijn [*Simon Stevin* **28** (1951), 203–217]:

Theorem D. *Let G be a finite, balanced, directed graph, and let G' be an oriented tree consisting of the vertices of G plus some of the arcs of G. Let R be the root of G' and let $e[V]$ be the arc of G' with initial vertex V. Let e_1 be any arc of G with $\text{init}\,(e_1) = R$. Then $P = (e_1, e_2, \ldots, e_m)$ is an Eulerian circuit if it is an oriented path for which*

i) *no arc is used more than once; i.e., $e_j \neq e_k$ when $j \neq k$.*
ii) *$e[V]$ is not used in P unless it is the only choice consistent with rule* (i)*; i.e., if $e_j = e[V]$ and if e is an arc with $\text{init}\,(e) = V$, then $e = e_k$ for some $k \leq j$.*
iii) *P terminates only when it cannot be continued by rule* (i)*; i.e., if $\text{init}\,(e) = \text{fin}\,(e_m)$, then $e = e_k$ for some k.*

Proof. By (iii) and the argument in the proof of Theorem G, we must have $\text{fin}\,(e_m) = \text{init}\,(e_1) = R$. Now if e is an arc not appearing in P, let $V = \text{fin}\,(e)$. Since G is balanced, it follows that V is the initial vertex of some arc not in P; and if $V \neq R$, $e[V]$ must not be in P by condition (ii). Now use the same argument with $e = e[V]$, and we ultimately find R is the initial vertex of some arc not in the path, contradicting (iii). ∎

The essence of Theorem D is that it shows us a simple way to construct an Eulerian circuit in a balanced directed graph, given any oriented subtree of the graph. (See the example in exercise 14.) In fact, Theorem D allows us to count the exact number of Eulerian circuits in a directed graph; this result and many other important consequences of the ideas developed in this section appear in the exercises which follow.

EXERCISES

1. [*M20*] Prove that if V and V' are vertices of a directed graph and if there is an oriented path from V to V', then there is a simple oriented path from V to V'.

2. [*15*] Which of the ten "fundamental cycles" listed in (3) of Section 2.3.4.1 are *oriented* cycles in the directed graph (Fig. 32) of that section?

3. [*16*] Draw the diagram for a directed graph that is connected but not rooted.

▶ **4.** [*M20*] The concept of *topological sorting* can be defined for any finite directed graph G as a linear arrangement of the vertices such that init (e) precedes fin (e) in the ordering for all edges e of G. (Cf. Section 2.2.3, Figs. 6 and 7.) Not all finite directed graphs can be topologically sorted; which ones can be? (Use the terminology of this section to give the answer.)

5. [*M21*] Let G be a directed graph which contains an oriented path $(e_1, \ldots, e_n)$ with fin (e_n) = init (e_1). Give a proof that G is not an oriented tree, using the terminology defined in this section.

6. [*M21*] True or false: A directed graph which is rooted and contains no cycles and no oriented cycles is an oriented tree.

▶ **7.** [*M22*] True or false: A directed graph satisfying properties (a) and (b) of the definition of oriented tree, and having no oriented cycles, is an oriented tree.

8. [*HM40*] Study the properties of *automorphism groups* of oriented trees, i.e., the groups consisting of those permutations π of the vertices and arcs such that init $(e\pi)$ = init $(e)\pi$, fin $(e\pi)$ = fin $(e)\pi$.

9. [*18*] By assigning directions to the edges, draw the oriented tree corresponding to the free tree in Fig. 29 of Section 2.3.4.1, with G as the root.

10. [*22*] An oriented tree with vertices $V_1, \ldots, V_n$ can be represented inside a computer by using a table $F[1], \ldots, F[n]$ as follows: If V_j is the root, $F[j] = 0$; otherwise $F[j] = k$, if the arc $e[V_j]$ goes from V_j to V_k. (Thus $F[1], \ldots, F[n]$ is the same as the "father" table used in Algorithm 2.3.3E.)

The text shows how a free tree can be converted into an oriented tree by choosing any desired vertex to be the root. Consequently, it is possible to start with an oriented tree that has root R, then to convert this into a free tree by neglecting the orientation of the arcs, and finally to assign new orientations, obtaining an oriented tree with any specified vertex as the root. Design an algorithm which performs this transformation: Starting with a table $F[1], \ldots, F[n]$, representing an oriented tree, and given an integer j, $1 \leq j \leq n$, design the algorithm to transform the F table so that it represents the same free tree but with V_j as the root.

▶ **11.** [*28*] Using the assumptions of exercise 2.3.4.1–9, but with (a_k, b_k) representing an edge whose arrow points from V_{a_k} to V_{b_k}, design an algorithm which not only prints out a free subtree as in that algorithm, but also prints out the fundamental cycles. [*Hint:* The algorithm given in the solution to exercise 2.3.4.1–9 can be combined with the algorithm in the preceding exercise.]

12. [*M10*] In the correspondence between oriented trees as defined here and oriented trees as defined at the beginning of Section 2.3, is the *degree* of a tree node equal to the *in-degree* or the *out-degree* of the corresponding vertex?

▸ **13.** [*M24*] Prove that if R is a root of a (possibly infinite) directed graph G, then G contains an oriented subtree with the same vertices as G and with root R. (As a consequence, it is always possible to choose the free subtree in flow charts like Fig. 32 of Section 2.3.4.1 so that it is actually an *oriented* subtree; this would be the case in that diagram if we had selected e''_{13}, e''_{19}, e_{20}, and e_{17} instead of e'_{13}, e'_{19}, e_{23}, and e_{15}.)

14. [*21*] Let G be the directed graph shown in Fig. 36, and let G' be the oriented subtree with vertices V_0, V_1, V_2 and arcs e_{01}, e_{21}. Find all paths P that meet the conditions of Theorem D, starting with arc e_{12}.

15. [*M20*] Prove that a directed graph which is connected and balanced is strongly connected.

▸ **16.** [*M24*] In a popular solitaire game called "clock," the 52 cards of an ordinary deck of playing cards are dealt face down into 13 piles of four each; 12 piles are arranged in a circle like the 12 hours of a clock and the thirteenth pile goes in the center. The solitaire game now proceeds by turning up the top card of the center pile, and then if its face value is k, we place it next to the kth pile. (1, 2, . . . , 13 are equivalent to A, 2, . . . , 10, J, Q, K.) Play continues by turning up the top card of the kth pile and putting it next to *its* pile, etc., until we reach a point where it is impossible to continue since there are no more cards to turn up on the designated pile. (The player has no choice in the game, since the above rules completely specify his actions.) The game is won if all cards are face up when play terminates. [*Reference:* A. Moyse, Jr., *150 ways to play solitaire* (Chicago: Whitman, 1950).]

Show that the game will be won if and only if the following directed graph is an oriented tree: The vertices are $V_1, V_2, \ldots, V_{13}$; the arcs are $e_1, e_2, \ldots, e_{12}$, where e_j goes from V_j to V_k if k is the *bottom* card in pile j after the deal.

(In particular, if the bottom card of pile j is a "j", for $j \neq 13$, it is easy to see that the game is certainly lost, since this card could never be turned up. The result proved in this exercise gives a much faster way to play the game!)

17. [*M32*] What is the probability of winning the solitaire game of clock (described in exercise 16), assuming the deck is randomly shuffled? What is the probability that exactly k cards are still face down when the game is over?

18. [*M30*] (Okada and Onodera, *Bull. Yamagato Univ.* **2** (1952), 89–117.) Let G be a graph with $n+1$ vertices $V_0, V_1, \ldots, V_n$ and m edges $e_1, \ldots, e_m$. Make G into a directed graph by assigning an arbitrary orientation to each edge; then construct the $m \times (n+1)$ matrix A with

$$a_{ij} = \begin{cases} +1, & \text{if init } (e_i) = V_j; \\ -1, & \text{if fin } (e_i) = V_j; \\ 0, & \text{otherwise.} \end{cases}$$

Let A_0 be the $m \times n$ matrix A with column 0 deleted.

a) If $m = n$, show that the determinant of A_0 is equal to 0 if G is not a free tree, and equal to ± 1 if G *is* a free tree.

b) Show that for general m the determinant of $A_0^T A_0$ is the number of free subtrees of G (i.e., the number of ways to choose n of the m edges so that the resulting graph is a free tree). [*Hint:* Use (a) and the result of exercise 1.2.3–46.]

19. [*M31*] (W. T. Tutte, *Proc. Cambridge Phil. Soc.* **44** (1948), 463–482.) Let G be a directed graph with vertices $V_0, V_1, \ldots, V_n$. Let A be the $(n+1)\times(n+1)$ matrix with

$$a_{ij} = \begin{cases} -k, & \text{if } i \neq j \text{ and there are } k \text{ arcs from } V_i \text{ to } V_j; \\ t, & \text{if } i = j \text{ and there are } t \text{ arcs from } V_j \text{ to other vertices.} \end{cases}$$

(It follows that $a_{i0} + a_{i1} + \cdots + a_{in} = 0$ for $0 \le i \le n$.) Let A_0 be the same matrix with row 0 and column 0 deleted. For example, if G is the directed graph of Fig. 36, we have

$$A = \begin{pmatrix} 2 & -2 & 0 \\ -1 & 3 & -2 \\ -1 & -1 & 2 \end{pmatrix}, \qquad A_0 = \begin{pmatrix} 3 & -2 \\ -1 & 2 \end{pmatrix}.$$

a) Show that in the special case $a_{00} = 0$ and $a_{jj} = 1$ for $1 \le j \le n$, and if G contains no arcs from a vertex to itself, then G is an oriented tree with root V_0 if and only if $\det A_0 = 1$; and if G is not an oriented tree, then $\det A_0 = 0$.

b) Show that in the general case, $\det A_0$ is the number of oriented subtrees of G with root V_0 (i.e., the number of ways to select n of the arcs of G so that the resulting directed graph is an oriented tree, with V_0 as the root). [*Hint:* Use induction on the number of arcs.]

20. [*M21*] If G is a graph on $n+1$ vertices $V_0, \ldots, V_n$, let B be the $n \times n$ matrix defined as follows for $1 \le i, j \le n$:

$$b_{ij} = \begin{cases} t, & \text{if } i = j \text{ and there are } t \text{ edges touching } V_j; \\ -1, & \text{if } i \neq j \text{ and } V_i \text{ is adjacent to } V_j; \\ 0, & \text{otherwise.} \end{cases}$$

For example, if G is the graph of Fig. 29 in Section 2.3.4.1, with $(V_0, V_1, V_2, V_3, V_4) = (A, B, C, D, E)$, we find that

$$B = \begin{pmatrix} 3 & 0 & -1 & -1 \\ 0 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix}.$$

Show that the number of free subtrees of G is $\det B$. [*Hint:* Use exercise 18 or 19.]

21. [*HM38*] Fig. 36 is an example of a directed graph that is not only balanced, it is *regular*, which means every vertex has the same in-degree and out-degree as every other vertex. Let G be a regular directed graph with $n+1$ vertices $V_0, V_1, \ldots, V_n$, in which every vertex has in-degree and out-degree equal to m. (Hence there are $(n+1)m$ arcs in all.) Let G^* be the graph with $(n+1)m$ vertices corresponding to the arcs of G; let a vertex of G^* corresponding to an arc from V_j to V_k in G be denoted by V_{jk}. An arc goes from V_{jk} to $V_{j'k'}$ in G^* if and only if $k = j'$. For example, if G is the directed graph of Fig. 36, G^* is as shown in Fig. 37. An Eulerian circuit in G is a Hamiltonian circuit in G^* and conversely.

Prove that the number of oriented subtrees of G^* is $m^{(n+1)(m-1)}$ times the number of oriented subtrees of G. [*Hint:* Use exercise 19.]

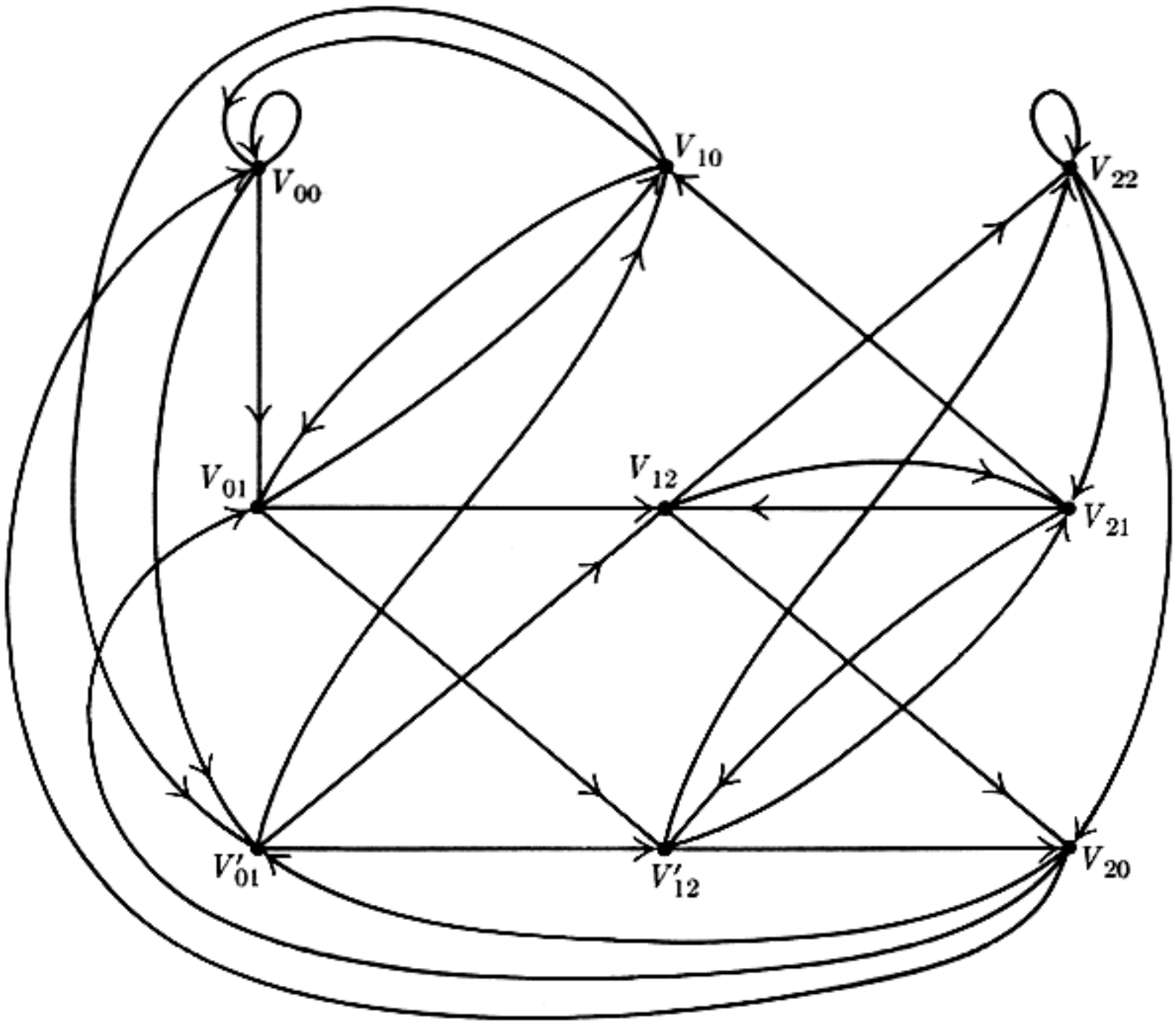


Fig. 37. Edge-graph corresponding to Fig. 36.

▸ **22.** [*M26*] Let G be a balanced, directed graph with vertices $V_1, V_2, \ldots, V_n$ and no isolated vertices. Let σ_j be the out-degree of V_j. Show that the number of Eulerian circuits of G is

$$(\sigma_1 + \sigma_2 + \cdots + \sigma_n)\, T \prod_{1 \le j \le n} (\sigma_j - 1)!,$$

where T is the number of oriented subtrees of G with root V_1. [*Note:* The factor $(\sigma_1 + \cdots + \sigma_n)$, which is the number of arcs of G, may be omitted if the Eulerian circuit $(e_1, \ldots, e_m)$ is regarded as equal to $(e_k, \ldots, e_m, e_1, \ldots, e_{k-1})$.]

▸ **23.** [*M33*] (N. G. de Bruijn.) For each set of nonnegative integers $x_1, \ldots, x_k$ less than m, let $f(x_1, \ldots, x_k)$ be a nonnegative integer less than m. Define an infinite sequence as follows: $X_1 = X_2 = \cdots = X_k = 0$; $X_{n+k+1} = f(X_{n+k}, \ldots, X_{n+1})$ when $n \ge 0$. For how many of the m^{m^k} possible functions f is this sequence periodic with a period of the maximum length m^k? [*Hint:* Construct a directed graph with vertices $(x_1, \ldots, x_{k-1})$ for all $0 \le x_j < m$, and with arcs from $(x_1, x_2, \ldots, x_{k-1})$ to $(x_2, \ldots, x_{k-1}, x_k)$; apply exercises 21 and 22.]

▸ **24.** [*M20*] Let G be a connected, directed graph with arcs $e_0, e_1, \ldots, e_m$. Let $E_0, E_1, \ldots, E_m$ be a set of positive integers which satisfy Kirchhoff's law for G, i.e., for each vertex V,

$$\sum_{\mathrm{init}(e_j)=V} E_j = \sum_{\mathrm{fin}(e_j)=V} E_j.$$

Assume further that $E_0 = 1$. Prove that there is an oriented path in G from fin (e_0) to init (e_0) such that edge e_0 does not appear in the path, and for $1 \le j \le m$ edge e_j appears exactly E_j times. [*Hint:* Apply Theorem G to a suitable directed graph.]

▸ **25.** [*26*] Design a computer representation for directed graphs which generalizes the right-threaded binary tree representation of a tree. Use two link fields `ALINK`, `BLINK` and two one-bit fields `ATAG`, `BTAG`; and design the representation so that: (a) there is one node for each *arc* of the directed graph (*not* for each vertex); (b) if the directed graph is an oriented tree with root R, and if we add an arc from R to a new vertex H, then the representation of this directed graph is essentially the same as a right-threaded representation of this oriented tree (with some order imposed on the sons in each family), such that `ALINK`, `BLINK`, `BTAG` are respectively the same as `LLINK`, `RLINK`, `RTAG` in Section 2.3.2; and (c) the representation is symmetric in the sense that interchanging `ALINK`, `ATAG` with `BLINK`, `BTAG` is equivalent to changing the direction on all the arcs of the directed graph.

▸ **26.** [*HM39*] ("Analysis of a random algorithm.") Let G be a directed graph on the vertices $V_1, V_2, \ldots, V_n$. Assume that G represents the flow chart for an algorithm, where V_1 is the "start" vertex and V_n is the "stop" vertex. (Therefore V_n is a root of G.) Suppose each arc e of G has been assigned a probability $p(e)$, where the probabilities satisfy the conditions

$$0 < p(e) \le 1; \qquad \sum_{\text{init}(e)=V_j} p(e) = 1, \qquad 1 \le j < n.$$

Consider a "random path," which starts at V_1 and which subsequently chooses branch e of G with probability $p(e)$, until V_n is reached; the choice of branch taken at each step is to be independent of all previous choices.

For example, consider the graph of exercise 2.3.4.1–6, and assign the respective probabilities $1, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 1, \frac{3}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}$ to arcs $e_1, e_2, \ldots, e_9$. Then the path "Start–A–B–C–A–D–B–C–Stop" is chosen with probability $1 \cdot \frac{1}{2} \cdot 1 \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{3}{4} \cdot 1 \cdot \frac{1}{4} = \frac{3}{128}$.

Such random paths are called *Markov chains*, after the Russian mathematician Andrei A. Markov who first made extensive studies of stochastic processes of this kind. The situation serves as a model for certain algorithms, although our requirement that each choice of path must be independent of the others is a very strong assumption. The problem we wish to solve here is to analyze the computation time for algorithms of this kind.

The analysis is facilitated by considering the $n \times n$ matrix $A = (a_{ij})$, where $a_{ij} = \sum p(e)$ summed over all arcs e which go from V_i to V_j. If there is no such arc, $a_{ij} = 0$. The matrix A for the example considered above is

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & \frac{1}{4} & \frac{1}{4} \\ 0 & 0 & \frac{3}{4} & 0 & 0 & \frac{1}{4} \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

It follows easily that $(A^k)_{ij}$ is the probability that a path starting at V_i will be at V_j after k steps.

Prove the following facts, for an arbitrary directed graph G of the above type: (a) The matrix $(I - A)$ is nonsingular. [Hint: Show there is no nonzero vector x with $xA^n = x$.] (b) The average number of times vertex V_j appears in the path is

$$(I - A)^{-1}_{1j} = \text{cofactor}_{j1}(I - A)/\det(I - A), \qquad \text{for} \qquad 1 \leq j \leq n.$$

[Thus in the example considered we find that the vertices A, B, C, D are traversed respectively $\frac{13}{6}$, $\frac{7}{3}$, $\frac{7}{3}$, $\frac{5}{3}$ times, on the average.] (c) The probability that V_j occurs in the path is

$$a_j = \text{cofactor}_{j1}(I - A)/\text{cofactor}_{jj}(I - A);$$

furthermore, $a_n = 1$, so the path terminates in a finite number of steps with probability one. (d) The probability that a random path starting at V_j will never return to V_j is

$$b_j = \det(I - A)/\text{cofactor}_{jj}(I - A).$$

(e) The probability that V_j occurs exactly k times in the path is

$$a_j(1 - b_j)^{k-1}b_j, \qquad \text{for} \qquad k \geq 1, \qquad 1 \leq j \leq n.$$

2.3.4.3. The "infinity lemma." Until now we have concentrated mainly on finite trees, i.e., trees with only finitely many vertices (nodes), but the definitions we have given for free trees and oriented trees apply to infinite graphs as well. Infinite *ordered* trees may be defined in several ways, for example, by extending the concepts of "Dewey decimal notation" to infinite collections of numbers, as in exercise 2.3–14. Even in the study of computer algorithms there is occasionally a need to know the properties of infinite trees (for example, in order to prove by contradiction that a certain tree is *not* infinite). One of the most fundamental properties of infinite trees, first stated in its full generality by D. König, is the following:

Theorem K. (*The "infinity lemma."*) *In any infinite oriented tree for which every vertex has finite degree, there is an "infinite path from the root," i.e., an infinite sequence of vertices* $V_0, V_1, V_2, \ldots$ *in which* V_0 *is the root and* fin $(e[V_{j+1}]) = V_j$ *for all* $j \geq 0$.

Proof. We define the path by starting with V_0, the root of the oriented tree. Assume that $j \geq 0$ and that V_j has been chosen having infinitely many descendants. The degree of V_j is finite by hypothesis, so V_j has finitely many sons $U_1, \ldots, U_n$. At least one of these sons must possess infinitely many descendants, so we take V_{j+1} to be such a son of V_j.

Now $V_0, V_1, V_2, \ldots$ is an infinite path from the root. ∎

Students of calculus may recognize that the argument used here is essentially like that used to prove the classical Bolzano-Weierstrass theorem, "A bounded, infinite set of real numbers has an accumulation point." One way of stating

Theorem K, as König observed, is: "Assuming the human race never dies out, there is a man now living having a line of descendants that will never die out."

Most people think that Theorem K is completely obvious when they first encounter it, but after more thought and a consideration of further examples they realize that there is something "profound" about the infinity lemma. Although the degree of each node of the tree is finite, we have not assumed it is *bounded* (less than some number N for all vertices), so there may be nodes with higher and higher degrees. If we stop to consider things carefully, it is at least conceivable that everyone's descendants will ultimately die out although there will be some families that go on a million generations, others a billion, etc., etc. In fact, H. W. Watson once published a "proof" that under certain laws of biological probability carried out indefinitely, there will be infinitely many people born in the future but each family line will die out with probability one. His paper [*J. Anthropological Inst. Gt. Britain and Ireland* **4** (1874), 138–144] actually contains important and far-reaching theorems in spite of the minor slip which caused him to make this erroneous statement, and it is significant that he did not find his conclusions to be logically inconsistent.

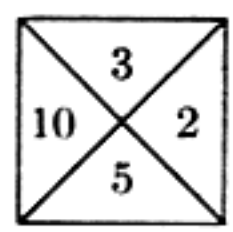
The contrapositive of Theorem K is directly applicable to computer algorithms: "If we have an algorithm that periodically divides itself up into finitely many subalgorithms, and if each chain of subalgorithms ultimately terminates, then the algorithm itself terminates."

Phrased yet another way, suppose we have a set S, finite or infinite, such that each element of S is a sequence $(x_1, x_2, \ldots, x_n)$ of positive integers of finite length $n \geq 0$. If we impose the conditions that

i) If $(x_1, \ldots, x_n)$ is in S, so is $(x_1, \ldots, x_k)$ for $0 \leq k \leq n$.
ii) If $(x_1, \ldots, x_n)$ is in S, only finitely many x_{n+1} exist for which $(x_1, \ldots, x_n, x_{n+1})$ is also in S.
iii) There is no infinite sequence $(x_1, x_2, \ldots)$ all of whose initial subsequences $(x_1, x_2, \ldots, x_n)$ lie in S.

Then S is essentially an oriented tree, specified essentially in a Dewey decimal notation, and Theorem K tells us S is *finite*.

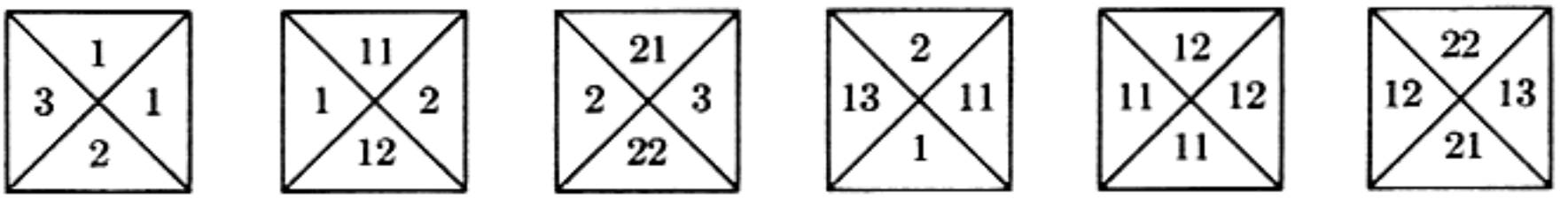
One of the most convincing examples of the potency of Theorem K has recently been given by Hao Wang, in connection with his "domino problem." A *domino type* is a square divided into four parts, each part having a specified number in it, e.g.,



(1)

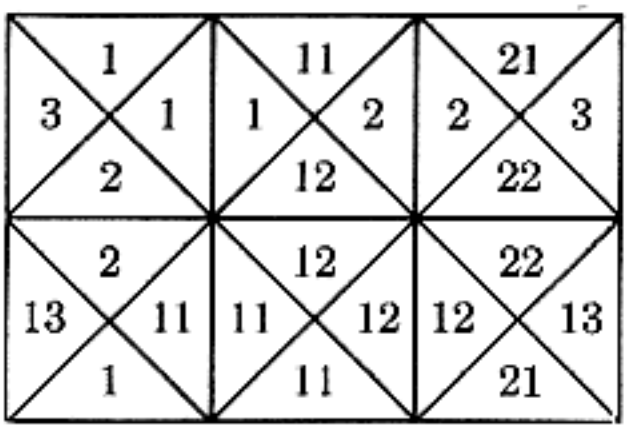
The problem of *tiling the plane* is to take a finite set of domino types, with an infinite supply of dominoes of each type, and to show how to place one in each square of an infinite plane (without rotating or reflecting the domino types) such that two dominoes are adjacent only if they have equal numbers where

they touch. For example, we can tile the plane using the six domino types



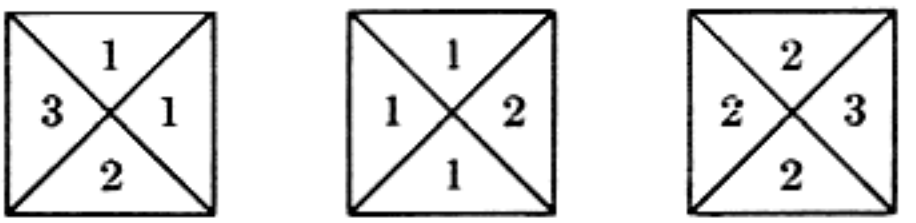
(2)

in essentially only one way, by repeating the rectangle



(3)

over and over. The reader may easily verify that there is no way to tile the plane with the three domino types



(4)

Wang's observation [see *Sci. Am.* **213** (November, 1965), 98–106] is that *if it is possible to tile the upper right quadrant of the plane, it is possible to tile the whole plane.* This is certainly unexpected, since a method for tiling the upper right quadrant involves a "boundary" along the x- and y-axes, and it would seem to give no hint as to how to tile the upper left quadrant of the plane (since domino types may not be rotated or reflected). We cannot get rid of the boundary merely by shifting the upper-quadrant solution down and to the left, since it does not make sense to shift the solution by more than a finite amount. But Wang's proof runs as follows: The existence of an upper-right-quadrant solution implies there is a way to tile a $2n \times 2n$ square, for all n. The set of all solutions to the problem of tiling squares with an even number of cells on each side forms an oriented tree, if the sons of each $2n \times 2n$ solution x are the possible $(2n+2) \times (2n+2)$ solutions that can be obtained by bordering x. The root of this oriented tree is the 0×0 solution; its sons are the 2×2 solutions, etc. Each node has only finitely many sons, since the problem of tiling the plane assumes only finitely many domino types are given; hence by the infinity lemma there is an infinite path from the root. This means there is a way to tile the whole plane!

EXERCISES

1. [*M10*] The text refers to a set S containing finite sequences of positive integers, and states that this set is "essentially an oriented tree." What is the root of this oriented tree, and what are the arcs?

2. [*20*] Show that if rotation of domino types is allowed, it is always possible to tile the plane.

▶ **3.** [*M23*] If it is possible to tile the upper right quadrant of the plane when given an *infinite* set of domino types, is it always possible to tile the whole plane?

4. [*M25*] (H. Wang.) The six domino types (2) lead to a "toroidal" solution to the tiling problem, i.e., a solution in which some rectangular pattern [namely (3)] is replicated throughout the entire plane.

Assume without proof that whenever it is possible to tile the plane with a finite set of domino types, there is a toroidal solution using those domino types. Use this assumption together with the infinity lemma to design an algorithm which, given the specifications of any finite set of domino types, determines in a finite number of steps whether or not there exists a way to tile the plane with these types.

5. [*M40*] Show that using the following 92 domino types it is possible to tile the plane, but that there is no "toroidal" solution in the sense defined in exercise 4.

To simplify the specification of the 92 types, let us first introduce some notation. Define the following "basic codes":

$$\begin{array}{llll}
\alpha = (1, 2, 1, 2) & \beta = (3, 4, 2, 1) & \gamma = (2, 1, 3, 4) & \delta = (4, 3, 4, 3) \\
a = (Q, D, P, R) & b = (\ ,\ , L, P) & c = (U, Q, T, S) & d = (\ ,\ , S, T) \\
N = (Y,\ ,X,\ ) & J = (D, U,\ , X) & K = (\ , Y, R, L) & B = (\ ,\ ,\ ,\ ) \\
R = (\ ,\ , R, R) & L = (\ ,\ , L, L) & P = (\ ,\ , P, P) & S = (\ ,\ , S, S) \\
 & T = (\ ,\ , T, T) & X = (\ ,\ , X, X) & \\
Y = (Y, Y,\ ,\ ) & U = (U, U,\ ,\ ) & D = (D, D,\ ,\ ) & Q = (Q, Q,\ ,\ )
\end{array}$$

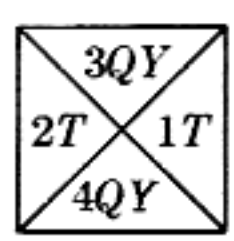
The domino types are now

$$\begin{array}{ll}
\alpha\{a, b, c, d\} & \text{[4 types]} \\
\beta\{Y\{B, U, Q\}\{P, T\}, \{B, U, D, Q\}\{P, S, T\}, K\{B, U, Q\}\} & \text{[21 types]} \\
\gamma\{\{\{X, B\}\{L, P, S, T\}, R\}\{B, Q\}, J\{L, P, S, T\}\} & \text{[22 types]} \\
\delta\{X\{L, P, S, T\}\{B, Q\}, Y\{B, U, Q\}\{P, T\}, N\{a, b, c, d\}, & \\
\quad J\{L, P, S, T\}, K\{B, U, Q\}, \{R, L, P, S, T\}\{B, U, D, Q\}\} & \text{[45 types]}
\end{array}$$

These abbreviations mean that the basic codes are to be put together component by component and sorted into alphabetic order in each component, thus: $\beta Y\{B, U, Q\}\{P, T\}$ stands for six types βYBP, βYUP, βYQP, βYBT, βYUT, βYQT. The type βYQT is

$$(3, 4, 2, 1)(Y, Y,\ ,\ )(Q, Q,\ ,\ )(\ ,\ , T, T) = (3QY, 4QY, 2T, 1T)$$

after multiplying corresponding components and sorting into order. This is intended to correspond to the domino type shown below where we use strings of symbols instead of numbers in the four quarters of the type. Two domino types can be placed next to each other only if they have the same string of symbols at the place they touch.



A domino type of "class β" means one which has a β in its specification as given above. To get started on the solution to this exercise, note that any domino of class β must have one of class α to its left and to its right, and that there must be one of class δ above and below. An "αa" domino must have "βKB" or "βKU" or "βKQ" to its right, and then must come an "αb" domino, etc.

(The above construction is a simplified version of a similar one given by Robert Berger, who went on to prove that the general problem in exercise 4, without the invalid assumption, cannot be solved. See *Memoirs Amer. Math. Soc.* **66** (1966).)

▸ **6.** [*M23*] (D. König.) In a famous paper [*Nieuw Archief voor Wiskunde* (2) **15** (1927), 212–216], B. L. van der Waerden proved that:

> "If k and m are positive integers, and if we have k sets $S_1, \ldots, S_k$ of positive integers with every positive integer included in at least one of these sets, then at least one of the sets S_j contains an arithmetic progression of length m."

(The latter statement means there exist integers a and $\delta > 0$ such that $a + \delta$, $a + 2\delta$, $\ldots$, $a + m\delta$ are all in S_j.) If possible, use this result and the infinity lemma to prove the stronger statement:

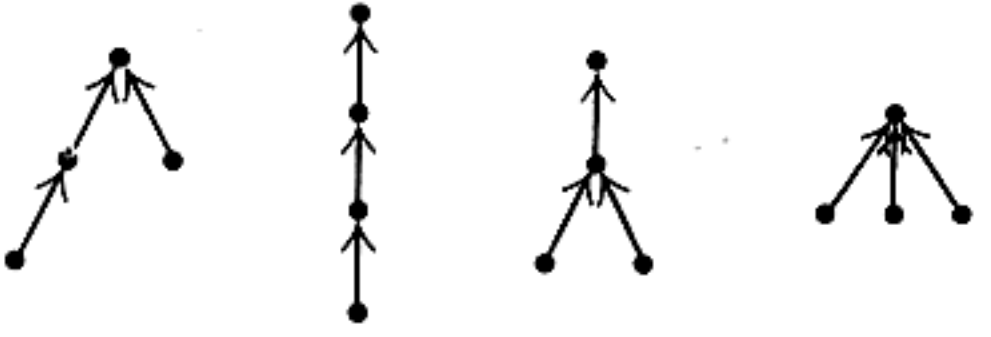
> "If k and m are positive integers, there is a number N such that if we have k sets $S_1, \ldots, S_k$ of integers with every integer between 1 and N included in at least one of these sets, then at least one of the sets S_j contains an arithmetic progression of length m."

▸ **7.** [*M30*] If possible, use van der Waerden's theorem of exercise 6 and the infinity lemma to prove the stronger statement:

> "If k is a positive integer, and if we have k sets $S_1, \ldots, S_k$ of integers with every positive integer included in at least one of these sets, then at least one of the sets S_j contains an infinitely long arithmetic progression."

▸ **8.** [*M34*] (J. B. Kruskal, *Trans. Am. Math. Soc.* **95** (1960), 210–225.) If T and T' are (finite, ordered) trees, let the notation $T \subseteq T'$ signify that T can be embedded in T', as in exercise 2.3.2–22. Prove that if $T_1, T_2, T_3, \ldots$ is any infinite sequence of trees, there exist integers $j < k$ such that $T_j \subseteq T_k$. (In other words, it is impossible to construct an infinite sequence of trees in which no tree "contains" any of the earlier trees of the sequence. This fact may be used to prove that certain algorithms must terminate.)

2.3.4.4. Enumeration of trees. Some of the most direct applications of the mathematical theory of trees to the analysis of algorithms are connected with formulas for counting how many different trees there are of various kinds. For example, if we want to know how many different oriented trees can be constructed having four vertices, we find there are just 4 possibilities:



(1)

For our first enumeration problem, let us determine the number a_n of structurally different oriented trees with n vertices. Obviously $a_1 = 1$. If $n > 1$, the tree has a root and various subtrees; suppose there are j_1 subtrees with 1 vertex, j_2 with 2 vertices, etc. Then we may choose j_k of the a_k possible trees in

$$\binom{a_k + j_k - 1}{j_k}$$

ways, since repetitions are allowed (cf. exercise 1.2.6–60), and so we see that

$$a_n = \sum_{j_1+2j_2+\cdots=n-1} \binom{a_1 + j_1 - 1}{j_1} \cdots \binom{a_{n-1} + j_{n-1} - 1}{j_{n-1}}. \tag{2}$$

If we consider the generating function $A(z) = \sum_n a_n z^n$, with $a_0 = 0$, we find that the identity

$$\frac{1}{(1 - z^r)^a} = \sum_j \binom{a + j - 1}{j} z^{rj}$$

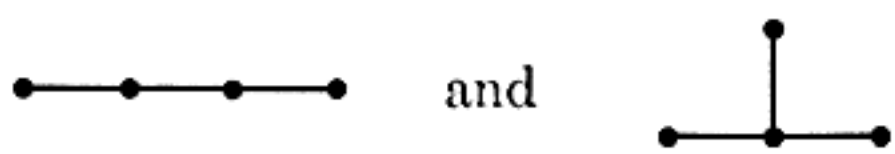
together with (2) implies

$$A(z) = z/(1 - z)^{a_1}(1 - z^2)^{a_2}(1 - z^3)^{a_3} \cdots. \tag{3}$$

This is not an especially nice form for $A(z)$, since it involves an infinite product and the coefficients $a_1, a_2, \ldots$ appear on the right-hand side; a somewhat more aesthetic way to represent $A(z)$ is given in exercise 1, and this leads to a reasonably efficient formula for calculating the values a_n (see exercise 2) and, in fact, it also shows the asymptotic behavior of a_n for large n (see exercise 4). We find that

$$\begin{aligned} A(z) = {} & z + z^2 + 2z^3 + 4z^4 + 9z^5 + 20z^6 + 48z^7 + 115z^8 \\ & + 286z^9 + 719z^{10} + 1842z^{11} + \cdots \end{aligned} \tag{4}$$

Now that we have essentially found the number of oriented trees, it is quite interesting to determine the number of structurally different *free trees* with n vertices. There are just two distinct free trees with four vertices, namely

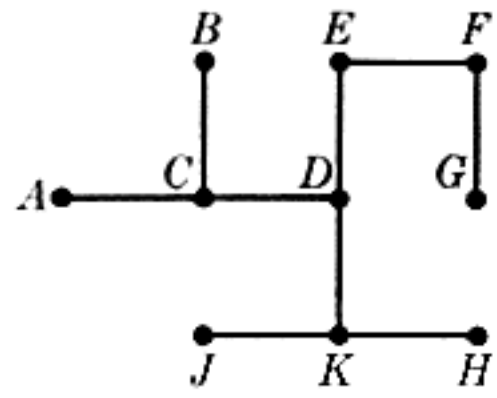


(5)

because the first two and last two oriented trees of (1) become identical when the orientation is dropped.

We have seen that it is possible to select any vertex X of a free tree and to assign directions to the edges in a unique way so that it becomes an oriented tree with X as root. Once this has been done, for a given vertex X, suppose there are k subtrees of the root X, with $s_1, s_2, \ldots, s_k$ vertices in these respective

subtrees. Clearly, k is the number of arcs touching X; and $s_1 + s_2 + \cdots + s_k = n - 1$, one less than the total number of vertices in the free tree. In these circumstances, we say that the *height* of X is $\max(s_1, s_2, \ldots, s_k)$. Thus in the tree



(6)

the vertex D has height 3 (each of the subtrees leading from D has three of the nine remaining vertices), and vertex E has height $\max(7, 2) = 7$. A vertex with minimum height is called a *centroid* of the free tree.

Let X and $s_1, s_2, \ldots, s_k$ be as above, and let $Y_1, Y_2, \ldots, Y_k$ be the roots of the subtrees emanating from X. Clearly, the height of Y_1 is at least $n - s_1 = 1 + s_2 + \cdots + s_k$, since when Y_1 is the assumed root there are $n - s_1$ points in its subtree through X. If there is a centroid Y in the Y_1 subtree, we have

$$\text{height}(X) = \max(s_1, s_2, \ldots, s_k) \geq \text{height}(Y) \geq 1 + s_2 + \cdots + s_k,$$

and this implies $s_1 > s_2 + \cdots + s_k$. A similar result may be derived if we replace Y_1 by Y_j in this discussion. So *at most one of the subtrees at a vertex can contain a centroid.*

This is a strong condition, for it implies that *there are at most two centroids in a free tree, and if two centroids exist, they are adjacent.* (See exercise 9.)

Conversely, if $s_1 > s_2 + \cdots + s_k$, there *is* a centroid in the Y_1 subtree, since

$$\text{height}(Y_1) \leq \max(s_1 - 1, 1 + s_2 + \cdots + s_k) \leq s_1 = \text{height}(X),$$

and the height of all nodes in the $Y_2, \ldots, Y_k$ subtrees is at least $s_1 + 1$. We have proved *the vertex X is the only centroid of a free tree if and only if*

$$s_j \leq s_1 + \cdots + s_k - s_j, \qquad \text{for} \qquad 1 \leq j \leq k. \tag{7}$$

Therefore the number of free trees with n vertices, having only one centroid, is the number of oriented trees with n vertices minus the number of oriented trees violating condition (7); the latter consist of an oriented tree with s_j vertices and another oriented tree with $n - s_j \leq s_j$ vertices. The number with one centroid therefore comes to

$$a_n - a_1 a_{n-1} - a_2 a_{n-2} - \cdots - a_{\lfloor n/2 \rfloor} a_{\lceil n/2 \rceil}. \tag{8}$$

A free tree with two centroids has an even number of vertices, and the height of each centroid is $n/2$ (see exercise 10). So if $n = 2m$, the number of bicen-

troidal free trees is the number of choices of 2 things out of a_m with repetition, namely

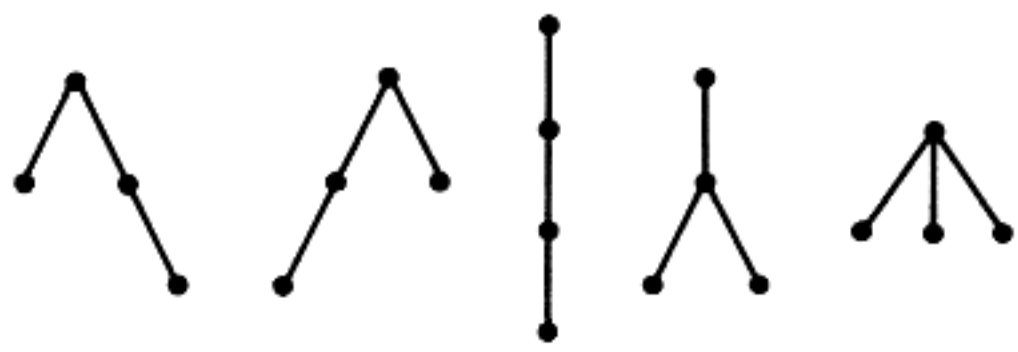
$$\binom{a_m + 1}{2}.$$

Thus, to get the total number of free trees, we add $\frac{1}{2}a_{n/2}(a_{n/2} + 1)$ to (8) when n is even. The form of Eq. (8) suggests a simple generating function, and, indeed, we find without difficulty that *the generating function for the number of structurally different free trees is*

$$\begin{aligned} F(z) &= A(z) - \tfrac{1}{2}A(z)^2 + \tfrac{1}{2}A(z^2) \\ &= z + z^2 + z^3 + 2z^4 + 3z^5 + 6z^6 + 11z^7 + 23z^8 \\ &\quad + 47z^9 + 106z^{10} + 235z^{11} + \cdots. \end{aligned} \tag{9}$$

This simple relation between $F(z)$ and $A(z)$ was first obtained by R. Otter in 1948.

Now let us turn to the question of enumerating *ordered trees*, which are our principal concern with respect to computer programming algorithms. There are five structurally different ordered trees with four vertices:



(10)

[The first two of these are identical as oriented trees, so only one of them appears in (1) above.]

Before we examine the number of different ordered tree structures, let us first consider the case of *binary trees*, since this is closer to actual computer representation and it is easier to study. Let b_n be the number of different binary trees with n nodes. From the definition of binary tree it is apparent that $b_0 = 1$, and for $n > 0$ the number of possibilities is the number of ways to put a binary tree with k nodes to the left of the root and another with $n - 1 - k$ nodes to the right. So

$$b_n = b_0 b_{n-1} + b_1 b_{n-2} + \cdots + b_{n-1} b_0, \qquad n \geq 1. \tag{11}$$

From this relation it is clear that the generating function

$$B(z) = b_0 + b_1 z + b_2 z^2 + \cdots$$

satisfies the equation

$$zB(z)^2 = B(z) - 1.$$

Solving this quadratic equation and using the fact that $B(0) = 1$, we obtain

$$\begin{aligned} B(z) &= \frac{1}{2z}\,(1 - \sqrt{1 - 4z}) \\ &= \frac{1}{2z}\left(1 - \sum_{n\geq 0}\binom{\frac{1}{2}}{n}(-4z)^n\right) \\ &= \sum_{m\geq 0}\binom{\frac{1}{2}}{m+1}(-1)^m 2^{2m+1} z^m \\ &= 1 + z + 2z^2 + 5z^3 + 14z^4 + 42z^5 + 132z^6 + 429z^7 \\ &\qquad + 1430z^8 + 4862z^9 + 16796z^{10} + \cdots . \end{aligned} \tag{12}$$

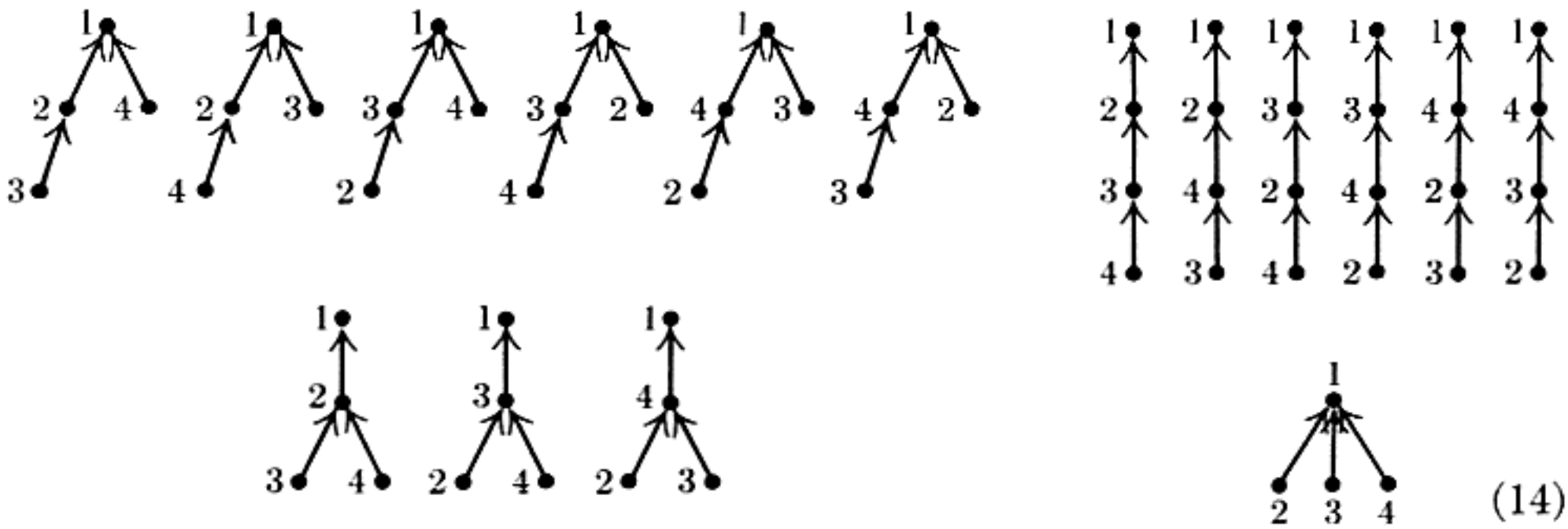
The desired answer is therefore

$$b_n = \binom{\frac{1}{2}}{n+1}(-1)^n 2^{2n+1} = \frac{1}{n+1}\binom{2n}{n}. \tag{13}$$

By Stirling's approximation, this is asymptotically $4^n/n\sqrt{\pi n} + O(4^n n^{-5/2})$.

Returning to our question about ordered trees with n nodes, we can see that this is essentially the same question as the number of binary trees, since we have a standard correspondence between binary trees and forests, and a tree minus its root is a forest. Hence *the number of (ordered) trees with n vertices is b_{n-1}, the number of binary trees with $n - 1$ vertices.*

The enumerations performed above assume that the vertices are indistinguishable points. If we label the vertices 1, 2, 3, 4 in (1) and insist that 1 is to be the root, we now get 16 different oriented trees:



(14)

The question of enumeration for labeled trees is clearly quite different from the one solved above. In this case it can be rephrased as follows: "Consider drawing lines pointing from each of the vertices 2, 3, and 4 to another vertex; there are three choices of lines emanating from each vertex, so there are $3^3 = 27$ possibilities in all. How many of these 27 ways will yield oriented trees with 1 as the root?" The answer, as we have seen, is 16. A similar reformulation of the

same problem, this time for the case of n vertices, is the following: "Let $f(x)$ be an integer-valued function such that $f(1) = 1$ and $1 \le f(x) \le n$ for all integers $1 \le x \le n$. We call f a 'tree function' if $f^n(x)$, that is, $f(f(\cdots(f(x))\cdots))$ iterated n times, equals 1, for all x. How many tree functions are there?" This problem comes up, for example, in connection with random number generation. We will find, rather surprisingly, that on the average exactly one out of every n such functions f is a tree function.

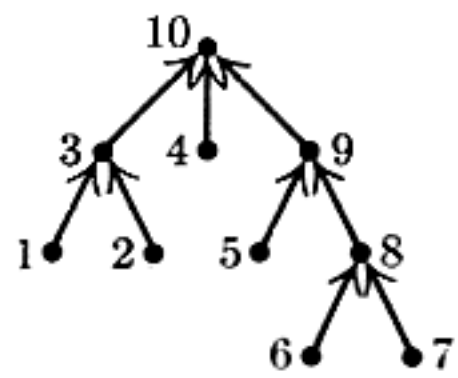
The solution to this enumeration problem can readily be derived using the general formulas for counting subtrees of graphs that have been developed in previous sections (see exercise 12). But there is a much more informative way to solve the problem, because it gives us a new and compact manner to represent oriented tree structure.

Let us suppose we are given an oriented tree with vertices $\{1, 2, \ldots, n\}$ and with $n - 1$ arcs, where the arcs go from j to $f(j)$ for all j except the root. There is at least one terminal vertex (leaf); let V_1 be the smallest number of a terminal vertex. If $n > 1$, write down $f(V_1)$ and delete both V_1 and the arc from V_1 to $f(V_1)$ from the tree; and then let V_2 be the smallest number whose vertex is terminal in the resulting tree. If $n > 2$, write down $f(V_2)$ and delete both V_2 and the arc from V_2 to $f(V_2)$ from the tree; and proceed in this way until all vertices have been deleted except the root. The resulting sequence of $n - 1$ numbers,

$$f(V_1), f(V_2), \ldots, f(V_{n-1}), \tag{15}$$

with $1 \le f(V_j) \le n$, is called the *canonical representation* of the original oriented tree.

For example, the oriented tree



(16)

with 10 vertices has the canonical representation 3, 3, 10, 10, 9, 8, 8, 9, 10.

The important point here is that we can reverse this process and go from any sequence of $n - 1$ numbers (15) back to an oriented tree which produced it. For if we have any sequence $x_1, x_2, \ldots, x_{n-1}$ of numbers between 1 and n, let V_1 be the smallest number which does not appear in the sequence $x_1, \ldots, x_{n-1}$; then let V_2 be the smallest number $\neq V_1$ which does not appear in the sequence $x_2, \ldots, x_{n-1}$; and so on. After obtaining a permutation $V_1, V_2, \ldots, V_n$ of the integers $1, 2, \ldots, n$ in this way, draw arcs from vertex V_j to vertex x_j, for $1 \le j < n$. This gives a construction of a directed graph with no oriented cycles, and by exercise 2.3.4.2–7 it is an oriented tree. Clearly, the sequence $x_1, x_2, \ldots, x_{n-1}$ is the same as the sequence (15) for this oriented tree.

Since the process is reversible, we have obtained a one-to-one correspondence between $(n-1)$-tuples of numbers $\{1, 2, \ldots, n\}$ and oriented trees on these vertices. Hence *there are n^{n-1} distinct oriented trees with n labeled vertices.* If we specify that one vertex is to be the root, there is clearly no difference between one vertex and another, so there are n^{n-2} distinct oriented trees on $\{1, 2, \ldots, n\}$ having a given root. This accounts for the $16 = 4^{4-2}$ trees in (14). From this information it is easy to determine the number of *free trees* with labeled vertices (see exercise 22). The number of *ordered trees* with labeled vertices is also easy to determine once we know the answer to this problem when no labels are involved (see exercise 23). So the problems of enumerating the three fundamental classes of trees, with both labeled and unlabeled vertices, have now been essentially resolved in this section.

It is interesting to see what would happen if we were to apply our usual method of generating functions to the problem of enumerating labeled oriented trees. For this purpose we would probably find it easiest to consider the quantity $r(n, q)$, the number of labeled directed graphs with n vertices, with no oriented cycles, and with one arc emanating from each of q designated vertices. The number of labeled oriented trees with a specified root is therefore $r(n, n-1)$. In this notation we find by simple counting arguments that, for fixed m,

$$r(n, q) = \sum_{\substack{k,t \\ t-k=m}} \binom{q}{k} r(t, k) r(n-t, q-k), \qquad \text{if} \qquad 0 < m < n-q, \tag{17}$$

$$r(n, q) = \sum_{k} \binom{q}{k} r(n-1, q-k), \qquad \text{if} \qquad q = n-1. \tag{18}$$

The first of these relations is obtained if we partition the undesignated vertices into two groups A and B, with m vertices in A and $n-q-m$ vertices in B; then the q designated vertices are partitioned into k vertices, which begin paths leading into A, and $q-k$ vertices, which begin paths leading into B. Relation (18) is obtained by considering oriented trees in which the root has degree k.

The form of these relations indicates that we can work profitably with the generating function

$$G_m(z) = r(m, 0) + r(m+1, 1)z + \frac{r(m+2, 2)z^2}{2!} = \cdots = \sum_k \frac{r(k+m, k)z^k}{k!}.$$

In these terms Eq. (17) says that $G_{n-q}(z) = G_m(z)G_{n-q-m}(z)$, and therefore by induction on m, we find that $G_m(z) = G_1(z)^m$. Now from Eq. (18), we obtain

$$G_1(z) = \sum_{n \geq 1} \frac{r(n, n-1)z^{n-1}}{(n-1)!} = \sum_{k \geq 0} \sum_{n \geq 1} \frac{r(n-1, n-1-k)z^{n-1}}{k!}$$

$$= \sum_{k \geq 0} \frac{z^k}{k!} G_k(z) = \sum_{k \geq 0} \frac{(zG_1(z))^k}{k!} = e^{zG_1(z)}.$$

In other words, putting $G_1(z) = w$, the solution to our problem comes from the coefficients of the solution to the transcendental equation

$$w = e^{zw}. \tag{19}$$

This equation can be solved with the use of Lagrange's inversion formula, i.e., $z = \zeta/f(\zeta)$ implies that

$$\zeta = \sum_{n \geq 1} \frac{z^n}{n!} g_n^{(n-1)}(0),$$

where $g_n(\zeta) = f(\zeta)^n$, when f is analytic in the neighborhood of the origin, and $f(0) \neq 0$ (see exercise 33). In this case, we may set $\zeta = zw$, $f(\zeta) = e^{\zeta}$, and we deduce the solution

$$w = \sum_{n \geq 0} \frac{(n+1)^{n-1}}{n!} z^n, \tag{20}$$

in agreement with the answer obtained above.

G. N. Raney has shown that we can extend this method in an important way to obtain an explicit power series for the solution to the considerably more general equation

$$w = y_1 e^{z_1 w} + y_2 e^{z_2 w} + \cdots + y_s e^{z_s w},$$

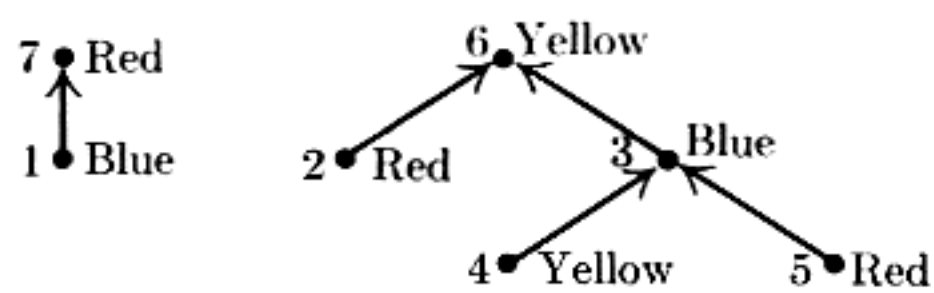
solving for w in terms of a power series in $y_1, \ldots, y_s$ and $z_1, \ldots, z_s$. For this generalization, let us consider s-dimensional vectors of integers

$$\mathbf{n} = (n_1, n_2, \ldots, n_s),$$

and let us write for convenience

$$\sum \mathbf{n} = n_1 + n_2 + \cdots + n_s.$$

Suppose that we have s "colors" $C_1, C_2, \ldots, C_s$, and consider directed graphs in which each vertex is assigned a color, e.g.,



(21)

Let $r(\mathbf{n}, \mathbf{q})$ be the number of ways to draw arcs and to assign colors to the vertices $\{1, 2, \ldots, n\}$, such that

i) for $1 \leq i \leq s$ there are exactly n_i vertices of color C_i (hence $n = \sum \mathbf{n}$);
ii) there are q arcs, one leading from each of the vertices $\{1, 2, \ldots, q\}$;
iii) for $1 \leq i \leq s$ there are exactly q_i arcs leading to vertices of color C_i (hence $q = \sum \mathbf{q}$);
iv) there are no oriented cycles (hence $q < n$).

Let us call this an $(\mathbf{n}, \mathbf{q})$-construction.

For example, if C_1 = red, C_2 = yellow, and C_3 = blue, then (21) shows a $((3, 2, 2), (1, 2, 2))$-construction. When there is only one color, we have the oriented tree problem which we have already solved.

Let $\mathbf{n}$ and $\mathbf{q}$ be fixed s-place vectors of nonnegative integers, and let $n = \sum\mathbf{n}$, $q = \sum\mathbf{q}$. For each $(\mathbf{n}, \mathbf{q})$-construction and each number k, $1 \leq k \leq n$, we will define a *canonical representation* consisting of four things:

a) a number t, with $q < t \leq n$;
b) a sequence of n colors, with n_i of color C_i;
c) a sequence of q colors, with q_i of color C_i;
d) for $1 \leq i \leq s$, a sequence of q_i elements of the set $\{1, 2, \ldots, n_i\}$.

The canonical representation is defined as follows: First list the vertices $\{1, 2, \ldots, q\}$ in the order $V_1, V_2, \ldots, V_q$ of the canonical representation of oriented trees (as given above), and then write below vertex V_j the number $f(V_j)$ of the vertex on the arc leading from V_j. Let $t = f(V_q)$; and let the sequence (c) of colors be the respective colors of the vertices $f(V_1), \ldots, f(V_q)$. Let the sequence (b) of colors be the respective colors of the vertices $k, k+1, \ldots, n, 1, \ldots, k-1$. Finally, let the ith sequence in (d) be $x_{i1}, x_{i2}, \ldots, x_{iq_i}$, where $x_{ij} = m$ if the jth C_i-colored element of the sequence $f(V_1), \ldots, f(V_q)$ is the mth C_i-colored element of the sequence $k, k+1, \ldots, n, 1, \ldots, k-1$.

For example, consider construction (21) and let $k = 3$. We start by listing $V_1, \ldots, V_5$ and $f(V_1), \ldots, f(V_5)$ below them as follows:

1	2	4	5	3
7	6	3	3	6

Hence $t = 6$, and sequence (c) represents the respective colors of 7, 6, 3, 3, 6, namely red, yellow, blue, blue, yellow. Sequence (c) represents the respective colors of 3, 4, 5, 6, 7, 1, 2, namely blue, yellow, red, yellow, red, blue, red. Finally, to get the sequences in (d), proceed as follows:

color	*elements this color in* 3, 4, 5, 6, 7, 1, 2	*elements this color in* 7, 6, 3, 3, 6	*encode column 3 by column 2*
red	5, 7, 2	7	2
yellow	4, 6	6, 6	2, 2
blue	3, 1	3, 3	1, 1

Hence the (d) sequences are 2; 2, 2; and 1, 1.

From the canonical representation, we can recover both the original $(\mathbf{n}, \mathbf{q})$-construction and the number k as follows: From (a) and (c) we know the color of vertex t. The last element of the (d) sequence for this color tells us, in conjunction with (b), the position of t in the sequence $k, \ldots, n, 1, \ldots, k-1$; hence we know k and the colors of all vertices. Then the subsequences in (d) together with (b) and (c) determine $f(V_1), f(V_2), \ldots, f(V_q)$, and finally the directed graph is reconstructed by locating $V_1, \ldots, V_q$ as we did for oriented trees.

The reversibility of this canonical representation allows us to count the number of possible $(\mathbf{n}, \mathbf{q})$-constructions, since there are $n - q$ choices for (a), the multinomial coefficient

$$\binom{n}{n_1, \ldots, n_s}$$

choices for (b),

$$\binom{q}{q_1, \ldots, q_s}$$

choices for (c), and $n_1^{q_1} n_2^{q_2} \cdots n_s^{q_s}$ choices for (d). Dividing by the n choices for k, we have the general result

$$r(\mathbf{n}, \mathbf{q}) = \frac{n - q}{n} \frac{n!}{n_1! \cdots n_s!} \frac{q!}{q_1! \cdots q_s!} n_1^{q_1} n_2^{q_2} \cdots n_s^{q_s}. \tag{22}$$

Furthermore, we can derive analogs of Eqs. (17) and (18):

$$r(\mathbf{n}, \mathbf{q}) = \sum_{\substack{\mathbf{k},\mathbf{t} \\ \Sigma(\mathbf{t}-\mathbf{k})=m}} \binom{\Sigma\mathbf{q}}{\Sigma\mathbf{k}} r(\mathbf{t}, \mathbf{k}) r(\mathbf{n} - \mathbf{t}, \mathbf{q} - \mathbf{k}), \quad \text{if} \quad 0 < m < \Sigma(\mathbf{n} - \mathbf{q}), \tag{23}$$

with the convention that $r(\mathbf{0}, \mathbf{0}) = 1$ and $r(\mathbf{n}, \mathbf{q}) = 0$ if any n_i or q_i is negative or if $q > n$;

$$r(\mathbf{n}, \mathbf{q}) = \sum_{1 \le i \le s} \sum_{k} \binom{\Sigma\mathbf{q}}{k} r(\mathbf{n} - \mathbf{e}_i, \mathbf{q} - k\mathbf{e}_i), \quad \text{if} \quad \Sigma\mathbf{n} = 1 + \Sigma\mathbf{q}, \tag{24}$$

where $\mathbf{e}_i$ is the vector with 1 in position i and zeros elsewhere. Relation (23) is based on breaking the vertices $\{q + 1, \ldots, n\}$ into two parts having m and $n - q - m$ elements, respectively; the second relation is derived by removing the unique root and considering the remaining structure. We now obtain the following result:

Theorem R (George N. Raney, *Canadian J. Math.* **16** (1964), 755–762). *Let*

$$w = \sum_{\substack{\mathbf{n},\mathbf{q} \\ \Sigma(\mathbf{n}-\mathbf{q})=1}} \frac{r(\mathbf{n}, \mathbf{q})}{(\Sigma\mathbf{q})!} y_1^{n_1} \cdots y_s^{n_s} z_1^{q_1} \cdots z_s^{q_s}, \tag{25}$$

where $r(\mathbf{n}, \mathbf{q})$ *is defined by* (22), *and where* $\mathbf{n}$, $\mathbf{q}$ *are* s*-dimensional integer vectors. Then* w *satisfies the identity*

$$w = y_1 e^{z_1 w} + y_2 e^{z_2 w} + \cdots + y_s e^{z_s w}. \tag{26}$$

Proof. By (23) and induction on m, we find that

$$w^m = \sum_{\substack{\mathbf{n},\mathbf{q} \\ \Sigma(\mathbf{n}-\mathbf{q})=m}} \frac{r(\mathbf{n}, \mathbf{q})}{(\Sigma\mathbf{q})!} y_1^{n_1} \cdots y_s^{n_s} z_1^{q_1} \cdots z_s^{q_s}. \tag{27}$$

Now by (24),

$$w = \sum_{1\le i\le s} \sum_{k} \sum_{\substack{\mathbf{n},\mathbf{q} \\ \Sigma(\mathbf{n}-\mathbf{q})=1}} \frac{r(\mathbf{n}-\mathbf{e}_i,\ \mathbf{q}-k\mathbf{e}_i)}{k!(\Sigma\mathbf{q}-k)!}\, y_1^{n_1}\cdots y_s^{n_s} z_1^{q_1}\cdots z_s^{q_s}$$

$$= \sum_{1\le i\le s} \sum_{k} \frac{1}{k!}\, y_i z_i^k \sum_{\substack{\mathbf{n},\mathbf{q} \\ \Sigma(\mathbf{n}-\mathbf{q})=k}} \frac{r(\mathbf{n},\ \mathbf{q})}{(\Sigma\mathbf{q})!}\, y_1^{n_1}\cdots y_s^{n_s} z_1^{q_1}\cdots z_s^{q_s}$$

$$= \sum_{1\le i\le s} \sum_{k} \frac{1}{k!}\, y_i z_i^k w^k. \quad \blacksquare$$

A survey of enumeration formulas for trees, based on skillful manipulations of generating functions, has been given by I. J. Good [*Proc. Cambridge Philos. Soc.* **61** (1965), 499–517; **64** (1968), 489]. This important paper contains extensive generalizations of many of the formulas derived in this section.

EXERCISES

1. [*M20*] (G. Polya.) Show that

$$A(z) = z \cdot \exp\left(A(z) + \tfrac{1}{2}A(z^2) + \tfrac{1}{3}A(z^3) + \cdots\right).$$

[*Hint:* Take logarithms of (3).]

2. [*HM24*] (R. Otter.) Show that the numbers a_n satisfy the following condition:

$$na_{n+1} = a_1 s_{n1} + 2a_2 s_{n2} + \cdots + na_n s_{nn},$$

where

$$s_{nk} = \sum_{1 \le j \le n/k} a_{n+1-jk}.$$

(These formulas are useful for the calculation of the a_n, since $s_{nk} = s_{(n-k)k} + a_{n+1-k}$.)

3. [*M40*] Write a computer program which determines the number of (unlabeled) free trees and of oriented trees with n vertices, for $n \le 100$. (Use the result of exercise 2.) Explore arithmetical properties of these numbers; can anything be said about their prime factors, or their residues modulo p?

▶ **4.** [*HM39*] (R. Otter, *Ann. Math.* **49** (1948), 583–599.) Using complex variable theory, determine the asymptotic value of the number of oriented trees as follows: (See exercise 1). (a) Show that there is a real number α between 0 and 1 for which $A(z)$ has radius of convergence α and $A(z)$ converges absolutely for all complex z such that $|z| \le \alpha$, having maximum value $A(\alpha) = a < \infty$. [*Hint:* When a power series has nonnegative coefficients, it either is entire or it has a positive real singularity; and show that $A(z)/z$ is bounded as $z \to \alpha-$.] (b) Let

$$F(z, w) = \exp\left(zw + \tfrac{1}{2}A(z^2) + \tfrac{1}{3}A(z^3) + \cdots\right) - w.$$

Show that in a neighborhood of $(z, w) = (\alpha, a/\alpha)$, $F(z, w)$ is analytic in each variable separately. (c) Show that at the point $(z, w) = (\alpha, a/\alpha)$, $\partial F/\partial w = 0$; hence $a = 1$. (d) At the point $(z, w) = (\alpha, 1/\alpha)$ show that

$$\frac{\partial F}{\partial z} = \beta = \alpha^{-2} + \sum_{k\geq 2} \alpha^{k-2} A'(\alpha^k), \qquad \text{and} \qquad \frac{\partial^2 F}{\partial w^2} = \alpha.$$

(e) When $|z| = \alpha$ and $z \neq \alpha$, show that $\partial F/\partial w \neq 0$; hence $A(z)$ has only one singularity on $|z| = \alpha$. (f) Prove there is a region larger than $|z| < \alpha$ in which

$$\frac{1}{z} A(z) = \frac{1}{\alpha} - \sqrt{2\beta(1 - z/\alpha)} + R(z),$$

where $R(z)$ is an analytic function of $\sqrt{z - \alpha}$. (g) Prove that consequently

$$a_n = \frac{1}{\alpha^{n-1} n} \sqrt{\beta/2\pi n} + O(n^{-5/2}\alpha^{-n}).$$

(*Note:* $1/\alpha = 2.95576$, and $\alpha\sqrt{\beta/2\pi} = 0.43992$.)

▸ **5.** [*M25*] (A. Cayley.) Let c_n be the number of (unlabeled) oriented trees having n leaves (i.e., vertices with in-degree zero) and having at least two subtrees at every other vertex. Thus $c_3 = 2$, by virtue of the two trees

Find a formula analogous to (3) for the generating function

$$C(z) = \sum_n c_n z^n.$$

6. [*M25*] Let an "oriented binary tree" be an oriented tree in which each vertex has in-degree two or less. Find a reasonably simple relation which defines the generating function $G(z)$ for the number of distinct oriented binary trees with n vertexes, and find the first few values.

7. [*HM40*] Obtain asymptotic values for the numbers of exercise 6. (See exercise 4.)

8. [*20*] According to Eq. (9), there are six free trees with six vertices. Draw them, and indicate their centroids.

9. [*M20*] From the fact that at most one subtree of a vertex in a free tree can contain a centroid, prove there are at most two centroids in a free tree, and if there are two, then they are adjacent.

▸ **10.** [*M22*] Prove that a free tree with n vertices and two centroids consists of two free trees with $n/2$ vertices, joined by an edge. Conversely, if two free trees with m vertices are joined by an edge, we obtain a free tree with $2m$ vertices and two centroids.

▸ **11.** [*M28*] The text derives the number of different binary trees with n nodes (Eq. 13). Generalize this to find the number of different t-ary trees with n nodes. (Cf. exercise 2.3.1; a t-ary tree is either empty or consists of a root and t disjoint t-ary trees.) [*Hint:* Use Eq. (21) of Section 1.2.9.]

12. [*M20*] Find the number of labeled oriented trees with n vertices by using determinants and the result of exercise 2.3.4.2–19. (See also exercise 1.2.3–36.)

13. [*15*] What oriented tree on the vertices 1, 2, . . . , 10 has the canonical representation 3, 1, 4, 1, 5, 9, 2, 6, 5?

14. [*10*] True or false: The last entry, $f(V_{n-1})$, in the canonical representation of an oriented tree, is always the root of that tree.

15. [*21*] Discuss the relationships that exist (if any) between the topological sort algorithm of Section 2.2.3 and the canonical representation of an oriented tree.

16. [*25*] Design an algorithm (as efficient as possible) which converts from the canonical representation of an oriented tree to a conventional computer representation using "FATHER" links.

▸ **17.** [*M26*] Let $f(x)$ be an integer-valued function, where $1 \le f(x) \le m$ for all integers $1 \le x \le m$. Define $x \equiv y$ if $f^r(x) = f^s(y)$ for some $r, s \ge 0$, where $f^0(x) = x$ and $f^{r+1}(x) = f(f^r(x))$. By using methods of enumeration like those in this section, show that the number of functions such that $x \equiv y$ for all x and y is $m^{m-1}Q(m)$, where $Q(m)$ is the function defined in Section 1.2.11.3. (For related results, see exercise 3.1–14.)

18. [*24*] Show that the following method is another way to define a one-to-one correspondence between $(n-1)$-tuples of numbers from 1 to n and oriented trees with n labeled vertices: Let the leaves of the tree be $V_1, \ldots, V_k$ in ascending order. Let $(V_1, V_{k+1}, V_{k+2}, \ldots, V_\ell)$ be the path from V_1 to the root, and write down the vertices $V_\ell, \ldots, V_{k+2}, V_{k+1}$. Then let $(V_2, V_{\ell+1}, V_{\ell+2}, \ldots, V_r)$ be the shortest oriented path from V_2 such that V_r has already been written down, and write down $V_r, \ldots, V_{\ell+2}, V_{\ell+1}$. Then let $(V_3, V_{r+1}, \ldots, V_s)$ be the shortest oriented path from V_3 such that V_s has already been written, and write $V_s, \ldots, V_{r+1}$; and so on. For example, the tree (16) would be encoded as 10, 3, 3, 10, 10, 9, 9, 8, 8. Show that this process is reversible, and, in particular, draw the oriented tree with vertices 1, 2, . . . , 10 and representation 3, 1, 4, 1, 5, 9, 2, 6, 5.

19. [*M24*] How many different labeled, oriented trees are there having n vertices, k of which are leaves (i.e., have in-degree zero)?

20. [*M24*] (J. Riordan.) How many different labeled, oriented trees are there having n vertices, k_0 of which have in-degree 0, k_1 have in-degree 1, k_2 have in-degree 2, . . . ? (Note that necessarily $k_0 + k_1 + k_2 + \cdots = n$, and $k_1 + 2k_2 + 3k_3 + \cdots = n - 1$.)

▸ **21.** [*M21*] Enumerate the number of labeled oriented trees in which each vertex has in-degree zero or two. (Cf. exercise 20 and exercise 2.3–20.)

22. [*M20*] How many *labeled* free trees are possible with n vertices? (In other words, if we are given n vertices, there are $2^{\binom{n}{2}}$ possible graphs having these vertices, depending on which of the $\binom{n}{2}$ possible edges are incorporated into the graph; how many of these graphs are free trees?)

23. [*M21*] How many ordered trees are possible with n labeled vertices? (Give a simple formula involving factorials.)

24. [*M16*] All labeled oriented trees with vertices 1, 2, 3, 4 and with root 1 are shown in (14). How many would there be if we listed all labeled *ordered* trees with these vertices and this root?

25. [*M20*] What is the value of the quantity $r(n, q)$ which appears in Eqs. (17) and (18)? (Give an explicit formula; the text only mentions that $r(n, n-1) = n^{n-2}$.)

26. [*20*] In terms of the notation at the end of this section, draw the ((3, 2, 4), (2, 3, 2))-construction [analogous to (21)], and find the number k, which corresponds to the canonical representation having $t = 8$, sequences of colors "red, yellow, blue, red, yellow, blue, red, blue, blue" and "red, yellow, blue, red, yellow, blue, yellow", and sequences 3, 2; 1, 2, 1; 2, 4.

▶ **27.** [*M28*] Let $U_1, U_2, \ldots, U_p, \ldots, U_q; V_1, V_2, \ldots, V_r$ be vertices of a directed graph, where $1 \le p \le q$. Let f be any function from the set $\{p+1, \ldots, q\}$ into the set $\{1, 2, \ldots, r\}$, and let the directed graph contain exactly $q - p$ arcs, from U_k to $V_{f(k)}$ for $p < k \le q$. Show that the number of ways to add r additional arcs, one from each of the V's to one of the U's, such that the resulting directed graph contains no oriented cycles, is $q^{r-1}p$. Prove this by generalizing the canonical representation method, i.e., setting up a one-to-one correspondence between all such ways of adding r further arcs and the set of all sequences of integers $a_1, a_2, \ldots, a_r$, where $1 \le a_k \le q$ for $1 \le k < r$, and $1 \le a_r \le p$.

28. [*M22*] Use the result of exercise 27 to enumerate the number of labeled free trees on vertices $U_1, \ldots, U_m, V_1, \ldots, V_n$, such that all edges go from U_j to V_k for some j and k.

29. [*HM22*] Prove that if $E_k(r, t) = r(r + kt)^{k-1}/k!$, and if $zx^t = \ln x$, then

$$x^r = \sum_k E_k(r, t) z^k$$

for sufficiently small $|z|$ and $|x - 1|$. [Use the fact that $G_m(z) = G_1(z)^m$ in the discussion following Eq. (18).] In this formula, r stands for an arbitrary real number. [*Note:* As a consequence of this formula we have the identity

$$\sum_k E_k(r, t) E_{n-k}(s, t) = E_n(r + s, t);$$

this implies Abel's binomial theorem (Eq. 16 of Section 1.2.6). Compare also Eq. (31) of that section.]

30. [*M23*] Let $n, x, y, z_1, \ldots, z_n$ be positive integers. Consider a set of $x + y + z_1 + \cdots + z_n$ vertices r_i, s_{jk}, t_j $(1 \le i \le x + y,\ 1 \le j \le n,\ 1 \le k < z_j)$, in which arcs have been drawn from s_{jk} to t_j for all j, k. According to exercise 27, there are $(x + y)(x + y + z_1 + \cdots + z_n)^{n-1}$ ways to draw arcs from $t_1, \ldots, t_n$ to other vertices such that the resulting directed graph contains no oriented cycles. Use this to prove Hurwitz's generalization of the binomial theorem:

$$\sum x(x + \epsilon_1 z_1 + \cdots + \epsilon_n z_n)^{\epsilon_1 + \cdots + \epsilon_n - 1} y\big(y + (1 - \epsilon_1)z_1 + \cdots + (1 - \epsilon_n)z_n\big)^{n-1-\epsilon_1-\cdots-\epsilon_n} = (x + y)(x + y + z_1 + \cdots + z_n)^{n-1},$$

where the sum is over all 2^n choices of $\epsilon_1, \ldots, \epsilon_n$ equal to 0 or 1.

31. [*M24*] Solve exercise 5 for ordered trees; i.e., derive the generating function for the number of unlabeled ordered trees with n terminal nodes and no nodes of degree 1.

32. [*M37*] (A. Erdélyi and I. M. H. Etherington, *Edinburgh Math. Notes* **32** (1940), 7–12.) How many (ordered, unlabeled) trees are there with n_0 nodes of degree 0, n_1 of degree 1, ..., n_m of degree m, and none of higher degree with m? (An explicit

solution to this problem can be given in terms of factorials, thereby considerably generalizing the result of exercise 11.)

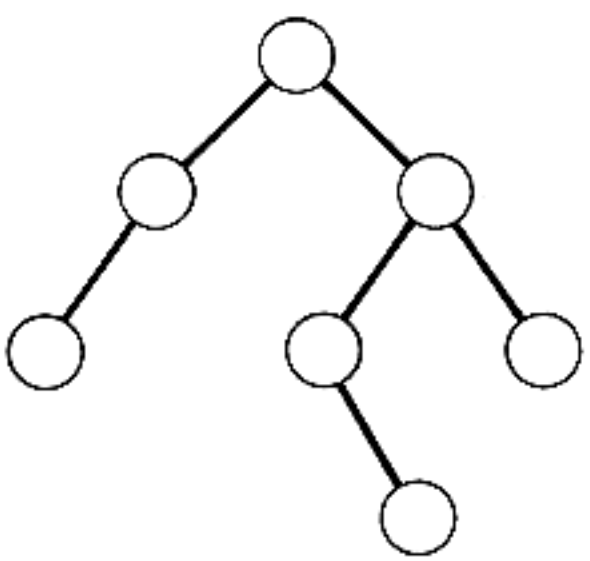
▶ **33.** [*M28*] The text gives an explicit power series solution of the equation $w = y_1 e^{z_1 w} + \cdots + y_r e^{z_r w}$, based on enumeration formulas for certain oriented forests. Similarly, show that the enumeration formula of exercise 32 leads to an explicit power series solution to the equation

$$w = z_1 w^{e_1} + z_2 w^{e_2} + \cdots + z_r w^{e_r},$$

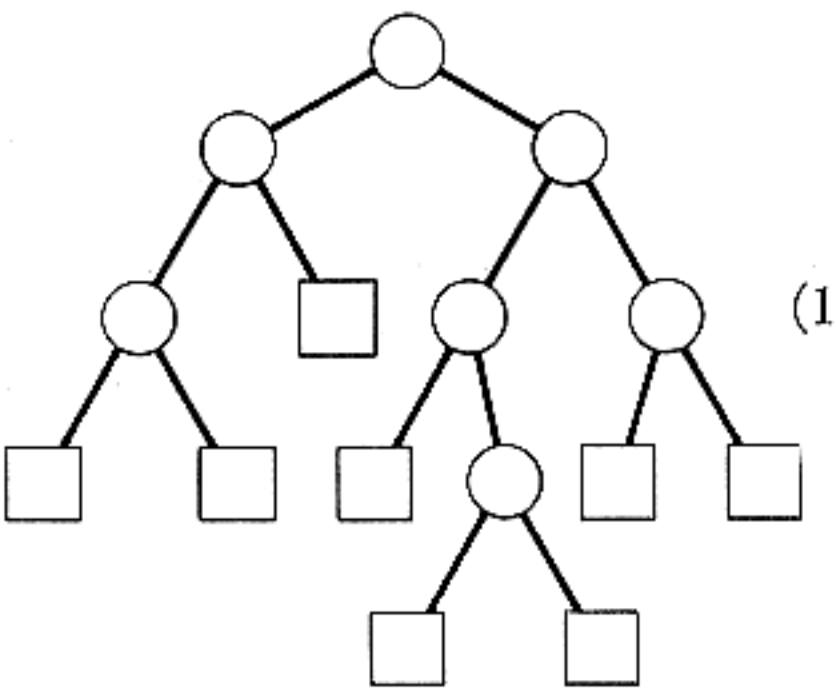
expressing w as a power series in $z_1, \ldots, z_r$. (Here $e_1, \ldots, e_r$ are fixed nonnegative integers, at least one of which is zero.)

2.3.4.5. Path length. The concept of the "path length" of a tree is of great importance in the analysis of algorithms, since this quantity is often directly related to the execution time. Our primary concern is with binary trees, since this is so close to the computer representations.

In the following discussion let us extend each binary tree diagram by adding special nodes wherever a null subtree was present in the original tree, so that



becomes



(1)

After the square-shaped nodes have been added in this way, the structure is sometimes more convenient to deal with theoretically. It is clear that in this augmented tree, every circular node has two sons and every square node has none. (Compare with exercise 2.3–20.) If there are n circular nodes and s square nodes, we have $n + s - 1$ edges (since the diagram is a free tree), and, counting another way, by the number of sons, we see there are $2n$ edges. Hence it is clear that

$$s = n + 1; \tag{2}$$

i.e., the number of "external" nodes just added is one more than the number of "internal" nodes we had originally. (For another proof, see exercise 2.3.1–14.) Formula (2) is correct even when $n = 0$.

Assume that a binary tree has been extended in this way. The *external path length of the tree*, E, is defined to be the sum—taken over all terminal (square) nodes—of the lengths of the paths from the root to each node. The *internal path length*, I, is the same quantity summed over the nonterminal (circular)

nodes. In (1) the external path length is $E = 3+3+2+3+4+4+3+3 = 25$, and the internal path length is $I = 2+1+0+2+3+1+2 = 11$. These two quantities are always related by the formula

$$E = I + 2n, \tag{3}$$

where n is the number of nonterminal nodes.

To prove formula (3), consider deleting a nonterminal node V at a distance k from the root, where both sons of V are terminal. The quantity E goes down $2(k+1)$, since the sons of V are removed, and it then goes up k, since V becomes terminal, so the net change in E is $-k-2$. The net change in I is $-k$, so (3) may be proved by induction.

It is not hard to see that the internal path length (and hence the external path length also) is highest when we have a degenerate tree with linear structure; in that case the internal path length is

$$(n-1) + (n-2) + \cdots + 1 + 0 = \tfrac{1}{2}(n^2 - n).$$

It can be shown that the "average" path length over all binary trees is essentially proportional to $n\sqrt{n}$ (see exercise 5).

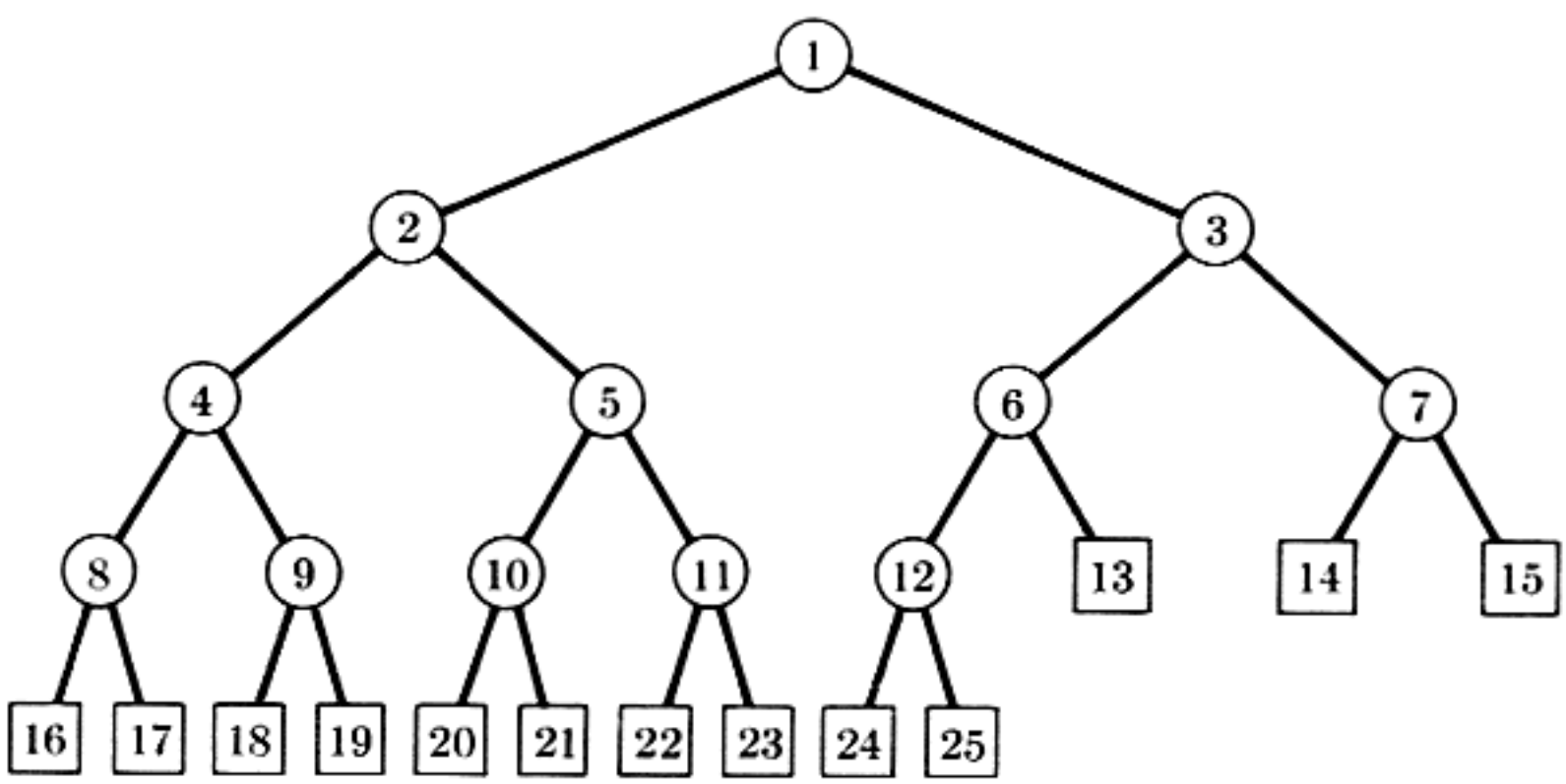
Consider now the problem of discovering a binary tree with n nodes having *minimum* path length: such a tree will be important, since it will minimize the computation time for various algorithms. Clearly, only one node (the root) can be at zero distance from the root; at most two nodes can be at distance 1 from the root, at most four can be 2 away, etc. So we see *the internal path length is always at least as big as the sum of the first n terms of the series*

$$0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, \ldots$$

This is the sum $\sum_{1\le k\le n} \lfloor \log_2 k \rfloor$, which we know from exercise 1.2.4–42 is

$$(n+1)q - 2^{q+1} + 2, \qquad q = \lfloor \log_2 (n+1) \rfloor. \tag{4}$$

Thus the optimum value (4) is essentially of the form $n \log n$; this optimum is clearly achieved in a tree which looks essentially like this (illustrated for $n = 12$):

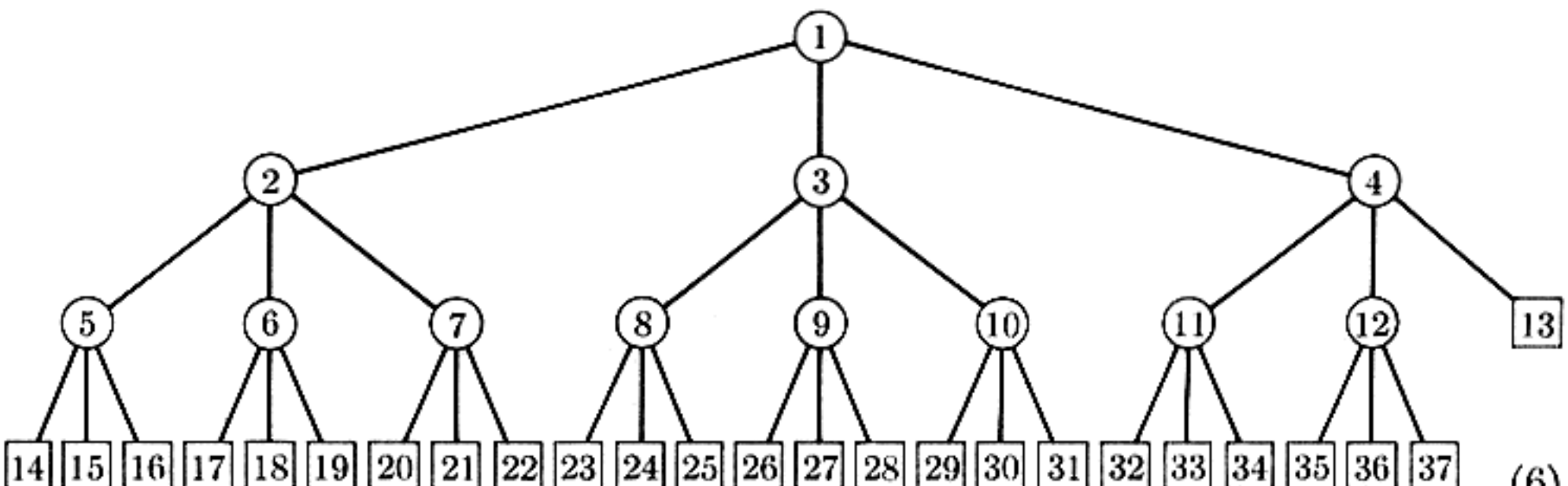


(5)

A tree such as (5) is called a *complete binary tree* with n internal nodes. In the general case we may number the nodes $1, 2, \ldots, n$; this numbering has the useful property that the father of node k is node $\lfloor k/2 \rfloor$; the sons of node k are nodes $2k$ and $2k + 1$. The terminal nodes are numbers $n + 1$ through $2n + 1$, inclusive.

It follows that a complete binary tree may be simply represented in sequential memory locations, with the structure implicit in the locations of the nodes. The complete binary tree appears explicitly or implicitly in many important computer algorithms, so the reader should give it special attention.

These concepts have important generalizations to ternary, quaternary, etc. trees. We define a *t-ary tree* as a set of nodes which is either empty or consists of a root and t ordered, disjoint t-ary trees. (Cf. the definition of binary tree in Section 2.3.) The *complete ternary tree* with 12 internal nodes is



(6)

It is easy to see how this generalizes to the complete t-ary tree with the internal nodes $1, 2, \ldots, n$: the father of node k is node

$$\lfloor (k + t - 2)/t \rfloor = \lceil (k - 1)/t \rceil,$$

and the sons of node k are

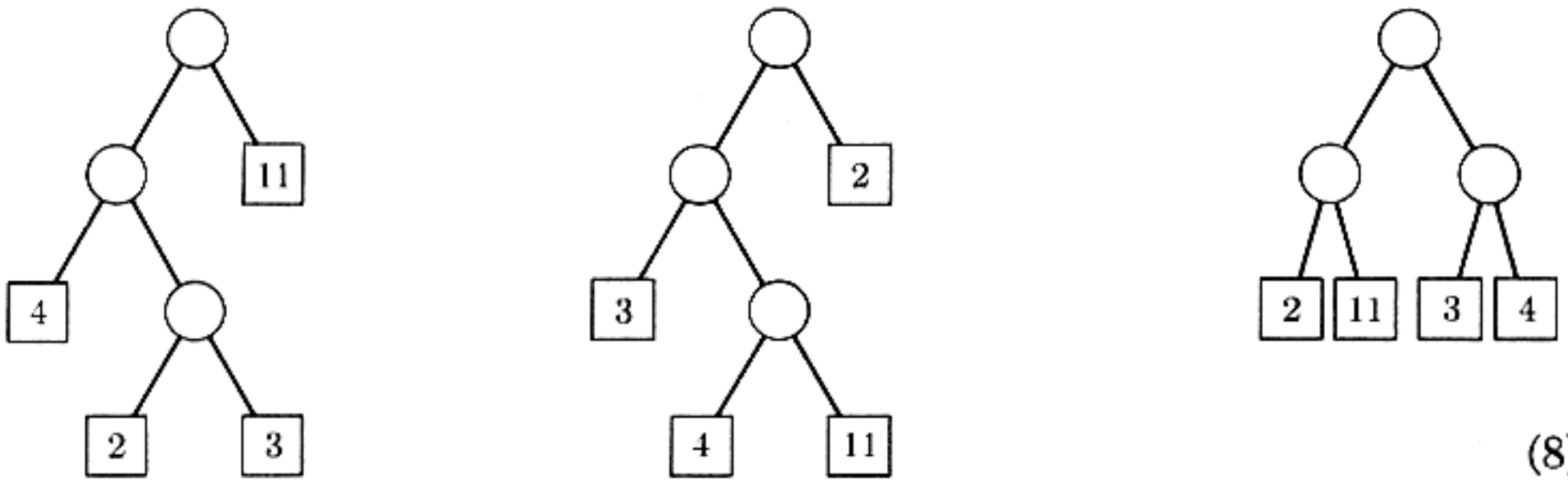
$$t(k - 1) + 2, \qquad t(k - 1) + 3, \qquad \ldots, \qquad tk + 1.$$

This tree has the minimum internal path length among all t-ary trees with n internal nodes: its internal path length is (see exercise 8)

$$\left(n + \frac{1}{t-1}\right) q - \frac{(t^{q+1} - t)}{(t-1)^2}, \qquad q = \lfloor \log_t ((t-1)n + 1) \rfloor. \tag{7}$$

These results have another important generalization if we shift our point of view slightly. Suppose that we are given m real numbers $w_1, w_2, \ldots, w_m$; the problem is to find an extended binary tree with m terminal nodes, and to associate the numbers $w_1, \ldots, w_m$ with these nodes, in such a way that the sum $\sum w_j l_j$ is minimized, where l_j is the length of path from the root and the sum is taken over all terminal nodes. For example, if the given numbers are 2, 3, 4, 11,

we can form extended binary trees such as these three:

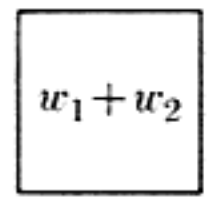


(8)

Here the "weighted" path lengths $\sum w_j l_j$ are 34, 53, and 40, respectively. (Note that a perfectly balanced tree does *not* give the minimum weighted path length when the weights are 2, 3, 4, and 11, although we have seen that it does give the minimum in the special case $w_1 = w_2 = \cdots = w_m = 1$.)

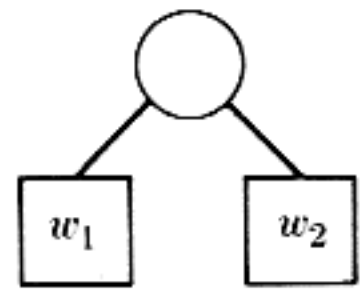
There are several interpretations of weighted path length in connection with different computer algorithms; for example, we can apply it to the merging of sorted strings of respective lengths $w_1, w_2, \ldots, w_m$ (see Chapter 5). One of the most straightforward applications of this idea is to consider a binary tree as a general search procedure, where we start at the root and then make some test; the outcome of the test sends us to one of the two branches, where we may make further tests, etc. For example, if we want to decide which of four different alternatives is true, and if these possibilities will be true with the respective probabilities $\frac{2}{20}$, $\frac{3}{20}$, $\frac{4}{20}$, and $\frac{11}{20}$, the tree which minimizes the weighted path length constitutes an *optimal search procedure* in this case. [These are the weights shown in (8).]

An elegant algorithm for finding a tree with minimum weighted path length has been given by D. Huffman: First find the two w's of lowest value, say w_1 and w_2. Then solve the problem for $m - 1$ weights $w_1 + w_2, w_3, \ldots, w_m$, and replace the node



(9)

in this solution by



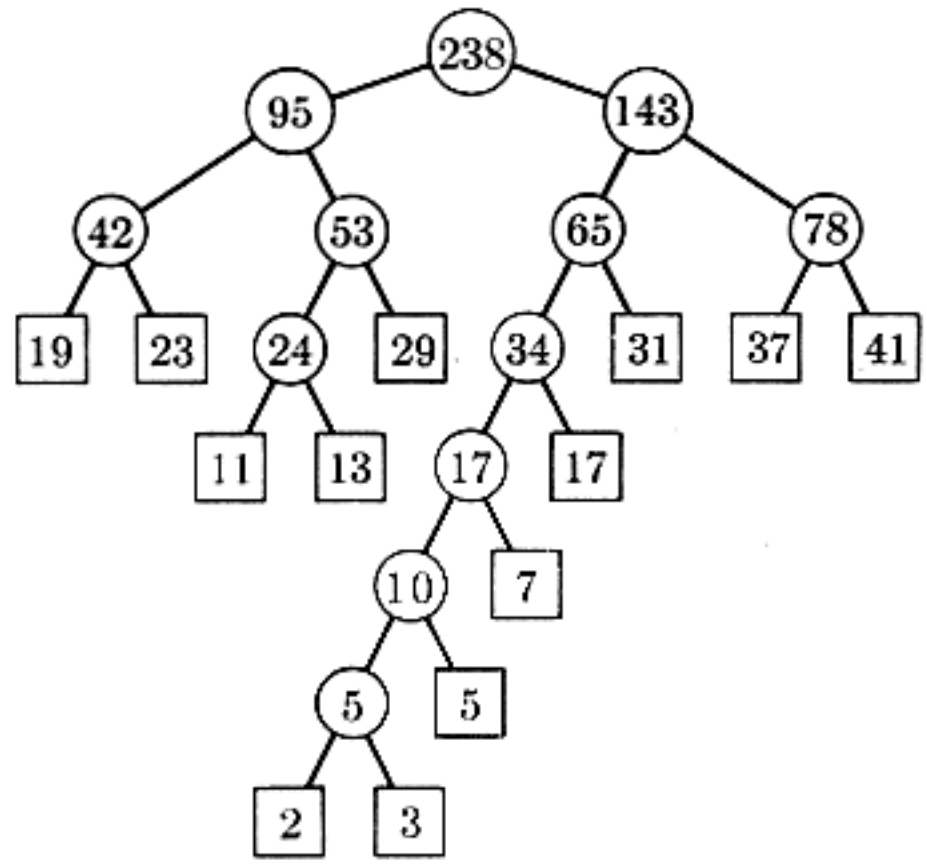
(10)

As an example of Huffman's method, let us find the optimal tree for the weights 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41. First we combine $2 + 3$, and look for the solution to $5, 5, 7, \ldots, 41$; then we combine $5 + 5$, etc. The

computation is summarized as follows:

2	3	5	7	11	13	17	19	23	29	31	37	41
	5	5	7	11	13	17	19	23	29	31	37	41
		10	7	11	13	17	19	23	29	31	37	41
			17	11	13	17	19	23	29	31	37	41
			17		24	17	19	23	29	31	37	41
					24	34	19	23	29	31	37	41
					24	34		42	29	31	37	41
						34		42	53	31	37	41
								42	53	65	37	41
								42	53	65		78
									95	65		78
									95			143
												238

Therefore the following tree corresponds to Huffman's construction:



(11)

(The numbers inside the circular nodes show the correspondence between this tree and our computation; see also exercise 9.)

It is not hard to show that this method does in fact minimize the weighted path length, by induction on m. Suppose that $m \geq 2$ and $w_1 \leq w_2 \leq w_3 \leq \cdots \leq w_m$, and suppose that we are given a tree which minimizes the weighted path length. (Such a tree certainly exists, since only finitely many binary trees with m terminal nodes are possible.) Let V be an internal node of maximum distance from the root. If w_1 and w_2 are not the weights already attached to the sons of V, we can interchange them with the values which are already there and not increase the weighted path length. Thus there is a tree which minimizes the weighted path length and which contains the subtree (10). Now it is easy to prove that the weighted path length of such a tree is minimized if and only if the tree with (10) replaced by (9) has minimum path length for the weights $w_1 + w_2, w_3, \ldots, w_m$. (See exercise 9.)

In general, there are many trees which minimize $\sum w_j l_j$. If the w's are kept in order throughout the construction, and if when w_1, w_2 are removed the quantity $w_1 + w_2$ is placed higher in the ordering than any of the other weights of the same value (i.e., between w_k and w_{k+1}, where $w_k \le w_1 + w_2 < w_{k+1}$), then the tree constructed by Huffman's method has the smallest value of $\max l_j$ and of $\sum l_j$ among all trees which minimize $\sum w_j l_j$. [See the article by Eugene S. Schwartz, *Information and Control* **7** (1964), 37–44.]

Huffman's method can be generalized to t-ary trees as well as binary trees. (See exercise 10.)

EXERCISES

1. [*12*] Are there any other binary trees with 12 internal nodes and minimum path length, besides the complete binary tree (5)?

2. [*17*] Draw an extended binary tree with terminal nodes containing the weights 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, having minimum weighted path length.

▶ **3.** [*M24*] An extended binary tree with m terminal nodes determines a set of path lengths $l_1, l_2, \ldots, l_m$ which describe the length of path from the root to the respective terminal nodes. Conversely, if we are given a set of numbers $l_1, l_2, \ldots, l_m$, is it always possible to construct an extended binary tree in which these numbers are the path lengths in some order? Show that this is possible if and only if $\sum_{1\le j\le m} 2^{-l_j} = 1$.

▶ **4.** [*M25*] (E. S. Schwartz.) Assume that $w_1 \le w_2 \le \cdots \le w_m$. Show that there is an extended binary tree which minimizes $\sum w_j l_j$ and for which the terminal nodes in left to right order contain the respective values $w_1, w_2, \ldots, w_m$. [For example, tree (11) does *not* meet this condition since the weights appear in the order 19, 23, 11, 13, 29, 2, 3, 5, 7, 17, 31, 37, 41. We seek a tree for which the weights appear in ascending order, and this does not always happen with Huffman's construction.]

5. [*HM26*] Let

$$B(w, z) = \sum_{n,p\ge 0} b_{np} w^p z^n,$$

where b_{np} is the number of binary trees with n nodes and internal path length p. [Thus,

$$B(w, z) = 1 + z + 2wz^2 + (w^2 + 4w^3)z^3 + (4w^4 + 2w^5 + 8w^6)z^4 + \cdots;$$

$B(1, z)$ is the function $B(z)$ of Eq. (12) in Section 2.3.4.4.] (a) Find a functional relation which characterizes $B(w, z)$. (b) Use the result of (a) to determine the *average internal path length* of a binary tree with n nodes, assuming that each of the

$$\frac{1}{n+1}\binom{2n}{n}$$

trees is equally probable. (c) Find the asymptotic value of this quantity.

6. [*16*] If a t-ary tree is extended with "square" nodes as in (1), what is the relation between the number of square and circular nodes corresponding to Eq. (2)?

7. [*M21*] What is the relation between external and internal path length in a t-ary tree? (Cf. exercise 6; a generalization of Eq. (3) is desired.)

8. [*M23*] Prove Eq. (7).

9. [*M21*] The numbers which appear in the circular nodes of (11) are equal to the sums of the weights in the terminal nodes of the corresponding subtree. Show that the sum of all values in the circular nodes is equal to the weighted path length.

▸ **10.** [*M24*] (D. Huffman.) Show how to construct a t-ary tree with minimum weighted path length, given weights $w_1, w_2, \ldots, w_m$. Construct an optimal ternary tree for weights 1, 4, 9, 16, 25, 36, 49, 64, 81, 100.

11. [*16*] Is there any connection between the complete binary tree (5) and the "Dewey decimal notation" for binary trees described in exercise 2.3.1–5?

2.3.4.6. History and bibliography. Trees have of course been in existence since the third day of creation and perhaps earlier, and through the ages tree structures (especially *family* trees) have been in common use. The concept of tree as a formally defined *mathematical* entity seems to have appeared first in the work of G. Kirchhoff [*Annalen der Physik und Chemie* **72** (1847), 497–508], who used trees to find a set of fundamental cycles in an electrical network in connection with the law that bears his name, essentially as we did in Section 2.3.4.1. The concept also appeared at about the same time in the book *Geometrie der Lage* (pp. 20–21) by K. G. Chr. von Staudt. The name "tree" and many results dealing mostly with enumeration of trees began to appear ten years later in a series of papers by Arthur Cayley [see *Collected Mathematical Papers of A. Cayley* **3** (1857), 242–246; **4** (1859) 114–115; **9** (1874), 202–204; **9** (1875), 427–460; **10** (1877), 598–600; **11** (1881), 365–367; **13** (1889), 26–28]. Cayley was unaware of the previous work of Kirchhoff and von Staudt; his investigations began with studies of the structure of algebraic formulas, and they were later inspired chiefly by applications to the problem of isomers in chemistry. Tree structures were also independently studied by J. B. Listing [*Göttingen Abhandlungen* **10** (1862)] and C. Jordan [*Journal f. d. reine und angewandte Math.* **70** (1869).

The infinity lemma was formulated first by Denes König [*Fundamenta Mathematicae* **8** (1926), 114–134], and he gave it a prominent place in his classic book *Theorie der endlichen und unendlichen Graphen* (Leipzig, 1936), Chapter 6. A similar result called the "fan theorem" occurred slightly earlier in the work of L. E. J. Brouwer [*Verhandelingen Akad. Amsterdam* **12** (1919), 7], but this involved much stronger hypotheses; for a discussion of Brouwer's work see A. Heyting, *Intuitionism* (1956), Section 3.4.

Formula (3) of Section 2.3.4.4 for enumerating unlabeled oriented trees was given by Cayley in his first paper on trees. In his second paper he enumerated unlabeled ordered trees; an equivalent problem had already been proposed and solved by J. de Segner and L. Euler 100 years earlier (*Novi Commentarii Academiae Scientiarum Petropolitanae* **7** (1760), 13–15, 203–209), and it was the subject of seven papers by G. Lamé, E. Catalan, O. Rodrigues, and J. Binet in *Journal de Mathématiques* **3, 4** (1838, 1839).

Cayley published the formula n^{n-2} as the number of *labeled* free trees 32 years after his first paper on the subject, and at that time he hinted at the connection between trees and $(n-1)$-tuples of numbers. (Cayley's discussion in that paper is extremely vague and can hardly be considered a proof of the formula n^{n-2}, since he merely worked out a few special cases and asserted that the same procedure would work in general, without really saying what the procedure was. There is, of course, little doubt that Cayley could have proved the formula if he had been pressed to do so.) The explicit correspondence between trees and $(n-1)$-tuples of numbers was first published by Heinz Prüfer [*Arch. Math. u. Phys.* **27** (1918), 142–144], quite independently of Cayley's prior work. An interesting bibliography, which lists over 25 papers having to do with the formula n^{n-2} and its generalizations, has been compiled by J. W. Moon, in *A Seminar on Graph Theory* (F. Harary, ed.), Chapter 11, (New York: Holt, Rinehart, and Winston, 1967), 70–78.

A very important paper on the enumeration of trees and many other kinds of enumeration problems was published by G. Polya in *Acta Math.* **68** (1937), 145–253. For discussion of enumeration problems for graphs and an excellent bibliography see the survey paper, "Unsolved Problems in the Enumeration of Graphs" by F. Harary, *Matematikai Kutató Intézetének Közleményei* (Publ. of the Math. Institute), Hungarian Academy of Sciences, **A5** (1960), 63–95.

The principle of minimizing weighted path length by repeatedly combining the smallest weights was discovered by D. Huffman [*Proc. IRE* **40** (1952), 1098–1101], in connection with the design of codes for minimizing message lengths. The same idea was independently published by Seth Zimmerman [*AMM* **66** (1959), 690–693].

Several noteworthy recent papers dealing with the theory of tree structures have been cited in Sections 2.3.4.1 through 2.3.4.5 in connection with particular topics.

For further discussion of the mathematical properties of trees, see the following references and their bibliographies:

CLAUDE BERGE, *The Theory of Graphs*, tr. by Alison Doig (London: Methuen, 1962), Chapter 16–17.

ØYSTEIN ORE, *Theory of Graphs* (Amer. Math Society, 1962), Chapter 4.

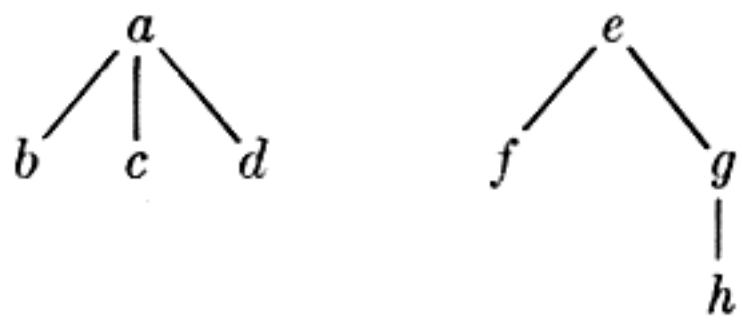
JOHN RIORDAN, *Introduction to Combinatorial Analysis* (New York: Wiley, 1958), Chapter 6.

The theoretical study of generalizations of graphs, to structures involving nodes containing several information and link fields, is still in its infancy, and it promises to be important in the future.

2.3.5. Lists and Garbage Collection

Near the beginning of Section 2.3 we defined a List as "a finite sequence of zero or more atoms or Lists."

Any forest is a List; for example,

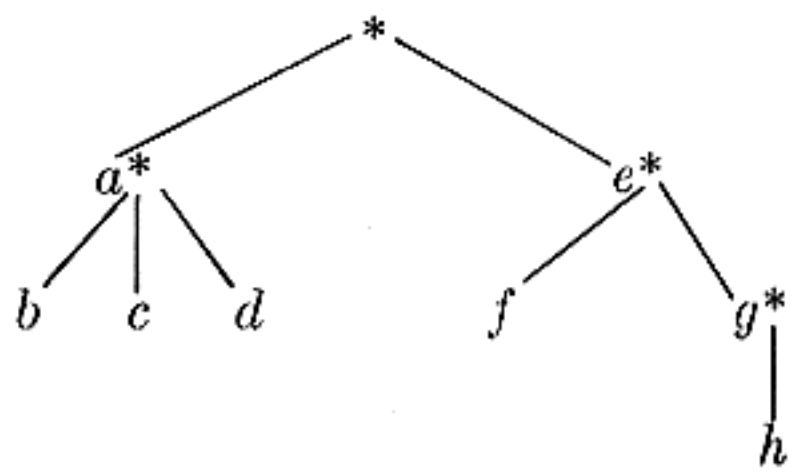


(1)

may be regarded as the List

$$(a:(b, c, d), e:(f, g:(h)), \tag{2}$$

and the corresponding List diagram would be



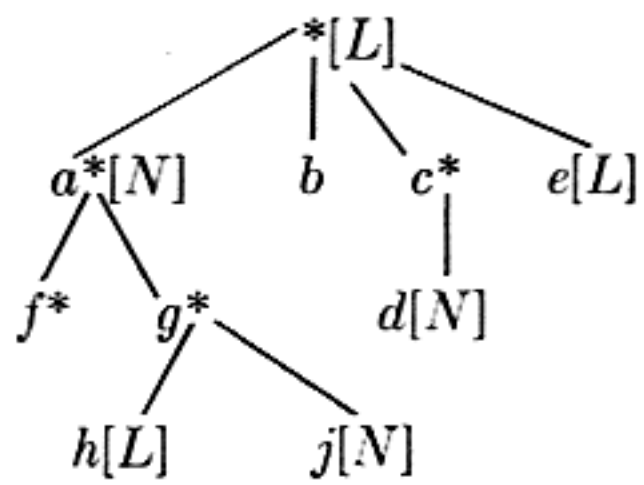
(3)

The reader should review at this point the introduction to Lists given earlier, in particular (3), (4), (5), (6), (7) in the beginning of Section 2.3. Recall that the notation "a:" which appears in (2) above means that the List (b, c, d) is "labeled" with the attribute "a" besides its structural information that it is a List of three atoms b, c, and d. This is compatible with our general convention that each node of a tree may contain information besides its structural connections. However, as was discussed for trees in Section 2.3.3, it is quite possible and sometimes desirable to insist that all Lists be unlabeled, so that all the information appears in the atoms.

Although any forest may be regarded as a List, the converse is not true. The following List is perhaps more typical than (2) and (3) since it shows how the restrictions of tree structure may be violated:

$$L = (a:N, b, c:(d:N), e:L), \qquad N = (f:(), g:(h:L, j:N)) \tag{4}$$

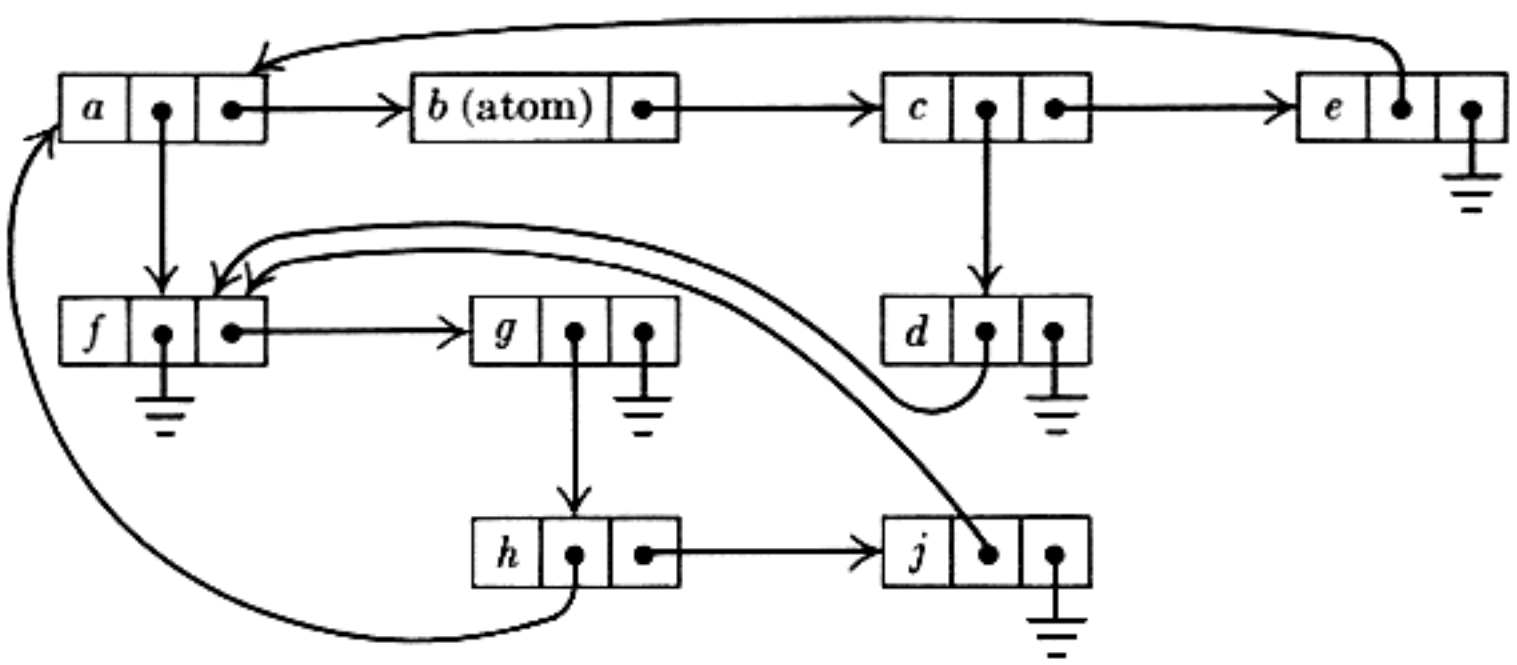
which may be diagramed as



(5)

[Cf. diagram (7) in Section 2.3.]

As we might expect, there are many ways to represent List structures within a computer memory. These are usually variations on the same basic theme according to which binary trees are used to represent general forests of trees: one field RLINK, say, is used to point to the next element of a List, and another field DLINK may be used to point to the first element of a sub-List. By a natural extension of the memory representation described in Section 2.3.2, we would represent the List (5) as follows:

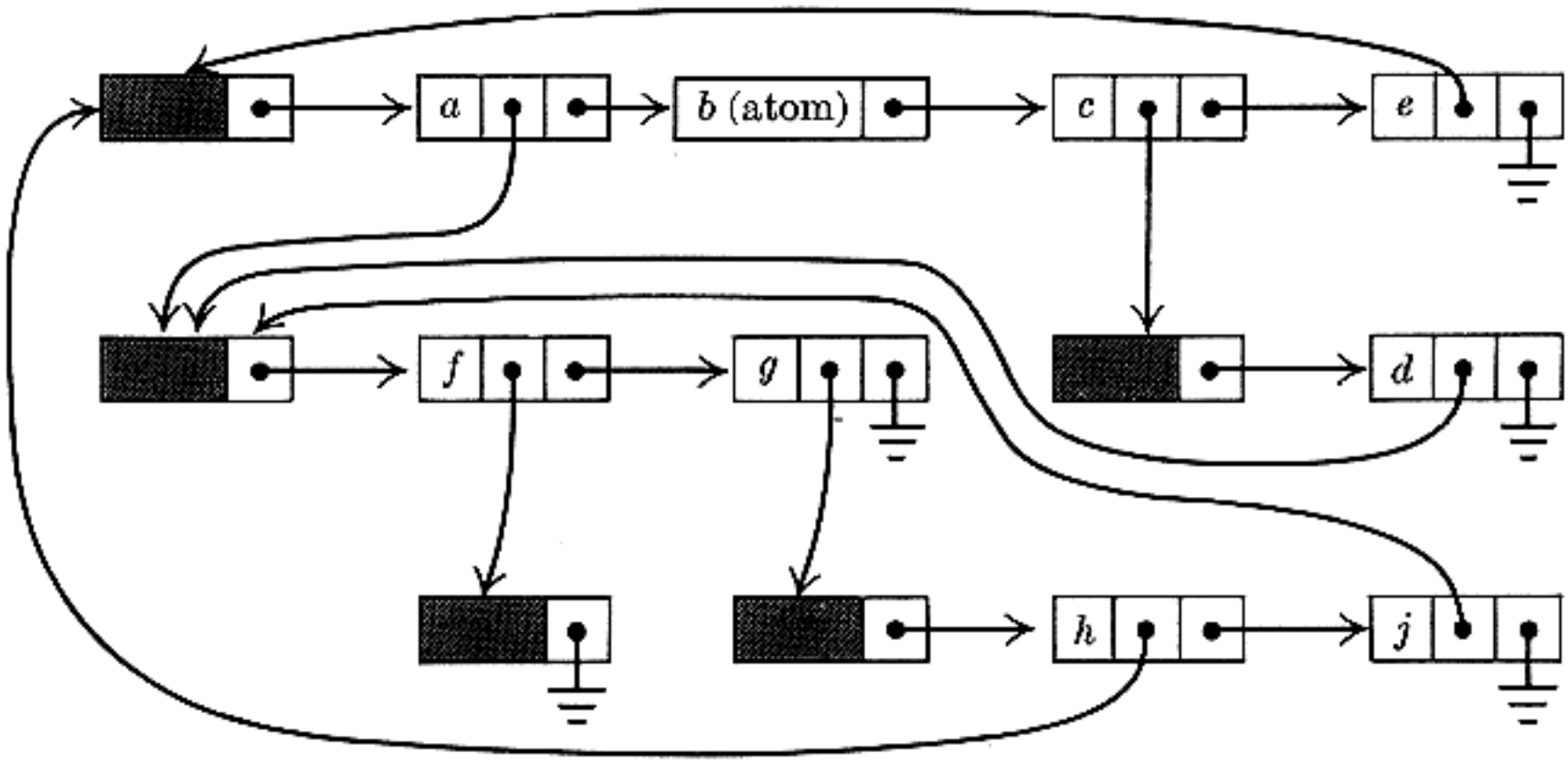


(6)

Unfortunately, this simple idea is *not* quite adequate for the most common List processing applications. For example, suppose that we have the List $L = (A, a, (A, A))$, which contains three references to another List $A = (b, c, d)$. One of the typical List processing operations is to remove the leftmost element of A, so that A becomes (c, d); but this requires *three* changes to the representation of L, if we are to use the technique shown in (6), since each pointer to A points to the element b that is being deleted. A moment's reflection will convince the reader that it is extremely undesirable to change the pointers in every reference to A just because the first element of A is being deleted. (*Note:* In this example we could try to be tricky, assuming that there are no pointers to the element c, by moving the entire element c into the location formerly occupied by b and then deleting the element c. But this trick fails to work when A loses its last element and becomes empty.)

For this reason the representation scheme (6) is generally replaced by another scheme which is similar, but uses a *List head* to begin each List, as was introduced in Section 2.2.4. Each List contains an additional node called its List head, so that the configuration (6) would, for example, be represented as shown in diagram (7) at the top of the next page.

The introduction of these header nodes is not really a waste of memory space in practice, since many uses for the apparently unused fields (which are shaded areas in diagram (7)) generally present themselves. For example, there is room for a reference count, or a pointer to the right end of the List, or an alphabetic name, or a "scratch" field which aids traversal algorithms, etc.



(7)

Note that in our original diagram (6), the node containing b is an atom while the node containing f specifies an empty List. These two things are structurally identical, and so the reader would be quite justified in asking why we bother to talk about "atoms" at all; with no loss of generality we could have defined Lists as merely "a finite sequence of zero or more Lists," with our usual convention that each node of a List may contain data besides its structural information. This point of view is certainly defensible and it makes the concept of an "atom" seem very artificial. There is, however, a good reason for singling out atoms as we have done, when efficient use of computer memory is taken into consideration, since atoms are not subject to the same sort of general-purpose manipulation that is desired for Lists. The memory representation (6) shows there is probably more room for information in an atomic node, b, than in a List node, f; and when List head nodes are also present as in (7), there is a dramatic difference between the storage requirements for the nodes b and f. Thus the concept of atoms is introduced primarily to aid in the effective use of computer memory. [Typical Lists contain many more atoms than our example would indicate; the example (4)–(7) is intended to show the complexities that are possible, not the simplicities that are usual.]

A List is in essence nothing more than a linear list whose elements may contain pointers to other Lists. The common operations we wish to perform on Lists are the usual ones desired for linear lists (creation, destruction, insertion, deletion, splitting, concatenation), plus further operations which are primarily of interest for tree structures (copying, traversal, input and output of nested information). For these purposes any of the three basic techniques for representing linked linear lists in memory—namely straight, circular, or double linkage—can be used, with varying degrees of efficiency depending on the algorithms being employed. For these three types of representation, diagram (7) might appear in memory as listed in (8) at the top of the next page.

Memory location	*Straight linkage* INFO	DLINK	RLINK	*Circular linkage* INFO	DLINK	RLINK	*Double linkage* INFO	DLINK	LLINK	RLINK
010:	—	head	020	—	head	020	—	head	050	020
020:	*a*	060	030	*a*	060	030	*a*	060	010	030
030:	*b*	atom	040	*b*	atom	040	*b*	atom	020	040
040:	*c*	090	050	*c*	090	050	*c*	090	030	050
050:	*e*	010	Λ	*e*	010	010	*e*	010	040	010
060:	—	head	070	—	head	070	—	head	080	070
070:	*f*	110	080	*f*	110	080	*f*	110	060	080
080:	*g*	120	Λ	*g*	120	060	*g*	120	070	060
090:	—	head	100	—	head	100	—	head	100	100
100:	*d*	060	Λ	*d*	060	090	*d*	060	090	090
110:	—	head	Λ	—	head	110	—	head	110	110
120:	—	head	130	—	head	130	—	head	140	130
130:	*h*	010	140	*h*	010	140	*h*	010	120	140
140:	*j*	060	Λ	*j*	060	120	*j*	060	130	120

(8)

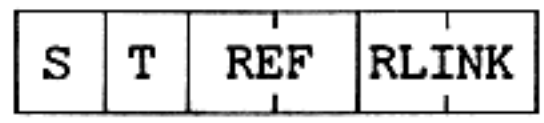
Here "LLINK" is used for a pointer to the left in a doubly linked representation. Note that the INFO and DLINK fields are identical in all three forms.

There is no need to repeat here the algorithms for List manipulation in any of these three forms, since the ideas are identical to those we have already seen many times in this chapter. The following important points about Lists, which distinguish them from the simpler special cases treated earlier, should be noted, however:

1) It is implicit in the above memory representation that atomic nodes are distinguishable from nonatomic nodes; furthermore, when circular or doubly linked lists are being used, it is desirable to distinguish header nodes from the other types, as an aid in traversing the Lists. Therefore each node generally contains a TYPE field which tells what kind of information the node represents. This TYPE field is often used also to distinguish between various types of atoms (e.g., between alphabetic, integer, or floating-point data, for use when printing or displaying answers).

2) The following are two examples of possible ways to design the format of nodes for general List manipulation with the MIX computer:

a) Possible one-word format, assuming all INFO appears in atoms:



(9)

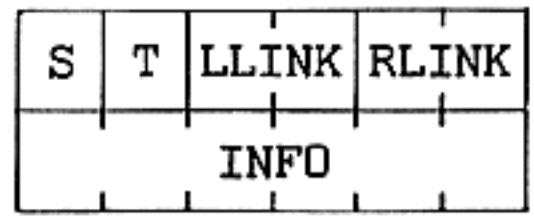
S (sign): "mark bit" used in "garbage collection" (see below).

T (type): T = 0 for List head; T = 1 for sub-List element; T > 1 for atoms.

REF: When T = 0, REF is a "reference count" (see below); when T = 1, REF points to the List head of the sub-List in question; when T > 1, REF points to a node containing a mark bit and five bytes of atomic information.

RLINK: Pointer for straight or circular linkage as in (8).

b) Possible two-word format:



(10)

S,T: As in (9).

LLINK,RLINK: Usual pointers for double linkage as in (8).

INFO: Full word of information associated with this node; for a header node this may include a reference count, a running pointer to the interior of the List to facilitate linear traversal, an alphabetic name, etc.

3) It is clear that Lists are very general structures; indeed, it seems fair to state that any structure whatsoever can be represented as a List when appropriate conventions are made. Because of this universality of Lists, a large number of programming systems have been designed to facilitate List manipulation, and there are usually several such systems available at any computer installation. These systems are based on some general-purpose format for nodes such as (9) or (10) above, designed for flexibility in List operations. Actually, it is clear that this general-purpose format is usually not the best format suited to a *particular* application, and the processing time using the general-purpose routines is noticeably slower than a person would achieve by hand-tailoring the system to his particular problem. For example, it is easy to see that nearly all of the applications we have worked out so far in this chapter would be encumbered by a general-List representation as in (9) or (10) instead of the node format that was given in each case. A List manipulation routine must often examine the T-field when it processes nodes, and this was not needed in any of our programs so far. This loss of efficiency is repaid in many instances by the comparative ease of programming and the reduction of debugging time that are obtained with the general-purpose system.

4) There is one extremely significant difference between algorithms for List processing and the algorithms given previously in this chapter. Since a single List may be contained in many other Lists, it is by no means clear exactly when a List should be returned to the pool of available storage. Our algorithms so far have always said "AVAIL ⇐ X", whenever NODE(X) was no longer needed. But since general Lists can grow and die in such an unpredictable manner while a program runs, it is often quite difficult to tell just when a particular node is superfluous. Therefore the problem of maintaining the list of available space is considerably more difficult with Lists than in the simple cases considered previously. We will devote the rest of this section to a discussion of this problem.

Let us imagine we are designing a general-purpose List processing system that will be used by hundreds of other programmers. Two principal methods

have been suggested for maintaining the available space list: the use of *reference counters*, and *garbage collection*. The reference-counter technique makes use of a new field in each node, which contains a count of how many arrows point to this node. Such a count is rather easy to maintain as a program runs, and whenever it drops to zero, the node in question becomes available. The garbage-collection technique requires a new one-bit field in each node called the "mark bit." The idea in this case is to write nearly all the algorithms so that they do not return any nodes to free storage, and to let the program run merrily along until all of the available storage is gone; then a "garbage-collector" algorithm makes use of the mark bits to return to available storage all nodes that are not currently accessible, and the program continues.

Neither of these two methods is completely satisfactory. The principal drawback of the reference-counter method is the fact that it does not always free all the nodes that are available. It works fine for overlapped Lists, but recursive Lists, like our examples L and N in (4), will *never* be returned to storage by the reference counter technique; their counts will be nonzero (since they refer to themselves) even when no other List accessible to the running program points to them. Furthermore, the reference-counter method uses a good chunk of space in each node (although sometimes this space is available anyway due to the computer word size).

The difficulty with the garbage-collection technique, besides the annoying loss of a bit in each node, is that it runs very slowly when nearly all the memory space is in use; and in such cases the number of free storage cells found by the reclamation process is not worth the effort. Those programs which exceed the capacity of storage (and many undebugged programs do!) often waste a good deal of time calling the garbage collector several, almost fruitless, times just before storage is finally exhausted. A partial solution to this problem is to let the programmer specify a number k, such that he does not wish to continue processing after a garbage collection run has found k or fewer free nodes. A further problem is the occasional difficulty of determining exactly what Lists are not garbage at a given stage; if the programmer has been using any nonstandard techniques or keeping any pointer values in unusual places, chances are good that the garbage collector will go awry. Some of the greatest mysteries in the history of computer program debugging have been caused by the fact that garbage collection suddenly took place at an unexpected time during the running of programs that had worked many times before. Garbage collection also requires that programmers keep valid information in all pointer fields at all times, although it is often convenient to leave meaningless information in fields which are never referred to by the program (for example, the link in the rear node of a queue, see exercise 2.2.3–6). We might also note that garbage collection is unsuitable for "real-time" applications, because even if the garbage collector goes into action infrequently, it requires large amounts of computer time on these occasions. (However, see exercise 12.)

Although garbage collection requires one mark bit for each node, it is possible to keep a separate table of all the mark bits packed together in another memory area, with a suitable correspondence between the location of a node and its mark bit. On some computers this idea can lead to a method of handling garbage collection which is more attractive than giving up a bit in each node, but on many other computers it slows down the process of garbage collection quite a bit.

J. Weizenbaum has suggested an interesting modification of the reference-counter technique. Using doubly linked List structures, he puts a reference counter only in the header of each List. Thus, when pointer variables traverse a List, they are not included in the reference counts for the individual nodes; but since the programmer knows the rules by which reference counts are maintained for entire Lists, he knows (in theory) how to avoid referring to any List that has a reference count of zero. The programmer also has the ability to explicitly override reference counts and to return certain Lists to available storage. These ideas require the programmer to exercise caution; they prove to be somewhat dangerous in the hands of inexperienced programmers and have tended to make program debugging more difficult due to the consequences of referring to nodes that have been erased. The nicest part of Weizenbaum's approach is his treatment of Lists whose reference count has just gone to zero: such a List is appended at the *end* of the current available space list—this is easy to do with doubly linked lists—and it is considered for available space only after all previously available cells are used up; then as the individual nodes of this List do become available, the reference counters of Lists *they* refer to are decreased by one. This delayed action of erasing the Lists is quite efficient with respect to running time; but it tends to make incorrect programs run correctly for awhile! For further details see *CACM* **6** (1963), 524–544.

Algorithms for garbage collection are quite interesting for several reasons. In the first place, such an algorithm is useful in other situations when we want to "mark all nodes directly or indirectly referred to by a given node." (For example, we might want to find all subroutines called directly or indirectly by a certain subroutine; cf. exercise 2.2.3–26. See also the ancestor algorithm in Chapter 7.)

Garbage collection generally proceeds in two phases. We assume that the mark bits of all nodes are initially zero (or we set them all to zero). Now the first phase marks all the nongarbage nodes, starting from those which are immediately accessible to the main program. The second phase makes a sequential pass over the entire memory pool area, putting all unmarked nodes onto the list of free space. The marking phase is the most interesting, and so we will concentrate our attention on it. There are variations on the second phase which make it nontrivial; see exercise 9.

The most interesting feature of garbage collection is the fact that while this algorithm is running, *there is only a very limited amount of storage available which*

we can use to control our marking algorithm. This intriguing problem will become clear in the following discussion; it is a difficulty which is not appreciated by most people when they first hear about the idea of garbage collection, and for many years there was no good solution to it.

The following marking algorithm is perhaps the most obvious:

Algorithm A (*Marking*). Let the entire memory used for List storage be `NODE(1)`, `NODE(2)`, ..., `NODE(M)`, and suppose that these words either are "atoms" or contain two link fields `REF` and `RLINK`. Assume that all nodes are initially *unmarked*. The purpose of this algorithm is to *mark* all of the nodes which can be reached by a chain of `REF` and/or `RLINK` pointers in nonatomic nodes, starting from a set of "immediately accessible" nodes.

A1. [Initialize.] Mark all nodes that are "immediately accessible," i.e., the nodes pointed to by certain fixed locations in the main program which are used as a source for all memory accesses. Set `K` $\leftarrow$ 1.

A2. [Does `NODE(K)` imply another?] Set `K1` $\leftarrow$ `K` + 1. If `NODE(K)` is an atom or unmarked, go to step A3. Otherwise, if `NODE(REF(K))` is unmarked, mark it, and if it is not an atom, set `K1` $\leftarrow$ min (`K1`, `REF(K)`). Similarly, if `NODE(RLINK(K))` is unmarked, mark it, and if it is not an atom, set `K1` $\leftarrow$ min (`K1`, `RLINK(K)`).

A3. [Done?] Set `K` $\leftarrow$ `K1`. If `K` $\leq$ `M`, return to step A2; otherwise the algorithm terminates. ▮

Throughout this algorithm and the ones which follow in this section, we will assume for convenience that the nonexistent node "`NODE(`Λ`)`" *is "marked."* (For example, `REF(K)` or `RLINK(K)` may equal Λ in step A2.)

A variant of Algorithm A sets `K1` $\leftarrow$ `M` + 1 in step A1, removes the operation "`K1` $\leftarrow$ `K` + 1" from step A2, and instead changes step A3 to

"**A3′.** [Done?] Set `K` $\leftarrow$ `K` + 1. If `K` $\leq$ `M`, return to step A2. Otherwise if `K1` $\leq$ `M`, set `K` $\leftarrow$ `K1` and `K1` $\leftarrow$ `M` + 1 and return to step A2. Otherwise the algorithm terminates."

It is very difficult to give a precise analysis of Algorithm A, or to determine whether it is better or worse than the variant just described, since no meaningful way to describe the probability distribution of the input presents itself. We can say it takes up time proportional to n`M` in the worst case, where n is the number of cells it marks, and, in general, we can be sure it is very slow when n is large. *Algorithm A would be too slow to make garbage collection a usable technique.*

Another fairly evident marking algorithm is to follow all paths and to record branch points on a stack as we go:

Algorithm B (*Marking*). This algorithm achieves the same effect as Algorithm A, using `STACK[1]`, `STACK[2]`, ... as auxiliary storage to keep track of all paths that have not yet been pursued to completion.

B1. [Initialize.] Let T be the number of immediately accessible nodes; mark them and place pointers to them in STACK[1], . . . , STACK[T].

B2. [Stack empty?] If T = 0, the algorithm terminates.

B3. [Remove top entry.] Set K ← STACK[T], T ← T − 1.

B4. [Examine links.] If NODE(K) is an atom, return to B2. Otherwise, if NODE(RLINK(K)) is unmarked, mark it and set T ← T + 1, STACK[T] ← RLINK(K); if NODE(REF(K)) is unmarked, mark it and set T ← T + 1, STACK[T] ← REF(K). Return to B2. ∎

Algorithm B clearly has an execution time essentially proportional to the number of cells it marks, and this is as good as we could possibly expect; but it is not really usable for garbage collection because there is no place to keep the stack! It does not seem unreasonable to assume that the stack in Algorithm B might grow up to, say, five percent of the size of memory; but when garbage collection is called, and all available space has been used up, there is only a fixed (rather small) number of cells to use for such a stack. Most of the early garbage collection procedures were essentially based on this algorithm, and if the special stack space was used up, then the entire program was terminated.

A somewhat better alternative is possible, using a fixed stack size, by combining Algorithms A and B:

Algorithm C (*Marking*). This algorithm achieves the same effect as Algorithms A and B, using an auxiliary table of H cells, STACK[0], STACK[1], . . . , STACK [H − 1].

In this algorithm, the action "insert X on the stack" means the following: "Set T ← (T + 1) mod H, and STACK[T] ← X. If T = B, set B ← (B + 1) mod H and K1 ← min (K1, STACK[B])." (Note that T points to the current top of the stack, and B points one place below the current bottom; STACK essentially operates as an input-restricted deque.)

C1. [Initialize.] Set T ← −1, B ← −1, K1 ← M + 1. Mark all the immediately accessible nodes, and successively insert their locations onto the stack (as just described above).

C2. [Stack empty?] If T = B, go to C5.

C3. [Remove top entry.] Set K ← STACK[T], T ← (T − 1) mod H.

C4. [Examine links.] If NODE(K) is an atom, return to C2. Otherwise, if NODE(RLINK(K)) is unmarked, mark it and insert RLINK(K) on the stack. Similarly, if NODE(REF(K)) is unmarked, mark it and insert REF(K) on the stack. Return to C2.

C5. [Sweep.] If K1 > M, the algorithm terminates. (The variable K1 represents the smallest location where there is a possibility of a new lead to a node that should be marked.) Otherwise, if NODE(K1) is unmarked, increase K1 by 1 and repeat this step. If NODE(K1) is marked, set K ← K1, increase K1 by 1, and go to C4. ∎

This algorithm and Algorithm B can be improved if X is never put on the stack when NODE(X) is an atom; such modifications are straightforward and they have been left out to avoid making the algorithms unnecessarily complicated.

Algorithm C is essentially Algorithm A when H = 1 and Algorithm B when H = M; clearly, it is gradually more efficient as H becomes larger. Unfortunately, Algorithm C defies a precise analysis for the same reason as Algorithm A, and we have no good idea how large H should be to make this method fast enough. It is plausible but uncomfortable to say a value of H = 50 is sufficient to make Algorithm C usable for garbage collection in most applications.

Algorithms B and C use a stack kept in sequential memory locations; we have seen earlier in this chapter that linked memory techniques are well suited to maintaining stacks which are not consecutive in memory. This suggests the idea that we might keep the stack of Algorithm B somehow scattered *through the same memory area in which we are collecting garbage.* This could be done easily if we were to give the garbage collection routine a little more room in which to breathe. Suppose, for example, we assume that all Lists begin with a List head node; assume further that although the RLINK field of each header node is used to point to the first node of the corresponding list, the REF field of each header word is completely meaningless and inaccessible to the main program. We can then redesign Algorithm B so that the stack is maintained in the REF fields of the header nodes:

Algorithm D (*Marking*). This algorithm achieves the same effect as Algorithms A, B, and C, but it assumes that the REF field in each List head node may be changed by the marking routine, and that the REF field in other nonatomic nodes points to a List head.

D1. [Initialize.] Set T ← Λ. Then for each pointer P to the head of an immediately accessible List (cf. step A1 of Algorithm A), if NODE(P) is unmarked, mark it and set REF(P) ← T, T ← P.

D2. [Stack empty?] If T = Λ, the algorithm terminates.

D3. [Remove top entry.] Set P ← T, T ← REF(T).

D4. [Move right.] Set P ← RLINK(P); if NODE(P) is marked, go to D2. Otherwise mark NODE(P). If NODE(P) is not an atom, set Q ← REF(P); and if furthermore NODE(Q) is unmarked, mark it and set REF(Q) ← T, T ← Q. Repeat step D4. ∎

(This algorithm works with circularly linked Lists as well as with straight linkage, since the condition "NODE(P) is marked" in step D4, which terminates motion to the right, includes as special cases "P = Λ" for straight linkage, or "P is the head of the current List" for circular linkage.)

Algorithm D may be compared to Algorithm B, which is quite similar, and its running time is essentially proportional to the number of nodes marked. However, Algorithm D is *not* recommended, because its seemingly rather mild restrictions are often too stringent for a general List-processing system. This

algorithm essentially requires that all List structures be well-formed [as in (7)] whenever garbage collection is called into action. But algorithms which manipulate List structures *momentarily* leave the List structures malformed, and it is important that a garbage collector such as Algorithm D will not be used during these momentary periods. Moreover, there are several List-manipulation algorithms which intentionally play havoc with the link fields in Lists during their operation, although they are designed so that well-formed Lists are restored again after the algorithm has been completed; for example, see the differentiation algorithm, 2.3.2D, or the algorithm of exercise 2.3.1–21. Care must also be taken in step D1 when the program contains pointers to the middle of a List.

These considerations bring us to Algorithm E, which is an elegant marking method discovered independently by Peter Deutsch and by Herbert Schorr and W. M. Waite in 1965. The assumptions used in this algorithm are just a little different from those of Algorithms A through D.

Algorithm E (*Marking*). Assume that a collection of nodes is given having the following fields:

MARK (a one-bit field, initially zero in each node),
ATOM (another one-bit field),
ALINK (a pointer field),
BLINK (a pointer field).

When ATOM = 0, the ALINK and BLINK fields may contain Λ or a pointer to another node of the same format; when ATOM = 1, the contents of the ALINK and BLINK fields are irrelevant to this algorithm.

Given a pointer PO, this algorithm sets the MARK field to 1 in NODE(PO) and in every other node which can be reached from NODE(PO) by a chain of ALINK and BLINK pointers in nodes with ATOM = 0. The algorithm uses three pointer variables, T, Q, and P, and modifies the links during its execution in such a way that all ATOM, ALINK, and BLINK fields are restored to their original settings after completion, although they may be changed temporarily.

E1. [Initialize.] Set T $\leftarrow \Lambda$, P $\leftarrow$ PO. (Throughout the remainder of this algorithm, the variable T has a dual significance: When T $\neq \Lambda$, it points to the top of what is essentially a stack as in Algorithm D; and the node that T points to once contained a link equal to P in place of the "artificial" stack link which currently occupies NODE(T).)

E2. [Mark.] Set MARK(P) $\leftarrow$ 1.

E3. [Atom?] If ATOM(P) = 1, go to E6.

E4. [Down ALINK.] Set Q $\leftarrow$ ALINK(P). If Q $\neq \Lambda$ and MARK(Q) = 0, set ATOM(P) $\leftarrow$ 1, ALINK(P) $\leftarrow$ T, T $\leftarrow$ P, P $\leftarrow$ Q, and go to E2. (Here the ATOM field and ALINK fields are temporarily being altered, so that the list structure in certain marked nodes has been rather drastically changed. But these changes will be restored in step E6.)

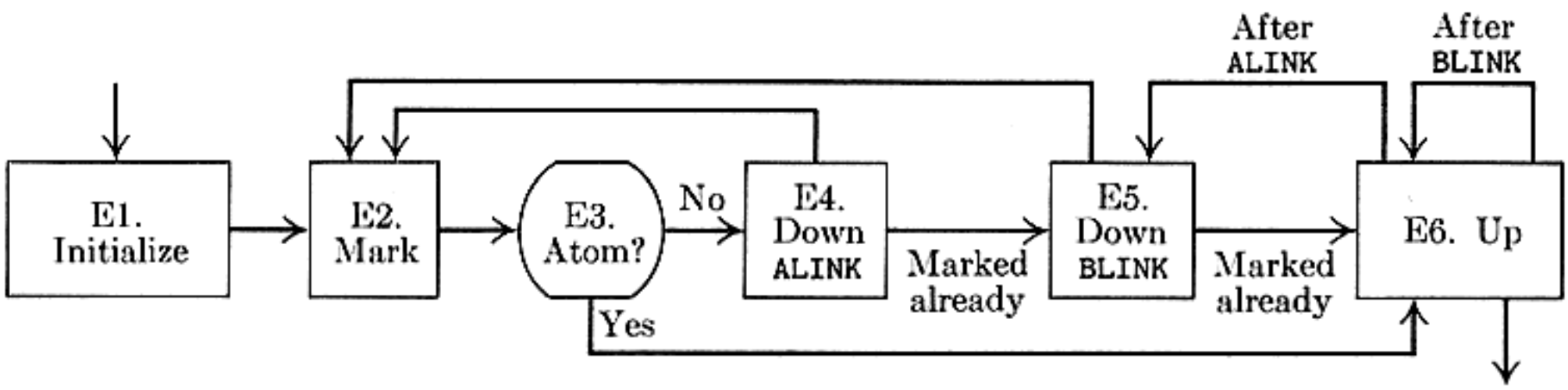


Fig. 38. Flowchart for Algorithm E.

E5. [Down BLINK.] Set Q ← BLINK(P). If Q ≠ Λ and MARK(Q) = 0, set BLINK(P) ← T, T ← P, P ← Q, and go to E2.

E6. [Up.] (This step undoes the link switching made in step E4 or E5; the setting of ATOM(T) tells whether ALINK(T) or BLINK(T) is to be restored.) If T = Λ, the algorithm terminates. Otherwise set Q ← T. If ATOM(Q) = 1, set ATOM(Q) ← 0, T ← ALINK(Q), ALINK(Q) ← P, P ← Q, and return to E5. If ATOM(Q) = 0, set T ← BLINK(Q), BLINK(Q) ← P, P ← Q, and return to E6. ∎

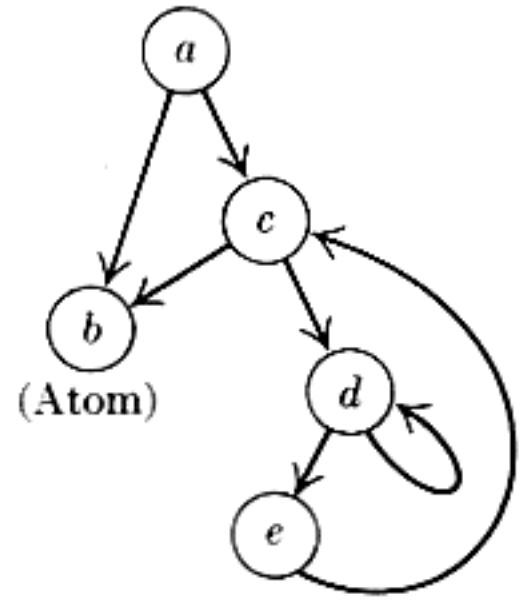
A flow chart of Algorithm E is shown in Fig. 38, and an example of the algorithm in action is given in Fig. 39, which shows the successive steps encountered for a simple List structure. The reader will find it worth while to study this algorithm very carefully; note how the list structure is artificially changed in steps E4 and E5, in order to keep track of the stack analogous to the stack in Algorithm D. When we return to a previous state, the ATOM field is used to tell which of ALINK, BLINK contains the artificial address. The "nesting" shown at the bottom of Fig. 39 illustrates how each nonatomic node is visited three times during Algorithm E (the same configuration (T, P) occurs at the beginning of steps E2, E5, and E6).

A proof that Algorithm E is valid can be formulated by induction on the number of nodes that are to be marked. One proves at the same time that P = P0 at the conclusion of the algorithm; for details, see exercise 3. Algorithm E will run faster if step E3 is deleted and instead special tests for "ATOM(Q) = 1" and appropriate actions are made in steps E4 and E5, as well as a test "ATOM(P0) = 1" in step E1. We have stated the algorithm in its present form for simplicity; the modifications just stated appear in the answer to exercise 4.

The idea used in Algorithm E can be applied to problems other than garbage collection; in fact, its use for tree traversal has already been mentioned in exercise 2.3.1–21. The reader may also find it useful to compare Algorithm E with the simpler problem solved in exercise 2.2.3–7.

Algorithms B and D are readily adapted to a situation where atomic nodes are indicated by a bit in the node that points to the atom, not in the atom itself. If Algorithm E is to be adapted to such a situation, it will be necessary to introduce another 1-bit field in nonatomic nodes. This is generally not hard to do. (For example, when there are two words per node, the least significant bit of each link field may be used to store temporary information.)

Although Algorithm E requires a time proportional to the number of nodes it marks, this constant of proportionality is not as small as in Algorithm B; the fastest garbage collection method known combines Algorithms B and E, as discussed in exercise 5.



a	ALINK[MARK]	b[0]	.	Λ[1]	.	b	.	.	.	.	.	.	.	.	.	.
	BLINK[ATOM]	c[0]	.	[1]	.	[0]	Λ	.	.	.	.	.	.	.	.	c
b	ALINK[MARK]	−[0]	.	.	[1]	.	.	.	.	.	.	.	.	.	.	.
	BLINK[ATOM]	−[1]	.	.	.	.	.	.	.	.	.	.	.	.	.	.
c	ALINK[MARK]	b[0]	.	.	.	.	.	[1]	.	.	.	.	.	.	.	.
	BLINK[ATOM]	d[0]	.	.	.	.	.	.	a	.	.	.	.	.	d	.
d	ALINK[MARK]	e[0]	.	.	.	.	.	.	.	c[1]	.	.	e	.	.	.
	BLINK[ATOM]	d[0]	.	.	.	.	.	.	.	[1]	.	.	[0]	.	.	.
e	ALINK[MARK]	Λ[0]	.	.	.	.	.	.	.	.	[1]	.	.	.	.	.
	BLINK[ATOM]	c[0]	.	.	.	.	.	.	.	.	.	.	.	.	.	.
	T	—	Λ	a	a	Λ	a	a	c	d	d	d	c	c	a	Λ
	P	—	a	b	b	a	c	c	d	e	e	e	d	d	c	a
	Next step	E1	E2	E2	E6	E5	E2	E5	E2	E2	E5	E6	E5	E6	E6	E6
	Nesting															

Fig. 39. A structure marked by Algorithm E. (The table shows only changes which have occurred since the previous step.)

Let us conclude this section with some quantitative remarks about the efficiency of garbage collection as opposed to the philosophy of "AVAIL ⇐ X", as was used in most of the previous examples in this chapter. In each of the previous cases we could have omitted all specific mention of returning nodes to free space and we could have substituted a garbage collector instead. (In a special-purpose application, as opposed to a set of general-purpose List manipulation subroutines, the programming and debugging of a garbage collector is more difficult than the methods we have used, and, of course, garbage collection requires an extra bit reserved in each node; but we are interested here in the relative speed of the programs once they have been written and debugged.)

The best garbage collection routines known have an execution time essentially of the form $c_1\mathtt{N} + c_2\mathtt{M}$, where c_1 and c_2 are constants, N is the number of nodes marked, and M is the total number of nodes in the memory. Thus M − N is the number of free nodes found, and the amount of time required to return

these nodes to free storage is $(c_1 N + c_2 M)/(M - N)$ per node. Let $N = \rho M$; this figure becomes $(c_1\rho + c_2)/(1 - \rho)$. So if $\rho = \frac{3}{4}$, i.e., if the memory is three-fourths full, it takes $3c_1 + 4c_2$ units of time per free node returned to storage; when $\rho = \frac{1}{4}$, the corresponding figure is only $\frac{1}{3}c_1 + \frac{4}{3}c_2$. If we do not use the garbage collection technique, the amount of time per node returned is essentially a constant, c_3, and it is doubtful that c_3/c_1 will be very large. Hence we can see to what extent garbage collection is inefficient when the memory becomes full, and how it is correspondingly efficient when the demand on memory is light.

It is possible to combine garbage collection with some of the other methods of returning cells to free storage; these ideas are not mutually exclusive, and some systems employ both the reference counter and the garbage collection schemes, besides allowing the programmer to explicitly erase nodes. The idea is to employ garbage collection only as a "last resort" whenever all other methods of returning cells have failed. This idea is of debatable merit, in view of the inefficiency of garbage collection when memory is nearly packed, and since so few programs come close to filling memory without exceeding the limits shortly thereafter.

EXERCISES

▶ **1.** [*M21*] In Section 2.3.4 we saw that trees are special cases of the "classical" mathematical concept of a directed graph. Can Lists be described in graph-theoretic terminology?

2. [*20*] In Section 2.3.1 we saw that tree traversal can be facilitated using a "threaded" representation inside the computer. Can List structures be threaded in an analogous way?

3. [*M26*] Prove the validity of Algorithm E. [*Hint:* See the proof of Algorithm 2.3.1T.]

4. [*28*] Write a MIX program for Algorithm E, assuming that nodes are represented as one MIX word, with MARK the (0:0) field [+ = 0, − = 1], ATOM the (1:1) field, ALINK the (2:3) field, BLINK the (4:5) field, and $\Lambda = 0$. Also, determine the execution time of your program in terms of relevant parameters. (Note that in the MIX computer the problem of determining whether a memory location contains −0 or +0 is not quite trivial, and this can be a factor in your program.)

5. [*25*] (Schorr and Waite.) Give a marking algorithm which combines Algorithms B and E as follows: The assumptions of Algorithm E with regard to the fields within nodes, etc., are assumed; however, an auxiliary stack STACK[1], STACK[2], . . . , STACK[N] is used as in Algorithm B, and the mechanism of Algorithm E is employed only when the stack is full.

6. [*00*] The quantitative discussion at the end of this section says garbage collection takes up approximately $c_1N + c_2M$ units of time; where does this "c_2M" term come from?

7. [*24*] (R. W. Floyd.) Design a marking algorithm that is similar to Algorithm E in that it uses no auxiliary stack, except (a) it has a more difficult task to do, in that each node contains only MARK, ALINK, and BLINK fields (so there is no ATOM field to

provide additional control), but (b) it has a simpler task to do, in that it marks only a binary tree instead of a general List. Here ALINK and BLINK are the usual LLINK and RLINK in a binary tree.

▸ **8.** [*27*] (L. P. Deutsch.) Design a marking algorithm similar to algorithms D and E in that it uses no auxiliary memory for a stack, but modify the method so that it works with nodes of variable size and with a variable number of pointers having the following format: The first word of a node has two fields MARK and SIZE; the MARK field is to be treated as in Algorithm E, and the SIZE field contains a number $n \geq 0$. This means there are n consecutive words after the first word, each containing two fields MARK (which is zero and should remain so) and LINK (which is Λ or points to the first word of another node). For example, a node with three pointers would comprise four consecutive words:

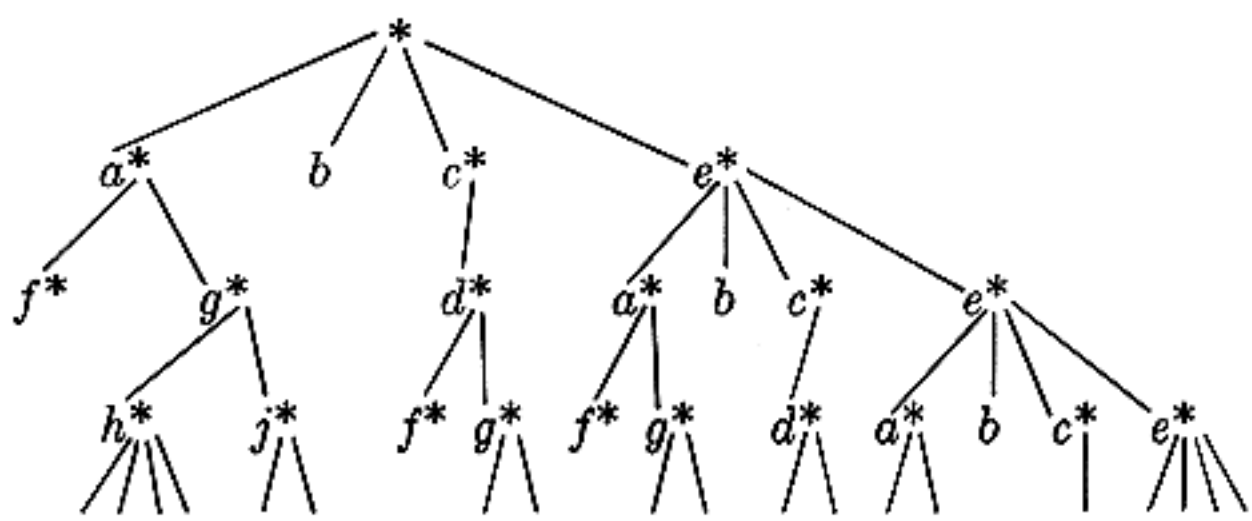
First word	MARK = 0 (Will be set to 1)	SIZE = 3
Second word	MARK = 0	LINK = first pointer
Third word	MARK = 0	LINK = second pointer
Fourth word	MARK = 0	LINK = third pointer.

9. [*28*] (D. Edwards.) Design an algorithm for the second phase of garbage collection which "compacts storage" in the following sense: Let NODE(1), . . . , NODE(M) be ▸ one-word nodes with fields MARK, ATOM, ALINK, and BLINK, as described in Algorithm E. Assume MARK = 1 in all nodes that are not garbage. The desired algorithm should relocate the marked nodes, if necessary, so that they all appear in consecutive locations NODE(1), . . . , NODE(k), and at the same time the ALINK and BLINK fields of nonatomic nodes should be altered if necessary so that the List structure is preserved.

10. [*28*] Design an algorithm which copies a List structure, assuming that an internal representation like that in (7) is being used. (Thus, if your procedure is asked to copy ▸ the List whose head is the node at the upper left corner of (7), a new set of Lists having 14 nodes, and with structure and information identical to that shown in (7), will be created.)

Assume that the List structure is stored in memory using S, T, REF, and RLINK fields as in (9), and that T(P0) = 0 (i.e., NODE(P0) is a List head). Assume further that the REF field in each List head node is Λ; if these REF fields are used by your copying procedure, they should be reset to Λ again afterwards.

11. [*M30*] Any List structure can be "fully expanded" into a tree structure by repeating all overlapping elements until none are left; when the List is recursive, this gives an infinite tree. For example, the List (5) would expand into an infinite tree whose first four levels are



Design an algorithm to test the equivalence of two List structures, in the sense that they have the same diagram when fully expanded. [For example, Lists A and B are equivalent in this sense, if

$$\begin{aligned} A &= (a{:}C, b, a{:}(b{:}D)) \\ B &= (a{:}(b{:}D), b, a{:}E) \\ C &= (b{:}(a{:}C)) \\ D &= (a{:}(b{:}D)) \\ E &= (b{:}(a{:}C)).] \end{aligned}$$

12. [*30*] (M. Minsky.) Show that it is possible to use a garbage collection method reliably in a "real time" application, e.g., when a computer is controlling some physical device, even when stringent upper bounds are placed on the maximum execution time required for each List operation performed. [*Hint:* Garbage collection can be arranged to work in parallel with the List operations, if appropriate care is taken.]

2.4. MULTILINKED STRUCTURES

Now that we have examined linear lists and tree structures in detail, the principles of representing structural information within a computer should be evident. In this section we will examine another application of these techniques, this time for the typical case in which the structural information is slightly more complicated: in higher-level applications, several types of structure are usually present simultaneously.

A "multilinked structure" involves nodes with several link fields in each node, not just one or two as in most of our previous examples. We have already seen some examples of multiple linkage, e.g., the simulated elevator system in Section 2.2.5 and the multivariate polynomials in Section 2.3.3.

We shall see that the presence of many different kinds of links per node does *not* necessarily make the accompanying algorithms any more difficult to write or to understand than the algorithms already studied. We will also discuss the important question, *"How much structural information ought to be explicitly recorded in memory?"*

The problem we will consider arises in connection with writing a compiler program for translating COBOL and related languages. A programmer who uses COBOL may give alphabetic names to the quantities in his program on several levels; for example, he may have two files of data for sales and purchases which have the following structure:

```
1 SALES                          1 PURCHASES
   2 DATE                           2 DATE
      3 MONTH                          3 DAY
      3 DAY                            3 MONTH
      3 YEAR                           3 YEAR
   2 TRANSACTION                    2 TRANSACTION
      3 ITEM                           3 ITEM
      3 QUANTITY                       3 QUANTITY
      3 PRICE                          3 PRICE
      3 TAX                            3 TAX
      3 BUYER                          3 SHIPPER
         4 NAME                           4 NAME
         4 ADDRESS                        4 ADDRESS
```

(1)

This configuration indicates that each item in SALES consists of two parts, the DATE and the TRANSACTION; the DATE is further divided into three parts, and likewise TRANSACTION has five subdivisions. Similar remarks apply to PURCHASES. The relative order of these names indicates the order in which the quantities appear in external representations of the file (e.g., punched cards or magnetic tape); note that in this example "DAY" and "MONTH" appear in opposite order in the two files. A COBOL programmer gives further information, not shown in this illustration, that tells how much space each item of information occupies and in what format it appears; these considerations are not relevant to us in this section, so they will not be mentioned further.

A COBOL programmer first describes the layout of his files and the other variables in his program, then he gives the algorithms that manipulate these quantities. To refer to an individual variable in the example above, it would not be sufficient merely to give the name DAY, since there is no way of telling if this variable called DAY is in the SALES file or in the PURCHASES file. Therefore a COBOL programmer is given the ability to write "DAY OF SALES" to refer to the DAY part of a SALES item. He could also write, more completely,

"DAY OF DATE OF SALES",

but in general there is no need to give more qualification than necessary to avoid ambiguity. Thus,

"NAME OF SHIPPER OF TRANSACTION OF PURCHASES"

may be abbreviated to

"NAME OF SHIPPER",

since only one part of the data has been called SHIPPER.

These rules of COBOL may be stated more precisely as follows:

a) Each name is immediately preceded by an associated positive integer called its "level number." A name either refers to an *elementary item* or else it is the name of a *group* of one or more items whose names follow. In the latter case, each item of the group must have the same level number, which must be greater than the level number of the group name. (For example, DATE and TRANSACTION above have level number 2, which is greater than the level number 1 of SALES.)

b) To refer to an elementary item or group of items named A_0, the general form is

$$A_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_n,$$

where $n \geq 0$ and where, for $0 \leq j < n$, A_j is the name of some item contained directly or indirectly within a group named A_{j+1}. There must be exactly one item A_0 satisfying this condition.

c) If the same name A_0 appears in several places, there must be a way to refer to each use of the name by using qualification.

As an example of rule (c), the data configuration

```
1 AA
  2 BB
    3 CC
    3 DD
  2 CC
```
(2)

would not be allowed, since there is no unambiguous way to refer to the second appearance of CC. (See exercise 4.)

There is another feature of COBOL which affects compiler writing and the application we are considering, namely an option in the language which makes it possible to refer to many items at once. A COBOL programmer may write

MOVE CORRESPONDING α TO β

which moves all items with corresponding names from data area α to data area β. For example, the COBOL statement

```
MOVE CORRESPONDING DATE OF SALES TO DATE OF PURCHASES
```

would mean that the values of MONTH, DAY, and YEAR from the SALES file are to be moved to the variables DAY, MONTH, YEAR in the PURCHASES file. (The relative order of DAY and MONTH is thereby interchanged.)

The problem we will investigate in this section is to design three algorithms suitable for use in a COBOL compiler, which are to do the following things:

Operation 1. To process a description of names and level numbers such as (1), putting the relevant information into tables within the compiler for use in operations 2 and 3.

Operation 2. To determine if a given qualified reference, as in rule (b), is valid, and when it is valid to locate the corresponding data item.

Operation 3. To find all corresponding pairs of items indicated by a "CORRESPONDING" statement.

We will assume that a "symbol table subroutine" exists within our compiler, which will convert an alphabetic name into a pointer to a memory location that contains a table entry for that name. (Methods for constructing symbol table algorithms are discussed in detail in Chapter 6.) In addition to the Symbol Table, there is a larger table which contains one entry for each item of data in the COBOL source program that is being compiled; we will call this the *Data Table.*

Clearly, we cannot design an algorithm for operation 1 until we know what kind of information is to be stored in the Data Table, and the form of the Data Table depends on what information we need to perform operations 2 and 3; thus we look first at operations 2 and 3.

In order to determine the meaning of the COBOL reference

$$A_0 \text{ OF } A_1 \text{ OF } \ldots \text{ OF } A_n, \qquad n \geq 0, \tag{3}$$

we should first look up the name A_0 in the Symbol Table. There ought to be a series of links from the Symbol Table entry to all Data Table entries for this name. Then for each Data Table entry we will want a link to the entry for the group item which contains it. Now if there is a further link field from the Data Table items back to the Symbol Table, it is not hard to see how a reference like (3) can be processed. Furthermore, we will want some sort of links from the Data Table entries for group items to the items in the group, in order to locate the pairs indicated by "MOVE CORRESPONDING".

We have thereby found a possible need for five link fields in each Data Table entry:

PREV (a link to the previous entry with the same name, if any);
FATHER (a link to the smallest group, if any, containing this item);
NAME (a link to the Symbol Table entry for this item);
SON (a link to the first subitem of a group);
BROTHER (a link to the next subitem in the group containing this item).

It is clear that COBOL data structures like those for SALES and PURCHASES above are essentially trees; and the FATHER, SON, and BROTHER links which appear here are familiar from our previous study. (The conventional binary tree representation of a tree consists of the SON and BROTHER links; adding the FATHER link gives what we have called a "triply linked tree." The five links above consist of these tree links together with PREV and NAME which superimpose further information on the tree structure.)

Perhaps not all five of these links will turn out to be necessary, or sufficient, but we will first try to design our algorithms under the tentative assumption that Data Table entries will involve these five link fields (plus further information irrelevant to our problems). As an example of the multiple linking used, consider the two COBOL data structures

```
1 A                 1 H
  3 B                 5 F
    7 C                 8 G
    7 D               5 B
  3 E                 5 C
  3 F                   9 E
    4 G                 9 D
                        9 G
```
(4)

They would be represented as shown in (5) (with links indicated symbolically). Note that the LINK field of the Symbol Table entries points to the most recently encountered Data Table entry for the symbolic name in question.

The first algorithm we require is one which builds the Data Table in such a form. Note the flexibility in choice of level numbers which is allowed by the COBOL rules; the left structure in (4) is completely equivalent to

```
1 A
  2 B
    3 C
    3 D
  2 E
  2 F
    3 G
```

because level numbers do not have to be sequential. [See rule (a), stated near

Symbol Table

	LINK
A:	A1
B:	B5
C:	C5
D:	D9
E:	E9
F:	F5
G:	G9
H:	H1

(Shading indicates additional information which is not relevant here)

Data Table

	PREV	FATHER	NAME	SON	BROTHER	
A1:	Λ	Λ	A	B3	H1	
B3:	Λ	A1	B	C7	E3	
C7:	Λ	B3	C	Λ	D7	
D7:	Λ	B3	D	Λ	Λ	
E3:	Λ	A1	E	Λ	F3	
F3:	Λ	A1	F	G4	Λ	
G4:	Λ	F3	G	Λ	Λ	
H1:	Λ	Λ	H	F5	Λ	
F5:	F3	H1	F	G8	B5	
G8:	G4	F5	G	Λ	Λ	
B5:	B3	H1	B	Λ	C5	
C5:	C7	H1	C	E9	Λ	
E9:	E3	C5	E	Λ	D9	
D9:	D7	C5	D	Λ	G9	
G9:	G8	C5	G	Λ	Λ	

(5)

the beginning of this section.] There are sequences of level numbers which may not be used, however; for example, if the level number of D in (4) were changed to "6" (in either place) we would have a meaningless data configuration which violates the rule that all items of a group should have the same number, which is higher than the group name. The following algorithm therefore makes sure that this restriction is met.

Algorithm A (*Build Data Table*). This algorithm is given a sequence of pairs (L, P), where L is a positive integer "level number" and P points to a Symbol Table entry, corresponding to COBOL data structures, such as (4) above. The algorithm builds a Data Table as in the example (5) above. When P points to a Symbol Table entry that has not appeared before, LINK(P) will equal Λ. This algorithm uses an auxiliary stack which is treated as usual (using either sequential memory locations, as in Section 2.2.2, or linked allocation, as in Section 2.2.3.)

A1. [Initialize.] Set the stack contents to the single entry (0, Λ). (The stack entries throughout this algorithm are pairs (L, P), where L is an integer and P a pointer; as this algorithm proceeds, the stack contains the level number

and pointers to the last data entries on all levels higher in the tree than the current level. For example, just before encountering the pair "3 F" in the above example, the stack would contain

$$(0, \Lambda) \qquad (1, \text{A1}) \qquad (3, \text{E3})$$

from bottom to top.)

A2. [Next item.] Let (L, P) be the next data item from the input. If the input is exhausted, however, the algorithm terminates. Set Q $\Leftarrow$ AVAIL (i.e., let Q be the location of a new node in which we can put the next Data Table entry).

A3. [Set name links.] Set

$$\texttt{PREV(Q)} \leftarrow \texttt{LINK(P)}, \qquad \texttt{LINK(P)} \leftarrow \texttt{Q}, \qquad \texttt{NAME(Q)} \leftarrow \texttt{P}.$$

(This properly sets two of the five links in NODE(Q). We now want to set FATHER, SON, and BROTHER appropriately.)

A4. [Compare levels.] Let the top entry of the stack be (L1, P1). If L1 $<$ L, set SON(P1) $\leftarrow$ Q (or, if P1 $= \Lambda$, set FIRST $\leftarrow$ Q, where FIRST is a variable which is to point to the first Data Table entry) and go to A6.

A5. [Remove top level.] If L1 $>$ L, remove the top stack entry, let (L1, P1) be the new entry which has just come to the top of the stack, and repeat step A5. If L1 $<$ L, signal an error (mixed numbers have occurred on the same level). Otherwise, i.e. when L1 $=$ L, set BROTHER(P1) $\leftarrow$ Q, remove the top stack entry, and let (L1, P1) be the pair which has just come to the top of the stack.

A6. [Set family links.] Set

$$\texttt{FATHER(Q)} \leftarrow \texttt{P1}, \ \texttt{SON(Q)} \leftarrow \Lambda, \ \texttt{BROTHER(Q)} \leftarrow \Lambda.$$

A7. [Add to stack.] Place (L, Q) on the top of the stack, and return to step A2. ∎

The introduction of an auxiliary stack, as explained in step A1, makes this algorithm so straight forward that it needs no further explanation.

The next problem is to locate the data table entry corresponding to a reference

$$A_0 \ \texttt{OF} \ A_1 \ \texttt{OF} \ \ldots \ \texttt{OF} \ A_n, \qquad n \geq 0. \tag{6}$$

A "good" compiler will also check to ensure that such a reference is unambiguous. In this case, a straightforward algorithm suggests itself immediately: all we need to do is to run through the list of Data Table entries for the name A_0 and make sure that exactly one of these entries matches the stated qualification $A_1, \ldots, A_n$.

Algorithm B (*Check a qualified reference*). Corresponding to reference (6), a Symbol Table subroutine will find pointers $P_0, P_1, \ldots, P_n$ to the Symbol Table entries for $A_0, A_1, \ldots, A_n$, respectively.

The purpose of this algorithm is to examine $P_0, P_1, \ldots, P_n$ and either to determine that reference (6) is in error, or to set variable Q to the address of the Data Table entry for the item refered to by (6).

B1. [Initialize.] Set Q $\leftarrow \Lambda$, P $\leftarrow$ LINK(P_0).

B2. [Done?] If P $= \Lambda$, the algorithm terminates; at this point Q will equal Λ if (6) does not correspond to any Data Table entry. Otherwise set S $\leftarrow$ P and $k \leftarrow 0$. (S is a pointer variable which will run from P up the tree through FATHER links; k is an integer variable which goes from 0 to n. In practice, the pointers $P_0, \ldots, P_n$ would often be kept in a linked list, and instead of k, we would substitute a pointer variable which traverses this list; see exercise 5.)

B3. [Match complete?] If $k < n$ go on to B4. Otherwise we have found a matching data table entry; if Q $\neq \Lambda$, this is the second entry found, so an error condition is signaled. Set Q $\leftarrow$ P, P $\leftarrow$ PREV(P), and go to B2.

B4. [Increase k.] Set $k \leftarrow k + 1$.

B5. [Move up tree.] Set S $\leftarrow$ FATHER(S). If S $= \Lambda$, we have failed to find a match; set P $\leftarrow$ PREV(P) and go to B2.

B6. [A_k match?] If NAME(S) $= P_k$, go to B3, otherwise go to B5. ∎

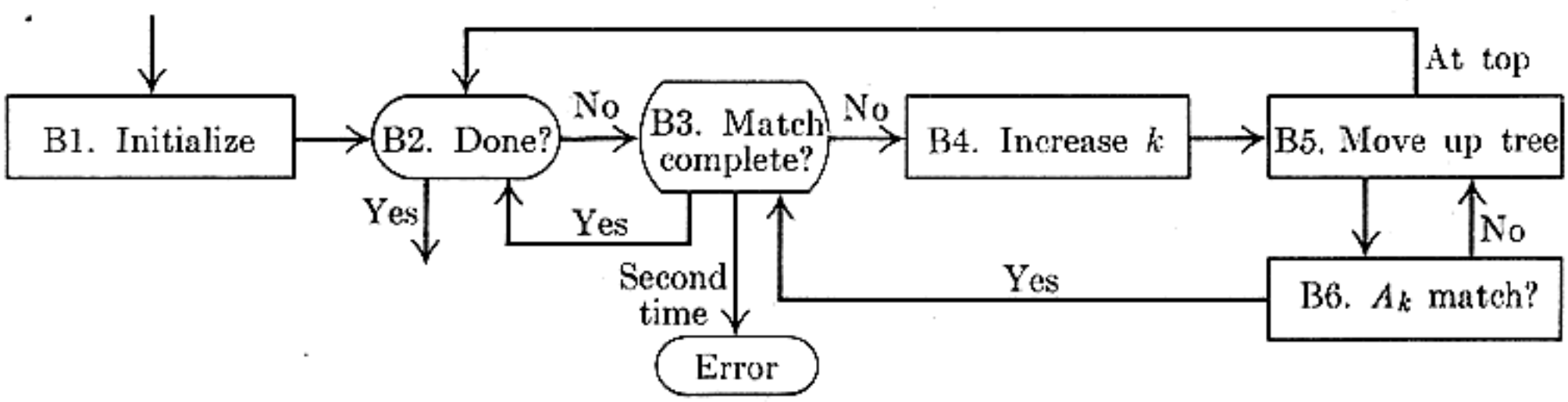


Fig. 40. Algorithm for checking a COBOL reference.

Note that the SON and BROTHER links are not needed by this algorithm.

The third and final algorithm that we need concerns "MOVE CORRESPONDING", and before we design such an algorithm, we must have a precise definition of what is required. The COBOL statement

$$\texttt{MOVE CORRESPONDING } \alpha \texttt{ TO } \beta \tag{7}$$

where α and β are references such as (6) to data items, is an abbreviation for the set of all statements

$$\texttt{MOVE } \alpha' \texttt{ TO } \beta'$$

where there exists an integer $n \geq 0$ and n names $A_0, A_1, \ldots, A_{n-1}$ such that

$$\begin{aligned} \alpha' &= A_0 \ \texttt{OF} \ A_1 \ \texttt{OF} \ \ldots \ \texttt{OF} \ A_{n-1} \ \texttt{OF} \ \alpha \\ \beta' &= A_0 \ \texttt{OF} \ A_1 \ \texttt{OF} \ \ldots \ \texttt{OF} \ A_{n-1} \ \texttt{OF} \ \beta \end{aligned} \tag{8}$$

and either α' or β' is an elementary item (not a group item). Furthermore we require that (8) show *complete* qualifications, i.e., that A_{j+1} is the father of A_j for $0 \leq j \leq n-1$; α' and β' must be exactly n levels farther down in the tree than α and β are.

In our example (4),

"MOVE CORRESPONDING A TO H"

is an abbreviation for the statements

```
MOVE B OF A TO B OF H
MOVE G OF F OF A TO G OF F OF H
```

The algorithm to recognize all corresponding pairs α', β' is quite interesting although not difficult; we move through the tree, whose root is α, in preorder, simultaneously looking in the β tree for matching names, and skipping over subtrees in which no corresponding elements can possibly occur. The names $A_0, \ldots, A_{n-1}$ of (8) are discovered in the opposite order $A_{n-1}, \ldots, A_0$.

Algorithm C (*Find* CORRESPONDING *pairs*). Given PO and QO, which point to Data Table entries for α and β, respectively, this algorithm successively finds all pairs (P, Q) of pointers to items (α', β') satisfying the constraints mentioned above.

C1. [Initialize.] Set P $\leftarrow$ PO, Q $\leftarrow$ QO. (In the remainder of this algorithm, the pointer variables P and Q will walk through trees having the respective roots α and β.)

C2. [Elementary?] If SON(P) $= \Lambda$ or SON(Q) $= \Lambda$, output (P,Q) as one of the desired pairs and go to C5. Otherwise set P $\leftarrow$ SON(P), Q $\leftarrow$ SON(Q). (In this step, P and Q point to items α' and β', satisfying (8), and we wish to MOVE α' TO β' if and only if either α' or β' (or both) is an elementary item.)

C3. [Match name.] (Now P and Q point to data items which have respective qualifications of the forms

$$A_0 \ \texttt{OF} \ A_1 \ \texttt{OF} \ \ldots \ \texttt{OF} \ A_{n-1} \ \texttt{OF} \ \alpha$$

and

$$B_0 \ \texttt{OF} \ A_1 \ \texttt{OF} \ \ldots \ \texttt{OF} \ A_{n-1} \ \texttt{OF} \ \beta.$$

The object is to see if we can make $B_0 = A_0$ by examining all the names of the group A_1 OF $\ldots$ OF A_{n-1} OF β.) If NAME(P) $=$ NAME(Q), go to C2 (a

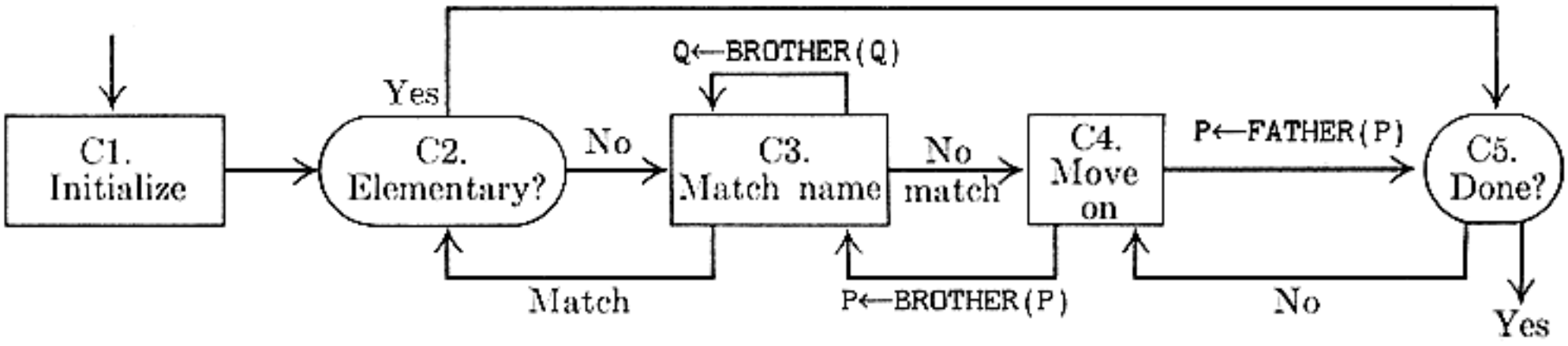


Fig. 41. Algorithm for "MOVE CORRESPONDING."

match has been found). Otherwise, if BROTHER(Q) $\neq \Lambda$, set Q $\leftarrow$ BROTHER(Q) and repeat step C3. (If BROTHER(Q) $= \Lambda$, no matching name is present in the group, and we continue on to step C4.)

C4. [Move on.] If BROTHER(P) $\neq \Lambda$, set

$$\texttt{P} \leftarrow \texttt{BROTHER(P)} \qquad \text{and} \qquad \texttt{Q} \leftarrow \texttt{SON(FATHER(Q))},$$

and go back to C3. If BROTHER(P) $= \Lambda$, set

$$\texttt{P} \leftarrow \texttt{FATHER(P)} \qquad \text{and} \qquad \texttt{Q} \leftarrow \texttt{FATHER(Q)}.$$

C5. [Done?] If P $=$ P0, the algorithm terminates; otherwise go to C4. ∎

A flow chart for this algorithm is shown in Fig. 41. A proof that this algorithm is valid can readily be constructed by induction on the size of the trees involved (see exercise 9).

At this point it is worth while to study the ways in which the five link fields PREV, FATHER, NAME, SON, and BROTHER are used by Algorithms B and C. The striking feature is that these five links constitute a "complete set" in the sense that Algorithms B and C do virtually the minimum amount of work as they move through the Data Table; whenever it is necessary to refer to another Data Table entry, its address is immediately available at our fingertips; we do not need to search fo. it. It would be difficult to imagine how Algorithms B and C could possibly be made any faster if any additional link information were present in the table. (See exercise 11, however.)

Each link field may be viewed as a *clue* to the program, planted there in order to make the algorithms run faster (although, of course, algorithms which build the tables, like Algorithm A, run correspondingly slower, since they have more links to fill in). On the other hand it is clear that the Data Table constructed above contains much redundant information. Let us consider what would happen if we were to *delete* certain of the link fields.

The PREV link, while not used in Algorithm C, is extremely important for Algorithm B, and it seems to be an essential part of any COBOL compiler unless lengthy searches are to be carried out. A field which links together all items of the same name therefore seems essential for efficiency. We could perhaps modify the strategy slightly and adopt circular linking instead of terminating each list

with Λ, but there is no reason to do this unless other link fields are changed or eliminated.

The FATHER link is used in both Algorithms B and C, although its use in Algorithm C could be avoided if we used an auxiliary stack in that algorithm, or if we augmented BROTHER so that "thread" links are included (cf. Section 2.3.2). So we see that the FATHER link has been used in an essential way only in Algorithm B. If the BROTHER link were threaded, so that the items which now have BROTHER = Λ would have BROTHER = FATHER instead, it would be possible to locate the father of any data item by following the BROTHER links; the added "thread" links could be distinguished either by having a new TAG field in each node that says whether the BROTHER link is a thread, or by the condition "BROTHER(P) < P" if the Data Table entries are kept consecutively in memory in order of appearance. This would mean a short search would be necessary in step B5, and the algorithm would be correspondingly slower.

The NAME link is used by the algorithms only in steps B6 and C3. In both cases we could make the tests "NAME(S) = P_k", "NAME(P) = NAME(Q)" in other ways if the NAME link were not present (cf. exercise 10), but this would significantly slow down the inner loops of both Algorithms B and C. Here again we see a trade-off between the space for a link and the speed of the algorithms. (The speed of Algorithm C is not especially significant in COBOL compilers, when typical uses of MOVE CORRESPONDING are considered; but Algorithm B should be fast.) Experience indicates that other important uses are found for the NAME link within a COBOL compiler, especially in printing diagnostic information.

Since Algorithm A builds the Data Table step by step, and never has occasion to return it to the pool of available storage, we usually find that Data Table entries take consecutive memory locations in the order of appearance of the data items in the COBOL source program. Thus in our example (5), locations A1, B3, . . . would follow each other. This sequential nature of the Data Table leads to certain simplifications; for example, the SON link of each node is either Λ or it points to the node immediately following, so SON can be reduced to a 1-bit field. Alternatively, SON could be removed in favor of a test if FATHER(P + c) = P, where c is the node size in the Data Table.

Thus the five link fields are not all essential, although they are helpful from the standpoint of speed in Algorithms B and C. This situation is fairly typical of most multilinked structures.

It is interesting to note that at least half a dozen people writing COBOL compilers have independently arrived at this same way to maintain a Data Table using five links (or four of the five, usually with the SON link missing). The first publication of such a technique was by H. W. Lawson, Jr. (*ACM National Conference Digest*, Syracuse, N.Y., 1962). But in 1965 an ingenious technique for achieving the effects of Algorithms B and C, using *only two link fields* and sequential storage of the Data Table, without a very great decrease in speed, was introduced by David Dahm; see exercises 12 through 14.

1. [*00*] Considering COBOL data configurations as tree structures, are the data items listed by a COBOL programmer in preorder, postorder, endorder, or none of these orders?

2. [*10*] Comment about the running time of Algorithm A.

3. [*22*] The PL/I language accepts data structures much like those in COBOL, except any sequence of level numbers is possible. For example, the sequence

```
1 A                          1 A
3 B                          2 B
5 C      is equivalent to    3 C
4 D                          3 D
2 E                          2 E
```

In general, rule (a) is modified to read, "The items of a group must have a sequence of nonincreasing level numbers, all of which are greater than the level number of the group name." What modifications to Algorithm A would change it from the COBOL convention to this PL/I convention?

▶ **4.** [*26*] Algorithm A does not detect the error if a COBOL programmer violates rule (c) stated in the text. How should Algorithm A be modified so that only data structures satisfying rule (c) will be accepted?

5. [*20*] In practice, Algorithm B may be given a linked list of Symbol Table references as input, instead of what we called "$P_0, P_1, \ldots, P_n$." Let T be a pointer variable such that

$$\texttt{INFO(T)} \equiv P_0,\ \texttt{INFO(RLINK(T))} \equiv P_1,\ \ldots,$$
$$\texttt{INFO(RLINK}^n\texttt{(T))} \equiv P_n,\ \texttt{RLINK}^{n+1}\texttt{(T)} = \Lambda.$$

Show how to modify Algorithm B so that it uses such a linked list as input.

6. [*23*] The PL/I language accepts data structures much like those in COBOL, but does not make the restriction of rule (c); instead, we have the rule that a qualified reference (3) is unambiguous if it shows "complete" qualification, i.e., if A_{j+1} is the father of A_j for $0 \le j < n$, and if A_n has no father. Rule (c) is now weakened to the simple condition that no two items of a group may have the same name. The second "CC" in (2) would be referred to as "CC OF AA" without ambiguity; the three data items

```
1 A
  2 A
    3 A
```

would be referred to as "A", "A OF A", "A OF A OF A" with respect to the PL/I convention just stated. (*Note:* Actually the word "OF" is replaced by a period in PL/I, and the order is reversed; "CC OF AA" would really be written "AA.CC" in PL/I, but this is not important for the purposes of the present exercise.) Show how to modify Algorithm B so that it follows the PL/I convention, i.e., it does not regard a complete qualification as ambiguous.

7. [*15*] What does the COBOL statement "MOVE CORRESPONDING SALES TO PURCHASES" mean, given the data structures in (1)?

8. [*10*] Under what circumstances is

"MOVE CORRESPONDING α TO β"

exactly the same as

"MOVE α TO β",

according to the definition in the text?

9. [*M23*] Prove that Algorithm C is correct.

10. [*23*] (a) How could the test "NAME(S) = P_k" in step B6 be performed if there were no NAME link in the Data Table nodes? (b) How could the test "NAME(P) = NAME(Q)" in step C3 be performed if there were no NAME link in the Data Table entries? (Assume that all other links are present as in the text.)

▶ **11.** [*23*] What additional links or changes in the strategy of the algorithms of the text could make Algorithm B or Algorithm C faster?

12. [*25*] (D. M. Dahm.) Consider representing the Data Table in sequential locations with just two links for each item:

PREV (as in the text);
SCOPE (links to the last elementary item in this group).

We have SCOPE(P) = P if and only if NODE(P) represents an elementary item. For example, the Data Table of (5) would be replaced by

	PREV	SCOPE		PREV	SCOPE
A1:	Λ	G4	H1:	Λ	G9
B3:	Λ	D7	F5:	F3	G8
C7:	Λ	C7	G8:	G4	G8
D7:	Λ	D7	B5:	B3	B5
E3:	Λ	E3	C5:	C7	G9
F3:	Λ	G4	E9:	E3	E9
G4:	Λ	G4	D9:	D7	D9
			G9:	G8	G9

(Compare with (4) of Section 2.3.3.) Note that NODE(P) is part of the tree below NODE(Q) if and only if Q < P ≤ SCOPE(Q). Design an algorithm which performs the function of Algorithm B when the Data Table has this format.

▶ **13.** [*24*] Give an algorithm to substitute for Algorithm A when the Data Table is to have the format shown in exercise 12.

▶ **14.** [*28*] Give an algorithm to substitute for Algorithm C when the Data Table has the format shown in exercise 12.

We have seen how the use of links implies that tables need not be sequentially located in memory; a number of tables may independently grow and shrink in a common "pooled" memory area. However, our discussions have always tacitly assumed that all nodes are the same size, i.e., that they take the same number of memory cells.

For a great many applications, a suitable compromise can be found so that a uniform node size is used for all tables (for example, see exercise 2). Instead of simply taking the maximum size that is needed and wasting space in smaller nodes, it is customary to pick a rather small node size and to employ what may be called the classical *linked-memory philosophy:* "If there isn't room for the information here, let's put it somewhere else and plant a link to it."

For a great many other applications, however, a single node size is not reasonable; we often wish to have nodes of varying sizes sharing a common memory area. Putting this another way, we want algorithms for reserving and freeing variable-size blocks of memory from a larger storage area, where these blocks are to consist of consecutive memory locations. Such techniques are generally called "dynamic storage allocation" algorithms.

Sometimes, often in simulation programs, we want dynamic storage allocation for nodes of rather small sizes (say one to ten words); and at other times, often in "executive" control programs, we are dealing primarily with rather large blocks of information. These two points of view lead to slightly different approaches to dynamic storage allocation, although the methods have much in common. For uniformity in terminology between these two approaches, we will generally use the terms *block* and *area* rather than "node" in this section, to denote a set of contiguous memory locations.

A. Reservation. Figure 42 shows a typical "memory map" or "checkerboard," a chart showing the current state of some memory pool. In this case the memory is shown partitioned into 53 blocks of storage that are "reserved," i.e. in use, mixed together with 21 "free" or "available" blocks that are not in use. After

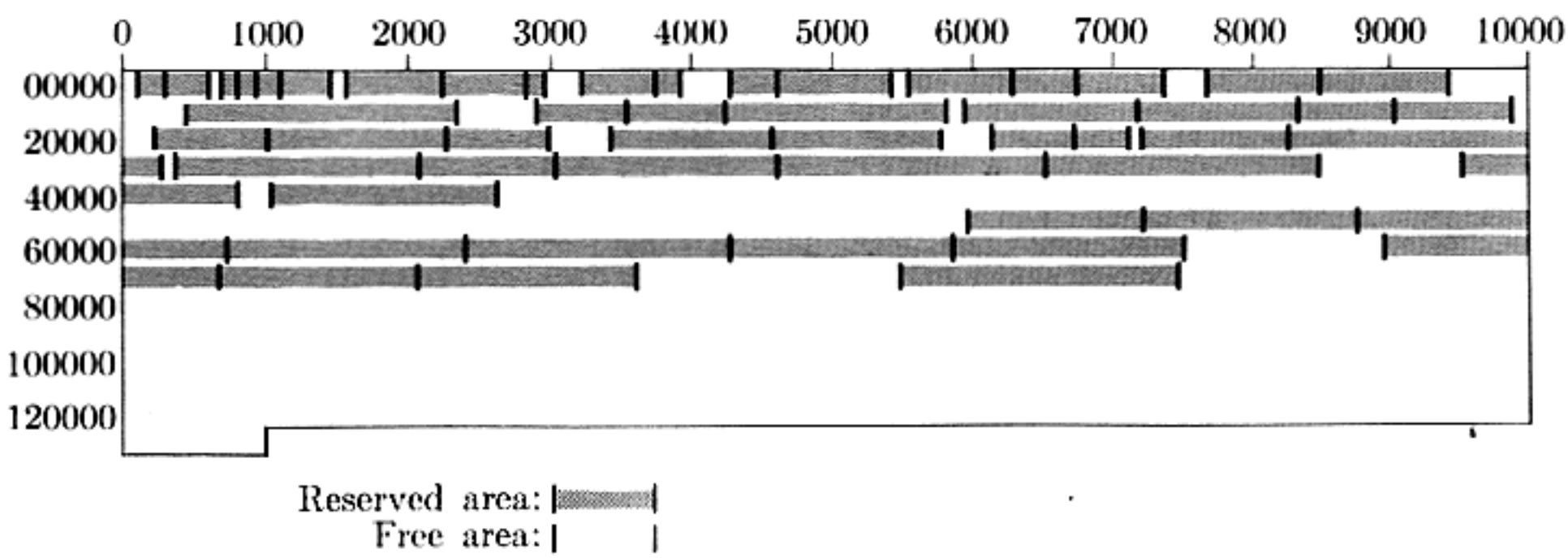


Fig. 42. A memory map.

dynamic storage allocation has been in operation for a while, the computer memory will perhaps look something like this; our first problem is

a) How is this partitioning of available space to be represented inside the computer?
b) Given such a representation of the available spaces, what is a good algorithm for finding a block of n consecutive free spaces and reserving them?

The answer to question (a) is, of course, to keep a *list* of the available space somewhere; this is almost always done best by using the available space *itself* to contain such a list. (An exception is the case when we are allocating storage for a disk file or other memory in which nonuniform access time makes it better to maintain a separate directory of available space.)

Thus, we can *link together* the available segments: the first word of each free storage area may contain the size of that block and the address of the next free area. The free blocks can be linked together in increasing or decreasing order of size, or in order of memory address, or in essentially random order.

For example, consider Fig. 42, which illustrates a large memory of 131,072 words, addressed from 0 to 131071. If we link together the available blocks in order of memory location, we would have one variable `AVAIL` that points to the first free block (in this case `AVAIL` would equal 0), and the other blocks would be represented as follows:

location	`SIZE`	`LINK`	
0	101	632	
632	42	1488	
⋮	⋮	⋮	[17 similar entries]
73654	1909	77519	
77519	53553	Λ	[special marker for last link]

Thus locations 0 through 100 form the first available block; after the reserved areas 101–290 and 291–631 shown in Fig. 42, we have more free space in location 632–673; etc.

As for question (b), if we want n consecutive words, clearly we must locate some block of $m \geq n$ available words and reduce its size to $m - n$. (Furthermore, when $m = n$, we must also delete this block from the list.) There may be several blocks with n or more cells, and so the question becomes *which* area should be chosen?

Two principal answers to this question suggest themselves: We can use the "best-fit" method or the "first-fit" method. In the former case, we decide to choose an area with m cells, where m is the smallest value present which is n or more. This usually requires searching the entire list of available space before a decision can be made. The "first-fit" method, on the other hand, simply chooses the first area encountered that has $\geq n$ words.

Historically, the best-fit method was widely used for several years; this naturally appears to be a good policy since it saves the larger available areas for a later time when they might be needed. But several objections to the best-fit technique can be raised: It is rather slow, since it requires a fairly long search; if "best fit" is not substantially better than "first fit" for other reasons, this extra searching time is not worth while. More importantly, the best-fit method tends to increase the number of very small blocks, and proliferation of small blocks is usually undesirable. There are certain situations in which the first-fit technique is demonstrably better than the best-fit method; for example, suppose we are given just two available areas of memory, of sizes 1300 and 1200, and suppose there are subsequent requests for blocks of sizes 1000, 1100, and 250:

memory request	*available areas, "first fit"*	*available areas, "best fit"*
—	1300, 1200	1300, 1200
1000	300, 1200	1300, 200
1100	300, 100	200, 200
250	50, 100	stuck

(1)

For these reasons the first-fit method can be recommended.

Algorithm A (*First-fit method*). Let AVAIL point to the first available block of storage, and suppose that each available block with address P has two fields: SIZE(P), the number of words in the block; and LINK(P), a pointer to the next available block. The last pointer is Λ. This algorithm searches for and reserves a block of N words, or reports failure.

A1. [Initialize.] Set Q ← LOC(AVAIL). (Throughout the algorithm we use two pointers, Q and P, which are generally related by the condition P = LINK(Q). *We assume that*

$$\texttt{LINK(LOC(AVAIL))} = \texttt{AVAIL}.)$$

A2. [End of list?] Set P ← LINK(Q). If P = Λ, the algorithm terminates unsuccessfully; there is no room for a block of N consecutive words.

A3. [Is SIZE enough?) If SIZE(P) ≥ N, go to A4; otherwise set Q ← P and return to step A2.

A4. [Reserve N.] Set K ← SIZE(P) − N. If K = 0, set LINK(Q) ← LINK(P). (This, of course, removes an empty area from the list.) Set SIZE(P) ← K. The algorithm terminates successfully, having reserved an area of length N beginning with location P + K. ▮

This algorithm is certainly straightforward enough. However, a significant improvement in its running speed can be made with only a rather slight change in strategy. This improvement is quite important, and the reader will find it a pleasure to discover it for himself (see exercise 6).

Algorithm A may be used whether storage allocation is desired for small N or large N. Let us temporarily assume, however, that we are primarily interested in *large* values of N. Then note what happens when SIZE(P) is equal to N + 1 in that algorithm: we get to step A4 and reduce SIZE(P) to 1. In other words, an available block of size 1 has just been created; this block is so small it is virtually useless, and it just clogs up the system. We would have been better off if we had reserved the whole block of N + 1 words, instead of saving the extra word; it is often better to expend a few words of memory to avoid handling some unimportant details. Similar remarks apply to blocks of N + K words when K is very small.

If we allow the possibility of reserving slightly more than N words, it will be necessary to remember how many words have been reserved, so that later when this block becomes available again the entire set of N + K words is freed. This added amount of bookkeeping means that we are tying up space in *every* block in order to make the system more efficient only in certain circumstances when a "tight fit" is found; so the strategy doesn't seem especially attractive. However, a special *control word* as the first word of each variable-size block often turns out to be desirable for other reasons, and so it is usually not unreasonable to expect the SIZE field to be present in the first word of every block whether it is available or not.

In accordance with these conventions, we would modify step A4 above to read as follows:

"**A4′.** [Reserve $\geq$N.] Set K $\leftarrow$ SIZE(P) $-$ N. If K $< c$ (where c is a small positive constant chosen to reflect an amount of storage we are willing to sacrifice in the interests of saving time), set

$$\texttt{LINK(Q)} \leftarrow \texttt{LINK(P)} \qquad \text{and} \qquad \texttt{L} \leftarrow \texttt{P}.$$

Otherwise set

$$\texttt{SIZE(P)} \leftarrow \texttt{K}, \qquad \texttt{L} \leftarrow \texttt{P} + \texttt{K}, \qquad \text{and} \qquad \texttt{SIZE(L)} \leftarrow \texttt{N}.$$

The algorithm terminates successfully, having reserved an area of length N or more beginning with location L."

A value for the constant c of about 8 or 10 is suggested, although very little theory or empirical evidence exists to compare this with other choices. When the best-fit method is being used, the test of K $< c$ is even *more* important than it is to the first-fit method, because tighter fits (smaller values of K) are much more likely to occur, and the number of available blocks should be kept as small as possible for that algorithm.

B. Liberation. Now let us consider the inverse problem: How should we return blocks to the available space list when they are no longer needed?

It is perhaps tempting to dismiss this problem by using "garbage collection" (see Section 2.3.5); we could follow a policy of simply doing nothing until space

runs out, then searching for all the areas currently in use and fashioning a new AVAIL list.

The idea of garbage collection is not to be recommended, however, for all applications. In the first place, we need a fairly "disciplined" use of pointers if we are to be able to guarantee that all areas currently in use will be easy to locate, and this amount of discipline is often lacking in the applications considered here. Secondly, as we have seen before, garbage collection tends to be slow when the memory is nearly full.

There is another more important reason why garbage collection is not satisfactory, due to a phenomenon which did not confront us in our previous discussion of the technique: Suppose that there are two adjacent areas of memory, both of which are available, but because of the garbage-collection philosophy one of them (shown shaded) is not in the AVAIL list.

(2)

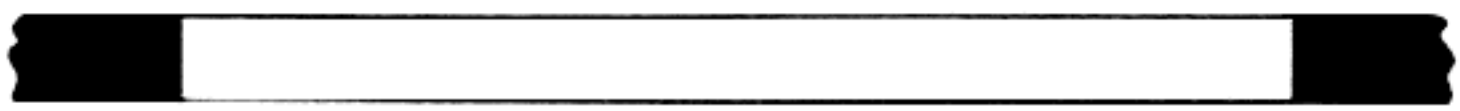
In this diagram, the heavily shaded areas at the extreme left and right are unavailable. We may now reserve a section of the area known to be available:

(3)

If garbage collection occurs at this point, we have two separate free areas,

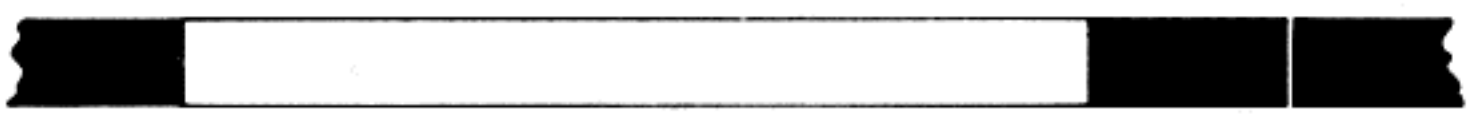
(4)

Boundaries between available and reserved areas have a tendency to perpetuate themselves, and as time goes on the situation gets progressively worse. But if we had used a philosophy of returning blocks to the AVAIL list as soon as they become free, *and collapsing adjacent available areas together*, we would have collapsed (2) into



(5)

and we would have obtained



(6)

which is much better than (4). This phenomenon causes the garbage-collection technique to leave memory more broken up than it should be.

In order to remove this difficulty, it is possible to use garbage collection together with the process of *compacting memory*, i.e., moving all the reserved blocks into consecutive locations, so that all available blocks come together whenever garbage collection is done. The allocation algorithm now becomes completely trivial by contrast with Algorithm A, since there is only one available block at all times. Even though this technique takes time to recopy all the

locations that are in use, and to change the value of the link fields therein, it can be applied with reasonable efficiency when there is a disciplined use of pointers, and when there is a spare link field in each block for use by the garbage collection algorithms. (See exercise 33.)

Since many applications do not meet these requirements for the feasibility of garbage collection, we shall now study methods for returning blocks of memory to the available space list. The only difficulty in these methods is the collapsing problem: two adjacent free areas should be merged into one. In fact, when an area bounded by two available blocks becomes free, all three areas should be merged together into one. In this way *a good balance is obtained in memory even though storage areas are continually reserved and freed over a long period of time.* (For a proof of this fact, see the "fifty-percent rule" below.)

The problem is to determine whether the areas at either side of the returned block are currently available; and if they are, we want to update the AVAIL list properly. (The latter operation is a little more difficult than it sounds.)

The first solution to these problems is to maintain the AVAIL list in order of increasing memory locations.

Algorithm B (*Liberation with sorted list*). Under the assumptions of Algorithm A, with the additional assumption that the AVAIL list is sorted by memory location (i.e., if P points to an available block and LINK(P) $\neq \Lambda$, then LINK(P) $>$ P), this algorithm adds the block of N consecutive cells beginning at location P0 to the AVAIL list. We naturally assume that none of these N cells is already available.

B1. [Initialize.] Set Q $\leftarrow$ LOC(AVAIL). (See the remarks in step A1 above.)

B2. [Advance P.] Set P $\leftarrow$ LINK(Q). If P $= \Lambda$, or if P $>$ P0, go to B3; otherwise set Q $\leftarrow$ P and repeat step B2.

B3. [Check upper bound.] If P0 $+$ N $=$ P (and P $\neq \Lambda$), set N $\leftarrow$ N $+$ SIZE(P) and set LINK(P0) $\leftarrow$ LINK(P). Otherwise set LINK(P0) $\leftarrow$ P.

B4. [Check lower bound.] If Q $+$ SIZE(Q) $=$ P0 [we assume that

$$\text{SIZE(LOC(AVAIL))} = 0,$$

so that this test always fails when Q $=$ LOC(AVAIL)], set SIZE(Q) $\leftarrow$ SIZE(Q) $+$ N and LINK(Q) $\leftarrow$ LINK(P0). Otherwise set LINK(Q) $\leftarrow$ P0, SIZE(P0) $\leftarrow$ N. ∎

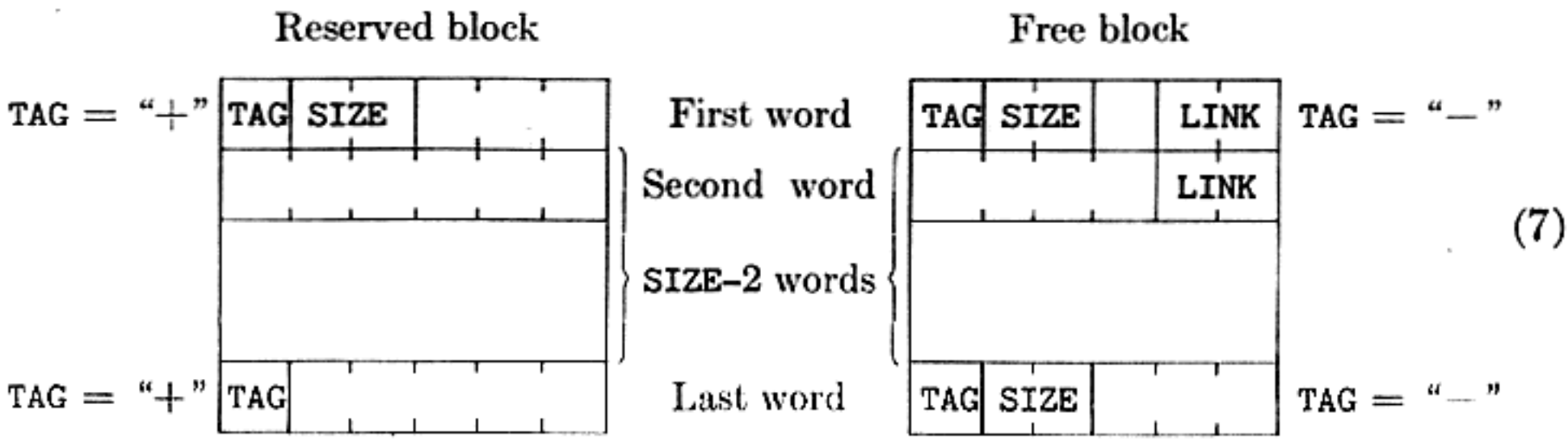
Steps B3 and B4 do the desired collapsing, based on the fact that Q $<$ P0 $<$ P are the beginning locations of three consecutive available areas.

If the AVAIL list is not maintained in order of locations, the reader can see that a "brute force" approach to the collapsing problem would require a complete search through the entire AVAIL list; Algorithm B reduces this to a search through about *half* of the AVAIL list (in step B2) on the average. Exercise 11 shows how Algorithm B can be modified so that, on the average, only about

one-third of the AVAIL list must be searched. But obviously, when the AVAIL list is long, all of these methods are much slower than we want them to be. Isn't there some way to reserve and free storage areas so that we don't need to do extensive searching through the AVAIL list?

We will now consider a method which eliminates all searching when storage is returned and which can be modified, as in exercise 6, to avoid almost all of the searching when storage is reserved. This technique makes use of a TAG field at both ends of each block, and a SIZE field in the first word of each block; this "overhead" is negligible when reasonably large size blocks are being used, although it is perhaps too much of a penalty to pay in situations when the blocks have a very small average size. Another method described in exercise 19 requires only one bit in the first word of each block, at the expense of a little more running time and a slightly more complicated program.

At any rate, let us now assume that we don't mind adding a little bit of control information, in order to save a good deal of time over Algorithm B when the AVAIL list is long. The method we will describe assumes each block has the following form:



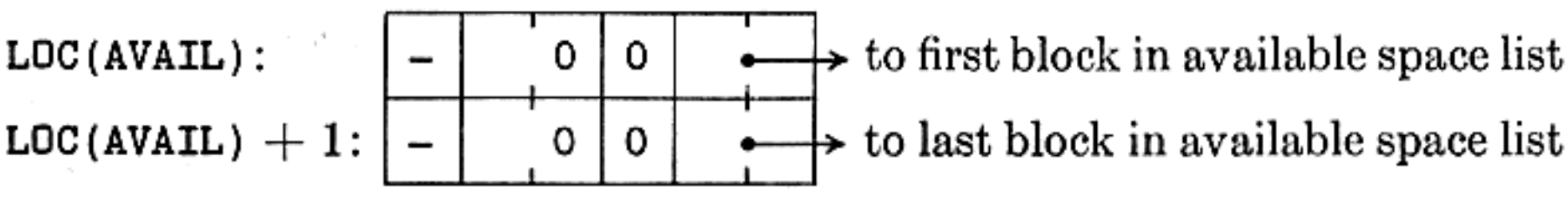
(7)

The idea in the following algorithm is to maintain a doubly linked AVAIL list, so that entries may conveniently be deleted from random parts of the list. The TAG field at either end of a block can be used to control the collapsing process, since we can tell easily whether or not both adjacent blocks are available.

Double linking is achieved in a familiar way, by letting the LINK in the first word point to the next free block in the list, and letting the LINK in the second word point back to the previous block; thus, if P is the address of an available block, we always have

$$\texttt{LINK(LINK(P)} + 1\texttt{)} = \texttt{P} = \texttt{LINK(LINK(P} + 1\texttt{))}. \tag{8}$$

To ensure proper "boundary conditions," AVAIL and the following location are set up as follows:



(9)

A "first-fit" reservation algorithm for this technique may be designed very much like Algorithm A, so we will not give it here (cf. exercise 12). The principal new feature of this method is the way a block can be freed in essentially a fixed amount of time:

Algorithm C (*Liberation with boundary tags*). Assume that blocks of locations have the forms shown in (7), and assume that the AVAIL list is doubly linked, as described above. This algorithm puts the block of locations starting with address P0 into the AVAIL list. If the pool of available storage runs from locations m_0 through m_1, inclusive, the algorithm assumes for convenience that

$$\texttt{TAG}(m_0 - 1) = \texttt{TAG}(m_1 + 1) = \text{“+”}.$$

C1. [Check lower bound.] If TAG(P0 − 1) = "+", go to C3.

C2. [Delete lower area.] Set P ← P0 − SIZE(P0 − 1), and then set

$$\texttt{P1} \leftarrow \texttt{LINK(P)}, \quad \texttt{P2} \leftarrow \texttt{LINK(P + 1)}, \quad \texttt{LINK(P1 + 1)} \leftarrow \texttt{P2},$$
$$\texttt{LINK(P2)} \leftarrow \texttt{P1}, \quad \texttt{SIZE(P)} \leftarrow \texttt{SIZE(P)} + \texttt{SIZE(P0)}, \quad \texttt{P0} \leftarrow \texttt{P}.$$

C3. [Check upper bound.] Set P ← P0 + SIZE(P0). If TAG(P) = "+", go to C5.

C4. [Delete upper area.] Set

$$\texttt{P1} \leftarrow \texttt{LINK(P)}, \quad \texttt{P2} \leftarrow \texttt{LINK(P + 1)}, \quad \texttt{LINK(P1 + 1)} \leftarrow \texttt{P2},$$
$$\texttt{LINK(P2)} \leftarrow \texttt{P1}, \quad \texttt{SIZE(P0)} \leftarrow \texttt{SIZE(P0)} + \texttt{SIZE(P)}, \quad \texttt{P} \leftarrow \texttt{P} + \texttt{SIZE(P)}.$$

C5. [Add to AVAIL list.] Set

$$\texttt{SIZE(P − 1)} \leftarrow \texttt{SIZE(P0)}, \quad \texttt{LINK(P0)} \leftarrow \texttt{LINK(AVAIL)},$$
$$\texttt{LINK(P0 + 1)} \leftarrow \texttt{LOC(AVAIL)}, \quad \texttt{LINK(LINK(AVAIL) + 1)} \leftarrow \texttt{P0},$$
$$\texttt{LINK(AVAIL)} \leftarrow \texttt{P0}, \quad \texttt{TAG(P0)} \leftarrow \texttt{TAG(P − 1)} \leftarrow \text{“−”}. \ \blacksquare$$

The steps of Algorithm C are straightforward consequences of the storage layout (7); a slightly longer algorithm which is a little faster appears in exercise 15.

C. The "buddy system." We will now describe another approach to dynamic storage allocation, suitable for use with binary computers. This method takes one bit of "overhead" in each block, and it requires all blocks to be of length 1, 2, 4, 8, or 16, etc. If a block is not 2^k words long for some integer k, the next higher power of 2 is chosen and extra unused space is allocated accordingly. When this method is applicable it has an advantage of speed, especially in "real-time" situations, since there is a positive guarantee against searches through a long AVAIL list.

The idea of this method is to keep separate lists of available blocks of each size 2^k, $0 \le k \le m$. The entire pool of memory space under allocation consists

of 2^m words, which we will assume for convenience have the addresses 0 through $2^m - 1$. Originally, the entire block of 2^m words is available. Later, when a block of 2^k words is desired, and if none of this size are available, a larger available block is *split* into two equal parts; ultimately, a block of the right size 2^k will appear. When one block splits into two (each of which is half as large as the original), these two blocks are called *buddies*. Later when both buddies are available again, they coalesce back into a single block; thus the process can be maintained indefinitely (unless we run out of space at some point).

The key fact underlying the practical usefulness of this method is that if we know the address of a block (i.e., the memory location of its first word), and if we also know the size of that block, we know the address of its buddy. For example, the buddy of the block of size 16 beginning in binary location 101110010110000 is a block starting in binary location 101110010100000. To see why this must be true, we first observe that as the algorithm proceeds, *the address of a block of size 2^k is a multiple of 2^k*. In other words, the address in binary notation has at least k zeros at the right. This observation is easily justified by induction: if it is true for all blocks of size 2^{k+1}, it is certainly true when such a block is halved.

Therefore a block of size, say, 32 has an address of the form $xx\ldots x00000$ (where the x's represent either 0 or 1); if it is split, the newly formed buddy blocks have the addresses $xx\ldots x00000$ and $xx\ldots x10000$. In general, let $\text{buddy}_k(x)$ = address of the buddy of the block of size 2^k whose address is x; we find that

$$\text{buddy}_k(x) = \begin{cases} x + 2^k, & \text{if } x \bmod 2^{k+1} = 0; \\ x - 2^k, & \text{if } x \bmod 2^{k+1} = 2^k. \end{cases} \tag{10}$$

This function is readily computed with the "exclusive or" instruction (sometimes called "selective complement" or "add without carry") usually found on binary computers; cf. exercise 28.

The buddy system makes use of a one-bit TAG field in each block:

$$\begin{aligned} &\texttt{TAG(P)} = 0, && \text{if the block with address P is reserved;} \\ &\texttt{TAG(P)} = 1, && \text{if the block with address P is available.} \end{aligned} \tag{11}$$

Besides this TAG field, which is present in all blocks, *available* blocks also have two link fields, LINKF and LINKB, which are the usual forward and backward links of a doubly linked list; and they also have a KVAL field to specify k when their size is 2^k. The algorithms below make use of the table locations AVAIL[0], AVAIL[1], . . . , AVAIL[m], which serve respectively as the heads of the lists of available storage of sizes 1, 2, 4, . . . , 2^m. These lists are doubly linked, so as usual (see Section 2.2.5) the list heads contain two pointers:

$$\begin{aligned} &\texttt{AVAILF[}k\texttt{]} = \texttt{LINKF(LOC(AVAIL[}k\texttt{]))} = \text{link to rear of AVAIL[}k\text{] list;} \\ &\texttt{AVAILB[}k\texttt{]} = \texttt{LINKB(LOC(AVAIL[}k\texttt{]))} = \text{link to front of AVAIL[}k\text{] list.} \end{aligned} \tag{12}$$

Initially, before any storage has been allocated, we have

$$\begin{aligned} \texttt{AVAILF}[m] &= \texttt{AVAILB}[m] = 0, \\ \texttt{LINKF}(0) &= \texttt{LINKB}(0) = \texttt{LOC}(\texttt{AVAIL}[m]), \\ \texttt{TAG}(0) &= 1,\ \texttt{KVAL}(0) = m \end{aligned} \tag{13}$$

(indicating a single available block of length 2^m, beginning in location 0), and also

$$\texttt{AVAILF}[k] = \texttt{AVAILB}[k] = \texttt{LOC}(\texttt{AVAIL}[k]), \quad \text{for} \quad 0 \le k < m \tag{14}$$

(indicating empty lists for available blocks of lengths 2^k for all $k < m$).

From this description of the buddy system, the reader may find it enjoyable to design the necessary algorithms for reserving and freeing storage areas by himself, and to compare his solutions with the algorithms given below. Note the comparative ease with which blocks can be halved in the reservation algorithm.

Algorithm R (*Buddy system reservation*). This algorithm finds and reserves a block of 2^k locations, or reports failure, using the organization of the buddy system as explained above.

R1. [Find block.] Let j be the smallest integer in the range $k \le j \le m$ for which $\texttt{AVAILF}[j] \neq \texttt{LOC}(\texttt{AVAIL}[j])$, that is, for which the list of available blocks of size 2^j is not empty. If no such j exists, the algorithm terminates unsuccessfully, since there are no known available blocks of sufficient size to meet the request.

R2. [Remove from list.] Set

$$\texttt{L} \leftarrow \texttt{AVAILF}[j], \quad \texttt{AVAILF}[j] \leftarrow \texttt{LINKF}(\texttt{L}),$$
$$\texttt{LINKB}(\texttt{LINKF}(\texttt{L})) \leftarrow \texttt{LOC}(\texttt{AVAIL}[j]), \quad \text{and} \quad \texttt{TAG}(\texttt{L}) \leftarrow 0.$$

R3. [Split required?] If $j = k$, the algorithm terminates (we have found and reserved an available block starting at address L).

R4. [Split.] Decrease j by 1. Then set

$$\texttt{P} \leftarrow \texttt{L} + 2^j, \quad \texttt{TAG}(\texttt{P}) \leftarrow 1, \quad \texttt{KVAL}(\texttt{P}) \leftarrow j, \quad \texttt{LINKF}(\texttt{P}) \leftarrow \texttt{LOC}(\texttt{AVAIL}[j]),$$
$$\texttt{LINKB}(\texttt{P}) \leftarrow \texttt{LOC}(\texttt{AVAIL}[j]), \quad \texttt{AVAILF}[j] \leftarrow \texttt{AVAILB}[j] \leftarrow \texttt{P}.$$

(This splits a large block and enters the unused half in the $\texttt{AVAIL}[j]$ list which was empty.) Go back to step R3. ∎

Algorithm S (*Buddy system liberation*). This algorithm returns a block of 2^k locations starting in address L to free storage, using the organization of the buddy system as explained above.

S1. [Is buddy available?] Set $\texttt{P} \leftarrow \text{buddy}_k(\texttt{L})$. (See Eq. 10.) If $k = m$ or if $\texttt{TAG}(\texttt{P}) = 0$, or if $\texttt{TAG}(\texttt{P}) = 1$ and $\texttt{KVAL}(\texttt{P}) \neq k$, go to S3.

S2. [Combine with buddy.] Set

$$\texttt{LINKF(LINKB(P))} \leftarrow \texttt{LINKF(P)}, \qquad \texttt{LINKB(LINKF(P))} \leftarrow \texttt{LINKB(P)}.$$

(This removes block P from the AVAIL[k] list.) Then set $k \leftarrow k + 1$, and if P $<$ L set L $\leftarrow$ P. Return to S1.

S3. [Put on list.] Set

$$\texttt{TAG(L)} \leftarrow 1, \quad \texttt{LINKF(L)} \leftarrow \texttt{AVAILF[}k\texttt{]}, \quad \texttt{LINKB(AVAILF[}k\texttt{])} \leftarrow \texttt{L},$$
$$\texttt{KVAL(L)} \leftarrow k, \quad \texttt{LINKB(L)} \leftarrow \texttt{LOC(AVAIL[}k\texttt{])}, \quad \texttt{AVAILF[}k\texttt{]} \leftarrow \texttt{L}.$$

(This puts block L on the AVAIL[k] list.) ∎

D. Comparison of the methods. Hardly any mathematical analysis of these dynamic storage-allocation algorithms has been carried out so far; the only result which we can prove is an interesting phenomenon, the "fifty-percent rule":

If Algorithms A and B are used continually in such a way that the system tends to an equilibrium condition, where there are N reserved blocks in the system, on the average, each with an independent lifetime, and where the quantity K *in Algorithm A takes on nonzero values (or, more generally, values $\geq c$ as in step A4′) with probability p, then the average number of available blocks tends to approximately $\frac{1}{2}pN$.*

This rule tells us approximately how long the AVAIL list will be. When the quantity p is near 1—this will happen if c is very small and if the block sizes are infrequently equal to each other—we have about half as many available blocks as unavailable ones; hence the name "fifty-percent rule."

It is not hard to derive this rule. Consider the following memory map:

This shows the reserved blocks divided into three categories:

A: when freed, the number of available blocks will decrease by one;
B: when freed, the number of available blocks will not change;
C: when freed, the number of available blocks will increase by one.

Now let N be the number of reserved blocks, and let M be the number of available ones; let A, B, and C be the number of blocks of the types identified above. We have

$$\begin{aligned} N &= A + B + C \\ M &= \tfrac{1}{2}(2A + B + \epsilon) \end{aligned} \tag{15}$$

where $\epsilon = 0$, 1, or 2 depending on conditions at the lower and upper boundaries. To derive the fifty-percent rule, we set

probability that M increases by one = probability that M decreases by one

(or, more precisely, the average change in M per unit time is set to zero during equilibrium). This leads to

$$C = A + (1 - p)N;$$

and by (15), with ϵ assumed to be zero (since M and N are assumed to be reasonably large), we get

$$N - 2M + A = A + (1 - p)N. \tag{16}$$

The fifty-percent rule follows. This derivation shows in fact that when M is momentarily less than $\frac{1}{2}pN$, there is higher probability that M will increase than that it will decrease, and conversely.

Besides this interesting rule, our knowledge of the performance of these algorithms is based solely on Monte Carlo experiments. The reader will find it instructive to conduct his own simulation experiments when he is choosing between storage allocation algorithms for a particular machine and a particular application or class of applications. The author carried out several such experiments just before writing this section (and, indeed, the fifty-percent rule was noticed during these experiments before a proof for it was found); let us briefly examine the methods and results of these experiments here.

The basic simulation program ran as follows, with `TIME` initially zero and with the memory area initially all available:

P1. Advance `TIME` by 1.

P2. Free all blocks in the system which are scheduled to be freed at the current value of `TIME`.

P3. Calculate two quantities S (a random size) and T (a random "lifetime"), based on some probability distributions, using the methods of Chapter 3.

P4. Reserve a new block of length S, which is due to be freed at (`TIME` + T). Return to P1. ▮

Whenever `TIME` was a multiple of 200, detailed statistics about the performances of the reservation and liberation algorithms were printed. The same sequence of values of S and T was used for each pair of algorithms tested. After `TIME` advanced past 2000, the system usually had reached a more or less steady state which gave every indication of being maintained indefinitely thereafter. However, depending on the total amount of storage available and on the distributions of S and T in step P3, the allocation algorithms would occasionally fail to find enough space and the simulation experiment was then terminated.

Let M be the total number of memory locations available, and let $\overline{S}$, $\overline{T}$ denote the average values of S and T in step P3. It is easy to see that the expected number of unavailable words of memory at any given time is $\overline{S} \cdot \overline{T}$, once `TIME` is sufficiently large. When $\overline{S} \cdot \overline{T}$ was greater than about $\frac{2}{3}M$ in the

experiments, memory overflow usually occurred, often before M words of memory were actually needed. The memory was able to become over 90 percent filled when the block size was small compared to M, but when the block sizes were allowed to exceed $\frac{1}{3}M$ (as well as taking on much smaller values) the memory tended to become "full" when less than $\frac{1}{2}M$ locations were in fact needed. Empirical evidence suggests strongly that *block sizes larger than* $\frac{1}{10}M$ *should not be used with dynamic storage allocation* if effective operation is expected.

The experiments were conducted with three size distributions for S:

(*S1*) An integer chosen uniformly between 100 and 2000;
(*S2*) Sizes (1, 2, 4, 8, 16, 32) chosen with respective probabilities ($\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{32}$, $\frac{1}{32}$);
(*S3*) Sizes (10, 12, 14, 16, 18, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 500, 1000, 2000, 3000, 4000) were selected with equal probability.

The time distribution T was usually a random integer chosen uniformly between 1 and t, for fixed $t = 10$, 100, or 1000.

Experiments were also made by choosing T uniformly between 1 and min ($\lfloor \frac{5}{4}U \rfloor$, 12500) in step P3, where U is the number of time units remaining until the first reserved block now in the system is scheduled to be freed. This time distribution was meant to simulate an "almost-last-in-first-out" behavior: for if T were always chosen $\leq U$, the storage allocation system would degenerate into simply a stack operation where no complex algorithms are necessary. (See exercise 1.) In this case, T is chosen greater than U about 20 percent of the time, so we have almost, but not quite, a stack operation. When this distribution was used, algorithms such as A, B, and C behaved much better than usual; there were rarely, if ever, more than two items in the entire AVAIL list, while there were about 14 reserved blocks. On the other hand, the buddy system algorithms R and S were slower when this distribution was used, because it is necessary to split and to coalesce blocks more frequently in a stack-like operation. The theoretical properties of this time distribution appear to be very difficult to deduce (see exercise 32).

Figure 42, which appeared near the beginning of this section, was the configuration of memory at TIME = 5000, with size distribution (*S1*) and with the time distribution chosen randomly between 1 and 100, using the "first-fit" method just as in Algorithms A and B above. For this experiment, the probability p which enters into the "fifty-percent rule" was essentially 1, so we would expect about half as many available blocks as reserved blocks. Actually Fig. 42 shows 21 available and 53 reserved. This does not disprove the fifty-percent rule: for example, at TIME = 4600 there were 25 available and 49 reserved. The configuration in Fig. 42 merely shows how the fifty-percent rule is subject to statistical variations. The number of available blocks generally ranged between 20 and 30, while the number of reserved blocks was generally between 45 and 55.

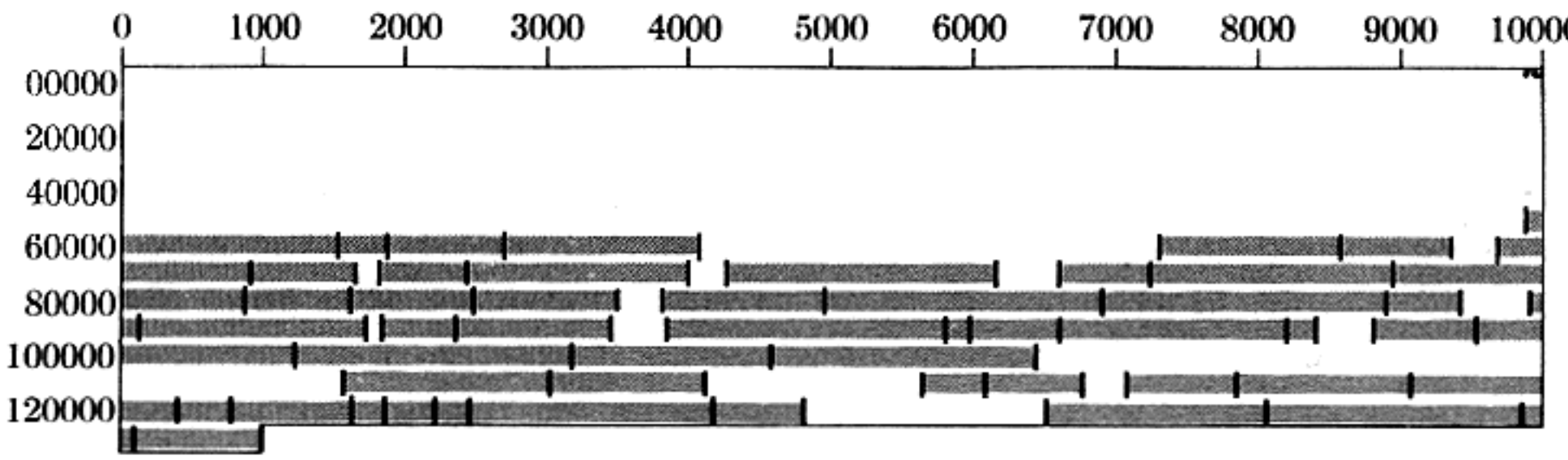


Fig. 43. Memory map obtained with the "best fit" method. (Compare this with Fig. 42, which shows the "first fit" method, and Fig. 44, which shows the "buddy system," for the same sequence of storage requests.)

Figure 43 shows the configuration of memory obtained with *the same data as Fig. 42* but with the "best-fit" method used instead of the "first-fit" method. The constant c in step A4′ was chosen as 16, to eliminate small blocks, and as a result the probability p dropped to about 0.7 and there were fewer available areas.

When the time distribution was changed from 1 to 1000 instead of 1 to 100, situations precisely analogous to those shown in Figs. 42 and 43 were obtained, with all appropriate quantities approximately multiplied by 10. For example, there were 515 reserved blocks; and 240 free blocks in the equivalent of Fig. 42, 176 free blocks in the equivalent of Fig. 43.

In all experiments comparing the best-fit and first-fit methods, the latter always appeared to be superior. When memory size was exhausted, the first-fit method actually stayed in action longer than the best-fit method before memory overflow occurred, in most instances.

The buddy system was also applied to the same data that led to Figs. 42 and 43, and Fig. 44 was the result. Here, all sizes in the range 257 to 512 were treated as 512, those between 513 and 1024 were raised to 1024, etc. On the average this means about four thirds as much memory was requested (see exercise 21); the buddy system, of course, works better on size distributions like that of (*S2*) above, instead of (*S1*). Note that there are available blocks of sizes 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, and 2^{14} in Fig. 44.

Simulation of the buddy system showed that it performs much better than first expected. It is clear that the buddy system will sometimes allow two adjacent areas to be available without merging them into one (if they are not "buddies"); but this situation is not present in Fig. 44 and, in fact, it is rare in practice. In cases where memory overflow occurred, memory was 95 percent packed, and this reflects a surprisingly good allocation balance. Furthermore, it was very seldom necessary to split blocks in Algorithm R, or to merge them in Algorithm S; the tree remained much like Fig. 44 with available blocks on the most commonly used levels. It would be most interesting to see mathematical results which account for this behavior.

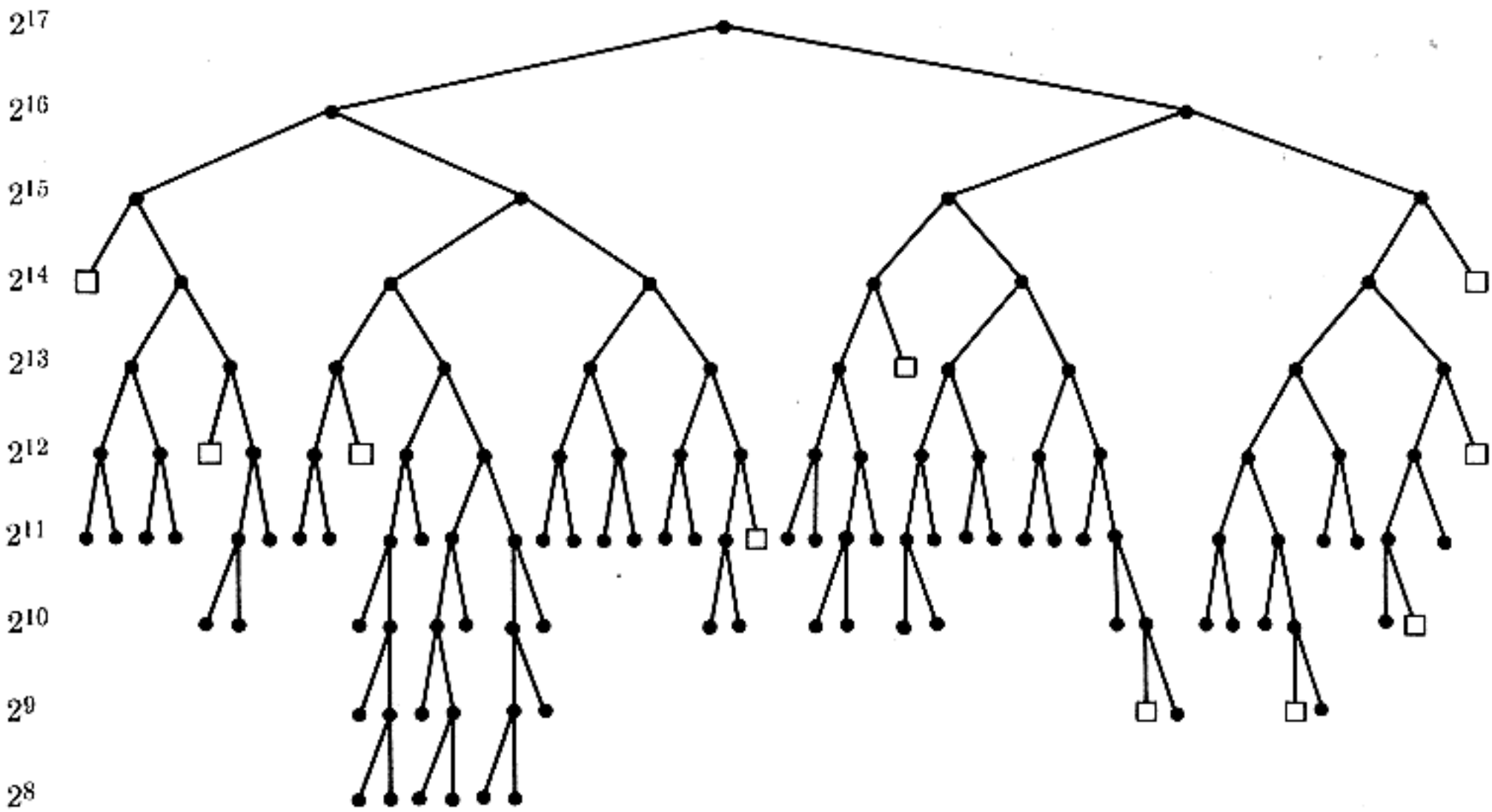


Fig. 44. Memory map obtained with the "buddy system." (The tree structure indicates the division of certain large blocks into "buddies" of half the size. Squares indicate available blocks.)

Another surprise was the excellent behavior of Algorithm A after the modification described in exercise 6; only 2.8 inspections of available block sizes were necessary on the average [using size distribution (*S1*) and times chosen uniformly between 1 and 1000], and more than half of the time only the minimum value, one iteration, was necessary. This was true in spite of the fact that about 250 available blocks were present. The same experiment with Algorithm A unmodified showed about 125 iterations were necessary on the average (so about half of the AVAIL list was being examined each time), and 20 percent of the time 200 or more interations were found to be necessary! This greatly increased efficiency was unexpected since the idea came as an afterthought while examining the output of earlier simulations. [*Note:* Algorithm A unmodified was not nearly so bad on size distribution (*S2*); only six iterations were needed on the average, even when 500 reserved blocks were present. In large part, this is due to the frequent requests for blocks of size 1 and the relatively short AVAIL list.]

The exercises below include MIX programs for the two principal algorithms which are recommended as a consequence of the above remarks—Algorithm A modified as in exercise 12 together with Algorithm C, as compared with the buddy system—and here are the approximate results:

	Time for reservation	Time for liberation
Boundary tag system:	$33 + 7A$	18, 29, 31, or 34
Buddy system:	$19 + 25R$	$27 + 26S$

Here $A \geq 1$ is the number of iterations necessary in searching for an available block which is large enough; $R \geq 0$ is the number of times a block is split in two (the initial difference of $j - k$ in Algorithm R); and $S \geq 0$ is the number of times buddy blocks are reunited during Algorithm S. The simulation experiments indicate that under the stated assumptions with size distribution (*S1*) and time chosen between 1 and 1000, we may take $A = 2.8$, $R = S = 0.04$ on the average. (The average values $A = 1.3$, $R = S = 0.9$ were observed when the "almost-last-in-first-out" time distribution was substituted as explained above.) This shows that both methods are quite fast, with the buddy system slightly faster in MIX's case. Remember that the buddy system requires about 40 percent more memory space when block sizes are not constrained to be powers of 2.

A corresponding time estimate for the garbage collection and compacting algorithm of exercise 33 is about 104 units of time to locate a free node, assuming that garbage collection occurs when the memory is approximately half full, and assuming that the nodes have an average length of 5 words with 2 links per node. The pros and cons of garbage collection are discussed in Section 2.3.5. When the memory is not heavily loaded and when the appropriate restrictions are met, garbage collection and compacting is very efficient; for example, on the MIX computer, the garbage collection method is faster than the other two, if the memory space never gets more than about one-third full, and if the nodes are relatively small.

The same simulation techniques were applied also to some other storage allocation algorithms. The other algorithms were so poor by comparison with the algorithms of this section that they will be given only brief mention here:

a) Separate AVAIL lists were kept for each size. A single free block was occasionally split into two smaller blocks when necessary, but no attempt was made to put such blocks together again. The memory map became fragmented into finer and finer parts until it was in terrible shape; a simple scheme like this might work when only two or three block sizes are required, but it is essentially equivalent to doing separate allocation in disjoint areas, one area for each block (node) size.

b) An attempt was made to do "two-level" allocation: The memory was divided into 32 large sectors. A brute-force allocation method was used to reserve large blocks of 1, 2, or 3 (rarely more) adjacent sectors; each large block such as this was subdivided to meet storage requests until no more room was left within the current large block, and then another large block was reserved for use in subsequent allocations. Each large block was returned to free storage only when *all* space within it became available. This method almost always ran out of storage space very quickly.

Although this particular method of "two level" allocation was a failure for the data considered in the author's simulation experiments, there are other circumstances (which occur not infrequently in practice) when a multiple-level

allocation strategy can be beneficial. For example, this occurs with rather large programs that operate in several stages, where it is known that certain types of nodes are needed only within a certain subroutine. It might also be desirable to use quite different allocation strategies for different classes of nodes in the same program. The idea of allocating storage by "zones," with possibly different strategies employed in each zone and with the ability to free an entire zone at once, is discussed by Douglas T. Ross in his article, "The AED Free Storage Package," *CACM* **10** (1967), 481–492.

E. Overflow. What do we do when no more room is available? Suppose there is a request for, say, n consecutive words, when all available blocks are too small. The first time this happens, there are usually more than n available locations present, but they are not consecutive; "compacting memory" (i.e., moving some of the locations which are in use, so that all the available locations are brought together) would mean we could continue processing. But compacting is slow; and the vast majority of cases in which the "first-fit" method runs out of room actually would soon thereafter run completely out of space anyway, no matter how much compacting and re-compacting is done. Therefore it is generally not worth while to write a compacting program, except under special circumstances in connection with garbage collection, as in exercise 33. If overflow is expected to occur, some method for removing items from memory and storing them on an external memory device can be used, with provision for bringing the information back again when it is needed. This implies that all programs referring to the dynamic memory area must be severely restricted with regard to the allowable references they make to other blocks, and special computer hardware (e.g., interrupt on absence of data, or automatic "paging") is generally required for efficient operation under these conditions.

Some decision procedure is necessary to decide which blocks are the most likely candidates for removal. One idea is to use a memory cell as an artificial "timer": the contents of this cell is increased by one periodically as a program is executed, and although it does not keep strict units of time it is possible to tell that one event has occurred before another if its "time" is smaller. When a block is accessed, the current timer reading is stored in the block, and thus an algorithm can be devised to place those blocks that are not recently active into the auxiliary memory. Instead of such a timer cell, the same effect can be achieved with a doubly linked list of reserved blocks, in which a block is moved up to the front of the list each time it is accessed; then the blocks are effectively sorted in order of their last access, and the block at the rear of the list is the one to remove first.

Further discussion of this problem belongs under the heading of executive control programs ("operating systems"), and so it is beyond the scope of this book. For most applications of dynamic storage allocation, let us hope there is plenty of memory space available.

EXERCISES

1. [*20*] What simplifications can be made to the reservation and liberation algorithms of this section, if storage requests always appear in a "last-in-first-out" manner, i.e., if no reserved block is freed until after all blocks that were reserved subsequently have already been freed?

2. [*HM23*] (E. Wolman.) Suppose that we want to choose a fixed node size for variable length items, and suppose also that when each node has length k and when an item has length l, it takes $\lceil l/(k - b) \rceil$ nodes to store this item. (Here b is a constant, signifying that b words of each node contain control information, such as a link to the next node.) If the average length l of a record is L, what choice of k minimizes the average amount of storage space required? (Assume that the average value of $(l/(k - b)) \bmod 1$ is equal to $\frac{1}{2}$, for any fixed k, as l varies.)

3. [*40*] By computer simulation, compare the best-fit, first-fit, and "worst-fit" methods of storage allocation; in the latter method, the largest available block is always chosen. Is there any significant difference in the memory usage?

4. [*22*] Write a MIX program for Algorithm A, paying special attention to making the inner loop fast. Assume that the SIZE field is (4:5), the LINK field is (0:2), and $\Lambda < 0$.

▶ **5.** [*18*] Suppose it is known that N is always 100 or more in Algorithm A. Would it be a good idea to set $c = 100$ in the modified step A4′?

▶ **6.** [*23*] After Algorithm A has been used repeatedly, there will be a strong tendency for blocks of small SIZE to remain at the front of the AVAIL list, so that it will often be necessary to search quite far into the list before finding a block of length N or more. For example, note how the size of the blocks essentially increases in Fig. 42, for both reserved and free blocks, from the beginning of memory to the end. (The AVAIL list used while Fig. 42 was being prepared was kept sorted by order of location, as required by Algorithm B.) Can you suggest a way to modify Algorithm A so that (a) short blocks won't tend to accumulate in a particular area, and (b) the AVAIL list may still be kept in order of increasing memory locations, for purposes of algorithms like Algorithm B?

7. [*10*] The example (1) shows that sometimes "first fit" can definitely be superior to "best fit." Give a similar example which shows a case where "best fit" is superior to "first fit."

8. [*21*] Show how to modify Algorithm A in a simple way to obtain an algorithm for the "best-fit" method, instead of "first fit."

9. [*26*] In what ways could a reservation algorithm be designed using the "best-fit" method, that avoids searching the whole AVAIL list? (Try to think of ways that cut down the necessary search as much as possible.)

10. [*22*] Show how to modify Algorithm B so that the block of N consecutive cells beginning at location P0 is made available, without assuming that none of these N cells is already available; assume, in fact, that the area being freed may actually overlap several blocks that are already free.

11. [*M25*] Show that the improvement to Algorithm A suggested in the answer to exercise 6 also can be used to lead to a slight improvement in Algorithm B, which cuts

the average length of search from half the length of the AVAIL list to one-third this length. (Assume that the block being freed will be inserted into a random place within the sorted AVAIL list.)

▸ **12.** [*20*] Modify Algorithm A so that it follows the conventions of (7), uses the modified step A4′ described in the text, and also incorporates the improvement of exercise 6.

13. [*21*] Write a MIX program for the algorithm of exercise 12.

14. [*21*] What difference would it make to Algorithm C and the algorithm of exercise 12, (a) if the SIZE field were not present in the last word of a free block? or (b) if the SIZE field were not present in the first word of a reserved block?

▸ **15.** [*24*] Show how to speed up Algorithm C at the expense of a slightly longer program, by not changing any more links than absolutely necessary in each of four cases depending on whether TAG(P0−1), TAG(P0 + SIZE(P0)) are plus or minus.

16. [*24*] Write a MIX program for Algorithm C, incorporating the ideas of exercise 15.

17. [*10*] What should be the contents of LOC(AVAIL) and LOC(AVAIL) + 1 in (9) when there are no available blocks present?

▸ **18.** [*20*] Figs. 42 and 43 were obtained using the same data, and essentially the same algorithms (Algorithms A and B), except that Fig. 43 was prepared by modifying Algorithm A to choose "best fit" instead of "first fit." Why did this cause Fig. 42 to have a large available area in the *higher* locations of memory, while in Fig. 43 there is a large available area in the *lower* locations?

▸ **19.** [*24*] Suppose that blocks of memory have the form of (7), except without the TAG or SIZE fields required in the last word of the block. Suppose further that the following simple algorithm is being used to make a reserved block free again: "Q ← LINK(AVAIL), LINK(P0) ← Q, LINK(P0+1) ← LOC(AVAIL), LINK(Q+1) ← P0, LINK(AVAIL) ← P0, TAG(P0) ← "−"." (This algorithm does nothing about collapsing adjacent areas together.)

Show that it is possible to design a reservation algorithm similar to Algorithm A, which does the necessary collapsing of adjacent free blocks while searching the AVAIL list, and at the same time it avoids any unnecessary fragmentation of memory as in (2), (3), and (4).

20. [*00*] Why is it desirable to have the AVAIL[*k*] lists in the buddy system doubly linked, instead of simply having straight linear lists?

21. [*HM25*] Examine the ratio a_n/b_n, where a_n is the sum of the first n terms of $1 + 2 + 4 + 4 + 8 + 8 + 8 + 8 + 16 + 16 + \cdots$, and b_n is the sum of the first n terms of $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + \cdots$, as n goes to infinity.

▸ **22.** [*21*] The text repeatedly states that the buddy system allows only blocks of size 2^k to be used, and exercise 21 shows this can lead to a substantial increase in the storage required. But if an 11-word block is needed in connection with the buddy system, why couldn't we find a 16-word block and divide it into an 11-word piece together with two free blocks of sizes 4 and 1?

23. [*05*] What is the binary address of the buddy of the block of size 4 whose binary address is 011011110000? What would it be if the block were of size 16 instead of 4?

24. [*20*] According to the algorithm in the text, the largest block (of size 2^m) has no buddy, since it represents all of storage. Would it be correct to define $\text{buddy}_m(0) = 0$

(i.e., to make this block its own buddy), and then to simplify step S1 to avoid testing the condition $k = m$?

▶ **25.** [*22*] Criticize the following idea: "Dynamic storage allocation using the buddy system will never reserve a block of size 2^m in practical situations (since this would fill the whole memory), and, in general, there is a maximum size 2^n for which no blocks of greater size are ever to be reserved. Therefore it is a waste of time to start with such large blocks available, and to combine buddies in Algorithm S when the combined block has a size larger than 2^n."

▶ **26.** [*21*] Explain how the buddy system could be used for dynamic storage allocation in memory locations 0 through `M` − 1 even though `M` does not have the form 2^m as required in the text.

27. [*24*] Write a `MIX` program for Algorithm R, and determine its running time.

28. [*25*] Assume that `MIX` is a binary computer, with a new operation code `XOR` defined as follows (using the notation of Section 1.3.1): "C = 5, F = 5; for each bit position in location M which equals 1, the corresponding bit position in register A is complemented (changed from 0 to 1 or 1 to 0)."

Write a `MIX` program for Algorithm S, and determine its running time.

29. [*20*] Could the buddy system be modified to avoid the tag bit in each reserved block?

30. [*M50*] Analyze the properties of the buddy system, in particular the average speed of Algorithms R and S, given reasonable distributions for the sequence of storage requests.

31. [*M40*] Can a storage allocation system analogous to the buddy system be designed using the Fibonacci sequence instead of powers of two? (Thus, we might start with F_m available words, and split an available block of F_k words into two buddies of respective lengths F_{k-1} and F_{k-2}.)

32. [*HM50*] Determine $\lim_{n\to\infty} \alpha_n$, if it exists, where α_n is the mean value of t_n in a sequence defined as follows: Let

$$g_k = \lfloor \tfrac{5}{4} \min\left(10000, f(t_{k-1} - 1), f(t_{k-2} - 2), \ldots, f(t_1 - (k-1))\right) \rfloor,$$

where $f(x) = x$ if $x > 0$, $f(x) = \infty$ if $x \leq 0$. The quantity t_k takes on any of the values $1, 2, \ldots, g_k$ with probability $1/g_k$. (*Note:* Some limited empirical tests indicate that α_n will be approximately 14, but this is not guaranteed to be very accurate.)

33. [*28*] (Garbage collection and compacting.) Assume that memory locations 1 through `AVAIL` − 1 are being used as a storage pool for nodes of varying sizes, having the following form: The first word of `NODE(P)` contains the fields

`SIZE(P)` = number of words in `NODE(P)`;
`T(P)` = number of link fields in `NODE(P)`; `T(P)` < `SIZE(P)`;
`LINK(P)` = special link field for use only during garbage collection.

The node immediately following `NODE(P)` in memory is `NODE(P + SIZE(P))`. Assume that the only fields in `NODE(P)` which are used as links to other nodes are `LINK(P + 1)`, `LINK(P + 2)`, ..., `LINK(P + T(P))`, and each of these link fields is either Λ or the address of the first word of another node. Finally, assume that there is one further link variable in the program, called `USE`, and it points to one of the nodes in the storage pool.

Design an algorithm which (a) determines all nodes accessible directly or indirectly from the variable USE, (b) moves these nodes into memory locations 1 through K − 1, for some K, changing all links so that structural relationships are preserved, and (c) sets AVAIL ← K.

For example, consider the following contents of memory, where INFO(L) denotes the contents of location L, excluding LINK(L):

```
 1:  SIZE = 2, T = 1
 2:  LINK = 6, INFO = A
 3:  SIZE = 3, T = 1
 4:  LINK = 8, INFO = B
 5:  CONTENTS = C
 6:  SIZE = 2, T = 0
 7:  CONTENTS = D
 8:  SIZE = 3, T = 2
 9:  LINK = 8, INFO = E
10:  LINK = 3, INFO = F
```

AVAIL = 11, USE = 3. Your algorithm should transform this into

```
1:  SIZE = 3, T = 1
2:  LINK = 4, INFO = B
3:  CONTENTS = C
4:  SIZE = 3, T = 2
5:  LINK = 4, INFO = E
6:  LINK = 1, INFO = F
```

AVAIL = 7, USE = 1.

34. [*29*] Write a MIX program for the algorithm of exercise 33, and determine its running time.

35. [*45*] Design an efficient, general-purpose system oriented towards the editing of fairly long sequences of characters.

36. [*22*] Contrast the dynamic storage allocation methods of this section with the techniques for variable-size sequential lists discussed at the end of Section 2.2.2.

▸ **37.** [*25*] ("Beat the allocator.") Assume that storage is being allocated dynamically by an algorithm that never relocates reserved blocks. Show that, no matter what algorithm is used, it is possible to have a series of requests which overflows a memory of $2n + 1$ cells, although no more than $n + 2$ cells are requested at any one time. (For example if $n = 2$, and if memory locations a_1, a_2, a_3, a_4, a_5 are available, we might first reserve four blocks of one cell each. Then if a_3 is unavailable, we liberate all other cells and request a block of 3 cells in a row. If a_3 is available, we liberate a_1 and a_5 and request a block of size 2. In both cases the request cannot be filled.)

Linear lists and rectangular arrays of information kept in consecutive memory locations were widely used from the earliest days of stored-program computers, and the earliest treatises on programming gave the basic algorithms for traversing these structures. [For example, see J. von Neumann, *Collected Works*, vol. V, 113–116 (written 1947); M. V. Wilkes, D. J. Wheeler, S. Gill, *The Preparation of Programs for an Electronic Digital Computer* (Reading, Mass.: Addison-Wesley, 1951), subroutine V–1.] Before the days of index registers, operations on sequential linear lists were done by performing arithmetic on the machine language instructions themselves, and this type of operation was one of the early motivations for having a computer whose programs share memory space with the data they manipulate.

Techniques which permit variable-length linear lists to share sequential locations, in such a way that they shift back and forth when necessary, as described in Section 2.2.2, were apparently a much later invention. J. Dunlap of Digitek Corporation developed these techniques in 1963 in connection with the design of a series of compiler programs; about the same time the idea independently appeared in the design of a COBOL compiler at IBM Corporation, and a collection of related subroutines called CITRUS was subsequently used at various installations. The techniques remained unpublished until after they had been independently developed by Jan Garwick of Norway; see *BIT* **4** (1964), 137–140.

The idea of having linear lists in *non*sequential locations seems to have originated in connection with the design of computers with drum memories; the first such computer was developed by General Electric Corporation during 1946–1948, and the IBM 650 was the most notable later machine of this type. After executing the instruction in location n, such a computer is usually not ready to get its next instruction from location $n + 1$ because the drum has already rotated past this point. Depending on the instruction being performed, the most favorable position for the next instruction might be $n + 7$ or $n + 18$, etc., and the machine can operate up to six or seven times faster if its instructions are optimally located rather than consecutive. [For a discussion of the interesting problems concerning best placement of these instructions, see the author's article in *JACM* **8** (1961), 119–150.] Therefore the machine design provides an extra address field in each machine language instruction, to serve as a link to the next instruction. Such a machine is called a "one-plus-one-address computer," as distinguished from `MIX` which is a "one-address computer." The design of one-plus-one-address computers is apparently the first appearance of the linked-list idea within computer programs, although the dynamic insertion and deletion operations which we have used so frequently in this chapter were still unknown.

Linked memory techniques were really born when A. Newell, C. Shaw, and H. Simon began their investigations of heuristic problem-solving by machine.

As an aid to writing programs that searched for proofs in mathematical logic, they designed the first "list-processing" language IPL-II in the spring of 1956. (IPL stands for *Information Processing Language.*) This was a system which made use of links and included important concepts like the list of available space, but the concept of stacks was not yet well developed; IPL-III was designed a year later, and it included "push down" and "pop up" for stacks as important basic operations. (For references to IPL-II see "The Logic Theory Machine," *IRE Transactions on Information Theory* **IT–2** (Sept. 1956), 61–70; see also *Proc. Western Joint Comp. Conf.* (Feb. 1957), 218–240. Material on IPL-III first appeared in course notes given at the University of Michigan in the summer of 1957.)

The work of Newell, Shaw, and Simon inspired many other people to use linked memory (which was often at the time referred to as NSS memory), mostly for problems dealing with simulation of human thought processes. Gradually, the techniques became recognized as basic computer-programming tools; the first article describing the usefulness of linked memory for "down-to-earth" problems was published by J. W. Carr, III, in *CACM* **2** (Feb. 1959), 4–6. Carr pointed out in this article that linked lists can readily be manipulated in ordinary programming languages, without requiring sophisticated subroutines or interpretive systems. See also G. A. Blaauw, *IBM J. Res. and Dev.* **3** (1959), 288–301.

At first, one-word nodes were used for linked tables, but about 1959 the usefulness of several consecutive words per node and "multilinked" lists was gradually being discovered by several different groups of people. The first article dealing specifically with this idea was published by D. T. Ross, *CACM* **4** (1961), 147–150; at that time he used the term "plex" for what has been called a "node" in this chapter, but he subsequently has used the word "plex" in a different sense to denote a class of nodes combined with associated algorithms for their traversal.

Notations for referring to fields within nodes are generally of two kinds: the name of the field either precedes or follows the pointer designation. Thus, while we have written "`INFO(P)`" in this chapter, some other authors write, for example, "`P.INFO`". At the time this chapter was prepared, the two notations seemed to be equally prominent. The notation adopted here has been used in the author's lectures since 1962 and it has also been independently devised by numerous other people, primarily because it is natural to use mathematical functional notation to describe attributes of a node. (For example, see the article by N. Wirth and C. A. R. Hoare, *CACM* **9** (1966), 413–432.) Note that "`INFO(P)`" is pronounced "info of `P`" in conventional mathematical verbalization, just as $f(x)$ is rendered "f of x." The alternative notation `P.INFO` has less of a natural flavor, since it tends to put the emphasis on `P`, although it can be read "`P`'s info"; the reason `INFO(P)` seems to be more natural is apparently the fact that `P` is variable, but `INFO` has a fixed significance when the notation is employed. By analogy, we could consider a vector $A = (A[1], A[2], \ldots, A[100])$ to be

a node having 100 fields named 1, 2, . . . , 100. Now the second field would be referred to as "2(P)" in our notation, where P points to the vector A; but if we are referring to the jth element of the vector, we find it more natural to write $A[j]$, putting the variable quantity "j" second. Similarly it seems most appropriate to put the variable quantity "P" second in the notation INFO(P).

Perhaps the first people to recognize that the concepts "stack" (last-in-first-out) and "queue" (first-in-first-out) are important objects of study were cost accountants interested in reducing income tax assessments; for a discussion of the "LIFO" and "FIFO" methods of pricing inventories, see any intermediate accounting textbook, e.g., C. F. and W. J. Schlatter, *Cost Accounting* (New York: Wiley, 1957), Chapter 7. In 1947 A. M. Turing developed a stack, called Reversion Storage, for use in subroutine linkage (see Section 1.4.5). No doubt simple uses of stacks kept in sequential memory locations were common in computer programming from the earliest days, since a stack is such a simple and natural concept. The programming of stacks in linked form appeared first in IPL, as stated above; the name "stack" stems from IPL terminology (although "pushdown list" was the more official IPL wording), and it was also independently introduced in Europe by E. W. Dijkstra. "Deque" is a term introduced by E. J. Schweppe.

The origin of circular and doubly linked lists is obscure; presumably these ideas occurred naturally to many people. A strong factor in the popularization of these techniques was the existence of general List-processing systems based on them [principally the Knotted List Structures, *CACM* **5** (1962), 161–165, and Symmetric List Processor, *CACM* **6** (1963), 524–544, of J. Weizenbaum].

Various methods for addressing and traversing multidimensional arrays of information were developed independently by clever programmers since the earliest days of computers, and thus another part of the unpublished computer folklore was born. This subject was first surveyed in print by H. Hellerman, *CACM* **5** (1962), 205–207. See also J. C. Gower, *Comp. J.* **4** (1962), 280–286.

Tree structures represented explicitly in computer memory were described for the first time in applications to algebraic formula manipulation. The A-1 compiler language, developed by G. M. Hopper in 1951, used arithmetic expressions written in a three-address code; the latter is equivalent to the INFO, LLINK, and RLINK of a binary tree representation. In 1952, H. G. Kahrimanian developed algorithms for differentiating algebraic formulas represented in the A-1 compiler language; see *Symposium on Automatic Programming* (Washington, D.C.: Office of Naval Research, May 1954), 6–14.

Since then, tree structures in various guises have been studied independently by many people in connection with numerous computer applications, but the basic techniques for tree manipulation (not general List manipulation) have seldom appeared in print except in detailed description of particular algorithms. The first general survey was made in connection with a more general study of all data structures by K. E. Iverson and L. R. Johnson [IBM Corp. research reports RC-390, RC-603, 1961; see Iverson, *A Programming Language* (New York: Wiley, 1962), Chapter 3]. See also G. Salton, *CACM* **5** (1962), 103–114.

The concept of *threaded* trees is due to A. J. Perlis and C. Thornton, *CACM* **3** (1960), 195–204. Their paper also introduced the important idea of traversing trees in various orders, and gave numerous examples of algebraic manipulation algorithms. Unfortunately, this important paper was hastily prepared and it contains many misprints. The threaded lists of Perlis and Thornton actually were only "right-threaded trees" in our terminology; binary trees which are threaded in *both* directions were independently discovered by A. W. Holt, *A Mathematical and Applied Investigation of Tree Structures* (Thesis, U. of Pennsylvania, 1963). Postorder and preorder for the nodes of trees were called "normal along order" and "dual along order" by Z. Pawlak, *Colloquium on the Foundation of Mathematics*, etc. (Tihany, 1962, published by Akadémiai Kiadó, Budapest, 1965), 227–238. Preorder was called "subtree order" by Iverson and Johnson in the references cited above. Graphical ways to represent the connection between tree structures and corresponding linear notations were described by A. G. Oettinger, *Proc. Harvard Symp. on Digital Computers and their Applications* (April, 1961), 203–224. Representation of trees in preorder by degrees, with associated algorithms relating this representation to Dewey decimal notation and other properties of trees, was presented by S. Gorn, *Proc. Symp. Math. Theory of Automata* (Brooklyn: Poly. Inst., 1962), 223–240.

The history of tree structures as mathematical entities, together with a bibliography of the subject, is reviewed in Section 2.3.4.6.

At the time this section was written, the most widespread knowledge about information structures was due to programmers' exposure to List processing systems, which have a very important part in this history. The first widely used system was IPL-V (a descendant of IPL-III, developed late in 1959); IPL-V is an interpretive system in which a programmer learns a machine-like language for List operations. At about the same time, FLPL (a set of FORTRAN subroutines for List manipulation, also inspired by IPL but using subroutine calls instead of interpretive language) was developed by H. Gelernter and others. A third system, LISP, was designed by J. McCarthy, also in 1959. LISP is quite different from its predecessors: programs for it are expressed in mathematical functional notation combined with "conditional expressions" (see Chapter 8), then converted into a List representation. Many List processing systems have come into existence since then, of which the most prominent historically is J. Weizenbaum's SLIP; this is a set of subroutines for use in FORTRAN programs, operating on doubly linked Lists.

An article by Bobrow and Raphael, *CACM* **7** (1964), 231–240, may be read as a brief introduction to IPL-V, LISP, and SLIP, and it gives a comparison of these systems. An excellent introduction to LISP has been given by P. M. Woodward and D. P. Jenkins, *Comp. J.* **4** (1961), 47–53. See also the authors' discussions of their own systems, which are each articles of considerable historical importance: "An introduction to IPL-V" by A. Newell and F. M. Tonge, *CACM* **3** (1960), 205–211; "A FORTRAN-compiled List Processing Language" by H. Gelernter, J. R. Hansen, and C. L. Gerberich, *JACM* **7** (1960), 87–101; "Recursive functions of symbolic expressions and their computation by machine,

I" by John McCarthy, *CACM* **3** (1960), 184–195; "Symmetric List Processor" by J. Weizenbaum, *CACM* **6** (1963), 524–544. The latter article includes a complete description of all of the algorithms used in SLIP. In recent years a number of books about these systems have also been written.

Several *string manipulation* systems have also appeared; these are primarily concerned with operations on variable-length strings of alphabetic information (looking for occurrences of certain substrings, etc.). Historically, the most important of these have been COMIT (V. H. Yngve, *CACM* **6** (1963), 83–84) and SNOBOL (D. J. Farber, R. E. Griswold, and I. P. Polonsky, *JACM* **11** (1964), 21–30). Although string manipulation systems have seen wide use and although they are primarily composed of algorithms such as we have seen in this chapter, they play a comparatively small role in the history of the techniques of information structure representation; users of these systems have largely been unconcerned about the details of the actual internal processes carried on by the computer. For a survey of string manipulation techniques, see S. E. Madnick, *CACM* **10** (1967), 420–424.

The IPL-V and FLPL systems for List-processing did not use either a garbage collection or a reference count technique for the problem of shared Lists; instead, each List was "owned" by one List and "borrowed" by all other Lists which referred to it, and a List was erased when its "owner" allowed it to be. Hence, the programmer was enjoined to make sure no List was still borrowing any Lists that were being erased. The reference counter technique for Lists was introduced by G. E. Collins, *CACM* **3** (1960), 655–657; see also the important sequel to this paper, *CACM* **9** (1966), 578–588. Garbage collection was first described in McCarthy's article cited above; see also *CACM* **7** (1964), 38, and an article by Cohen and Trilling, *BIT* **7** (1967), 22–30.

Unfortunately, too few people have realized that manipulation of links is not a magic, complex process that must be done by fancy List manipulation routines. There is still a widespread tendency to feel that the advantages of link manipulation can be obtained only by using a large List-processing system; actually, as we have seen in many examples throughout this chapter, it is quite easy to design linking algorithms which operate more efficiently than a general List-processing subroutine could possibly do. The author hopes that these detailed examples will help to convey a better-balanced attitude towards the use of structural links in data.

Dynamic storage allocation algorithms were in use several years before published information about them appeared. A very readable discussion is given by W. T. Comfort, *CACM* **7** (1964), 357–362 (an article written in 1961). The "boundary-tag" method, introduced in Section 2.5, was designed by the author in 1962 for use in a control program for the B-5000 computer. The "buddy system" was first used by H. Markowitz in connection with the SIMSCRIPT programming system in 1963, and it was independently discovered and published by K. Knowlton, *CACM* **8** (1965), 623–625; see also *CACM* **9** (1966), 616–625. For further discussion of dynamic storage allocation, see the articles by Iliffe

and Jodeit, *Comp. J.* **5** (1962), 200–209; Bailey, Barnett, and Burleson, *CACM* **7** (1964), 339–346; Berztiss, *CACM* **8** (1965), 512–513; and D. T. Ross, *CACM* **10** (1967), 481–492.

A general discussion of information structures and their relation to programming has been given by Mary d'Imperio, "Data Structures and their Representation in Storage," *Annual Review in Automatic Programming* **5** (Oxford: Pergamon Press, 1969). This paper is also a valuable guide to the history of the topic, since it includes a detailed analysis of the structures used in connection with twelve List processing and string manipulation systems. See also the proceedings of two symposia, *CACM* **3** (1960), 183–234 and *CACM* **9** (1966), 567–643, for further historical details. (Several of the individual papers from these proceedings have already been cited above.)

An excellent annotated bibliography, which is primarily oriented towards applications to symbol manipulation and algebraic formula manipulation but which has numerous connections with the material of this chapter, has been compiled by Jean E. Sammet, *Comput. Rev.* **7** (July–August 1966), B1–B31.

In this chapter we have looked at particular types of information structures in great detail, and (lest we fail to see the forest for the trees) it is perhaps wise to take stock of what we have learned and to briefly summarize the general subject of information structures from a broader perspective: Starting with the basic idea of a *node* as an element of data, we have seen many examples which illustrate the fact that it is convenient to represent structural relationships either implicitly (based on the relative order in which nodes are stored in computer memory) or explicitly (by means of links in the nodes, which point to other nodes). The amount of structural information that ought to be represented within the tables of a computer program depends on the operations that are to be performed on the nodes.

The discussion in this chapter does not cover the entire subject of information structures in full generality; at least three important aspects of the subject have not been treated here:

a) It is often necessary or desirable to search through a table to find a node or set of nodes possessing a certain value, and such an operation often has a profound effect on the structure of the table. This situation is explored in detail in Chapter 6.

b) We have primarily been concerned with the internal representation of structure within a computer, and this is obviously only part of the story, since structure must also be represented in the external input and output data. In simple cases, external structure can essentially be treated by the same techniques we have been considering; but the processes of converting between strings of characters and more complex structures are also very important. These processes are analyzed in Chapters 9 and 10.

c) We have primarily discussed representations of structures within a high-speed random-access memory. When slower memory devices (e.g., disks, drums, tapes) are being used, we find that all of the structural problems are

intensified; it is much more crucial to have efficient algorithms and efficient schemes for data representation. It is often necessary to attempt to place "neighboring" nodes, which link to each other, into nearby areas of the memory, etc.; usually the problems are highly dependent on the characteristics of individual machines, so it is difficult to discuss them in general. Hopefully, the simpler examples treated in this chapter will prepare the reader for solving the more difficult problems which arise in connection with less ideal memory devices.

What are the main implications of the subjects treated in this chapter? Perhaps the most important conclusion we can reach is that the ideas we have encountered are not limited to computer programming alone; they apply more generally to everyday life. A collection of nodes containing fields, some of which point to other nodes, appears to be a very good abstract model for structural relationships of all kinds; it shows how we can build up complicated structures from simple ones, and we have seen that corresponding algorithms for manipulating the structure can be designed in a natural manner.

Therefore it seems appropriate to develop much more theory about linked sets of nodes than we know at this time. Perhaps the most obvious way to start such a theory is to define a new kind of abstract machine or "automaton" which deals with linked structures. For example, such an automaton might be defined informally as follows: There are numbers k, l, r, and s, such that the automaton processes nodes containing k link fields and r information fields; it has l link registers and s information registers, which enable it to control the processes it is performing. The information fields and registers may contain any symbols from some given set of information symbols; each of the link fields and link registers either contains Λ or points to a node. The machine can (i) create new nodes (putting a link to the node into a register), (ii) compare information symbols or link values for equality, and (iii) transfer information symbols or link values between registers and nodes. Only nodes pointed to by link registers are immediately accessible. Suitable restrictions on the machine's behavior will make it equivalent to several older species of automata.

Some of the most interesting problems to solve for such devices would be to determine how fast they can solve certain problems, or how many nodes they need to solve certain problems (e.g., to translate certain formal languages). At the time this chapter was written, several interesting results of this kind have been obtained (notably by J. Hartmanis and R. E. Stearns), but only for special classes of "Turing machines" having multiple tapes and read/write heads, etc.; since the Turing machine model is comparatively unrealistic, these results tend to have little to do with practical problems. It is true that, as the number n of nodes created by a linking automaton approaches infinity, we must admit that we don't know how to build such a device physically, since we expect the machine operations will take the same amount of time regardless of the size of n; if linking is represented by using addresses as in a computer memory, it is necessary to put a bound on the number of nodes, since the link fields have a

fixed size. A multitape Turing machine is therefore a more realistic model when n approaches infinity. Yet it seems reasonable to believe that a linking automaton as described above leads to a more appropriate theory of the complexity of algorithms than Turing machines do, even when asymptotic formulas for large n are considered, because the theory is more likely to be relevant for practical values of n.

EXERCISE

1. [*M50*] Explore the properties of linking automata as defined in the text.

You will, I am sure, agree with me that if page 534 finds us only in the second chapter, the length of the first one must have been really intolerable.

—SHERLOCK HOLMES (*The Valley of Fear*, Chapter 1)

ANSWERS TO EXERCISES

NOTES ON THE EXERCISES

1. An average problem for a mathematically inclined reader.

3. See W. J. LeVeque, *Topics in Number Theory* (Addison-Wesley, 1956), vol. 2, chapter 3. (*Note:* One of the men who read a preliminary draft of the manuscript for this book reported that he had discovered a truly remarkable proof, which the margin of his copy was too small to contain.)

SECTION 1.1

1. $t \leftarrow a,\ a \leftarrow b,\ b \leftarrow c,\ c \leftarrow d,\ d \leftarrow t.$

2. After the first time, the values of the variables m, n are the previous values of n, r, respectively; and $n > r$.

3. Algorithm F (*Euclid's algorithm*). Given two positive integers m and n, find their greatest common divisor.

F1. [Remainder m/n.] Divide m by n and let r be the remainder.

F2. [Is it zero?] If $r = 0$, the algorithm terminates with answer n.

F3. [Remainder n/r.] Divide n by r and let m be the remainder.

F4. [Is it zero?] If $m = 0$, the algorithm terminates with answer r.

F5. [Remainder r/m.] Divide r by m and let n be the remainder.

F6. [Is it zero?] If $n = 0$, the algorithm terminates with answer m; otherwise go back to step F1. ∎

4. By algorithm E, n = 6099,2166,1767,399,171,57. Answer = 57.

5. Not finite nor definite nor effective, perhaps no output; in format, no letter is given before step numbers, no summary phrase appears, and there is no "∎".

6. We try Algorithm E with $n = 5$ and count the number of times step E1 is executed. For $m = 1, 2, 3, 4, 5$, respectively, we get 2, 3, 4, 3, 1 times; the average is $2.6 = T_5$.

7. In all but a finite number of cases, $n > m$. In this case, the first iteration of Algorithm E merely exchanges these numbers; so $U_m = T_m + 1$.

8. Let $A = \{a, b, c\}$, $N = 5$. The algorithm will terminate with the string $a^{\gcd(m,n)}$.

j	θ_j	ϕ_j	b_j	a_j	
0	ab	(empty)	1	2	Remove one a and one b, or go to 2.
1	(empty)	c	0	0	Add c at extreme left, go back to 0.
2	a	b	2	3	Change all a's to b's.
3	c	a	3	4	Change all c's to a's.
4	b	b	0	5	If b's remain, repeat.

9. For example we can say C_2 represents C_1 if there is a function g from I_1 into I_2, and function h from Q_2 into Q_1 taking Ω_2 into Ω_1, and a function j from Q_2 into the positive integers, satisfying the following conditions:

a) If x is in I_1, C_1 produces the output y from x if and only if there exists a y' in Ω_2 for which C_2 produces the output y' from $g(x)$ and $h(y') = y$.
b) If q is in Q_2 then $f_1(h(q)) = h(f_2^{j(q)}(q))$, where $f_2^{j(q)}$ means the function f_2 is to be iterated $j(q)$ times.

For example, let C_1 be as in (2) and let C_2 have $I_2 = \{(m, n)\}$, $\Omega_2 = \{(m, n, d)\}$, $Q_2 = I_2 \cup \Omega_2 \cup \{(m, n, a, b, 1)\} \cup \{(m, n, a, b, r, 2)\} \cup \{(m, n, a, b, r, 3)\} \cup \{(m, n, a, b, r, 4)\}$. Let $f_2(m, n) = (m, n, m, n, 1)$; $f_2(m, n, d) = (m, n, d)$; $f_2(m, n, a, b, 1) = (m, n, a, b, a \bmod b, 2)$; $f_2(m, n, a, b, r, 2) = (m, n, b)$ if $r = 0$, otherwise $(m, n, a, b, r, 3)$; $f_2(m, n, a, b, r, 3) = (m, n, b, b, r, 4)$; $f_2(m, n, a, b, r, 4) = (m, n, a, r, 1)$.

Now let $h(m, n) = (m, n) = g(m, n)$; $h(m, n, d) = (d)$; $h(m, n, a, b, 1) = (a, b, 0, 1)$ if $a = m$, $b = n$, otherwise $(a, b, b, 1)$; $h(m, n, a, b, r, 2) = (a, b, r, 2)$; $h(m, n, a, b, r, 3) = (a, b, r, 3)$; $h(m, n, a, b, r, 4) = h(f_2(m, n, a, b, r, 4))$; $j(m, n, a, b, r, 3) = j(m, n, a, b, r, 4) = 2$, otherwise $j(q) = 1$. Then C_2 represents C_1.

Notes: It is tempting to try to define things in a more simple way, e.g. to let g map Q_1 into Q_2 and to insist that when $x_0, x_1, \ldots$ is a computational sequence in C_1 then $g(x_0), g(x_1), \ldots$ is a subsequence of the computational sequence in C_2 that begins with $g(x_0)$. But this is inadequate, e.g. in the above example C_1 forgets the original values of m and n but C_2 does not.

If C_3 represents C_2 by means of functions g, h, j, and if C_2 represents C_1 by means of functions g', h', j', then C_3 represents C_1 by means of functions g'', h'', j'', where

$$g''(x) = g'(g(x)), \qquad h''(x) = h(h'(x)),$$

and

$$j''(q) = \sum_{0 \le k < j(h'(q))} j'(q_k),$$

if $q_0 = q$, $q_{k+1} = f_3^{j'(q_k)}(q_k)$. Hence the above relation is transitive. We can say C_2 *directly* represents C_1 if the function j is bounded; this relation is also transitive. The relation "C_2 represents C_1" generates an equivalence relation in which two computational methods apparently are equivalent if and only if they compute isomorphic functions of their inputs; the relation "C_2 directly represents C_1" generates a more interesting equivalence relation which perhaps matches the intuitive idea of being "essentially the same algorithm."

SECTION 1.2.1

1. (a) Prove $P(0)$. (b) Prove that $P(0), \ldots, P(n)$ implies $P(n+1)$, for all $n \geq 0$.

2. The theorem has not been proved for $n = 2$; in the second part of the proof, take $n = 1$; we assume there that $a^{-1} = 1$. If this condition is true (i.e. if $a = 1$) the theorem is indeed valid.

3. The correct answer is $1 - 1/n$. The mistake occurs in the proof for $n = 1$, when the formula on the left either may be assumed to be meaningless, or it may be assumed to be zero (since there are $n - 1$ terms).

5. If n is prime, it is trivially a product of primes. Otherwise by definition, n has factors, so $n = km$ for $1 < k, m < n$. Since both k and m are less than n, by induction they can be written as products of primes; hence n is the product of the primes appearing in the representations of k and m.

6. In the notation of Fig. 3, we prove *A5* implies *A6*. This is clear since *A5* implies $(a' - qa)m + (b' - qb)n = (a'm + b'n) - q(am + bn) = c - qd = r$.

7. Solution is $1 + 2 + \cdots + n$; or, $n(n+1)/2$.

8. (a) We will show $(n^2 - n + 1) + (n^2 - n + 3) + \cdots + (n^2 + n - 1)$ equals n^3. The sum is $\big(1 + 3 + \cdots + (n^2 + n - 1)\big) - \big(1 + 3 + \cdots + (n^2 - n - 1)\big) = \big((n^2 + n)/2\big)^2 - \big((n^2 - n)/2\big)^2 = n^3$. We have used Eq. (2); however, an inductive proof was requested, so another approach should be taken! For $n = 1$, the result is obvious. Let $n \geq 1$; $(n+1)^2 - (n+1) = n^2 - n + 2n$, so the first terms for $n+1$ are $2n$ larger; thus the sum for $n+1$ is the sum for n plus $2n + \cdots + 2n$ [n times] + $(n+1)^2 + (n+1) - 1$; this equals $n^3 + 2n^2 + n^2 + 3n + 1 = (n+1)^3$. (b) We have shown the first term for $(n+1)^3$ is two greater than the last term for n^3. Therefore by Eq. (2), $1^3 + 2^3 + \cdots + n^3 =$ sum of consecutive odd numbers starting with $1 =$ (number of terms)$^2 = (1 + 2 + \cdots + n)^2$.

10. Obvious for $n = 10$. If $n \geq 10$, we have $2^{n+1} = 2 \cdot 2^n > (1 + \frac{1}{10})^3 2^n$ and by induction this is greater than $(1 + 1/n)^3 n^3 = (n+1)^3$.

11. $(-1)^n(n+1)/(4(n+1)^2 + 1)$.

12. The only nontrivial part of the extension is the calculation of the integer q in E2. This can be done by repeated subtraction, reducing to the problem of determining whether $u + v\sqrt{2}$ is positive, negative, or zero, and the latter problem is readily solved.

It is easy to show that whenever $u + v\sqrt{2} = u' + v'\sqrt{2}$, we must have $u = u'$ and $v = v'$, since $\sqrt{2}$ is irrational. Now it is clear that 1 and $\sqrt{2}$ have no common divisor, if we define divisor in the sense that $u + v\sqrt{2}$ divides $a(u + v\sqrt{2})$ if and only if a is an integer.

(*Note:* However, if we extend the concept of divisor so that $u + v\sqrt{2}$ is said to divide $a(u + v\sqrt{2})$ if and only if a has the form $u' + v'\sqrt{2}$ for integers u' and v', there *is* a way to extend Algorithm E so that it always will terminate: If in step E2 we have $c = u + v\sqrt{2}$ and $d = u' + v'\sqrt{2}$, compute $c/d = c(u' - v'\sqrt{2})/(u'^2 - 2v'^2) = x + y\sqrt{2}$ where x and y are rational. Now let $q = u'' + v''\sqrt{2}$, where u'' and v'' are the nearest integers to x and y; and let $r = c - qd$. If $r = u''' + v'''\sqrt{2}$, it follows that $|u'''^2 - 2v'''^2| < |u'^2 - 2v'^2|$, hence the computation will terminate. For further information, see "quadratic Euclidean domains" in number theory textbooks.)

13. Add "$n = n_0$" to *A1* and *A2*; "$T \leq 3(n_0 - d) + k$" to the others, where $k = 2, 3, 3, 1$, respectively for *A3*, *A4*, *A5*, *A6*. Also add "$d > 0$" to *A4*.

15. (a) Let $A = S$ in (iii); this implies every nonempty well-ordered set has a "least" element.

(b) Let $x \prec y$ if $|x| < |y|$ or if $|x| = |y|$ and $x < 0 < y$.

(c) No, the subset of all positive reals fails to satisfy (iii). (*Note:* Using the "axiom of choice," a rather complicated argument can be given to show that every set can be well-ordered somehow; but nobody has yet been able to define an explicit relation which well-orders the real numbers.)

(d) To prove (iii) for T_n, use induction on n: Let A be a nonempty subset of T_n and consider A_1, the set of first components of A. Since A_1 is a nonempty subset of S, and S is well-ordered, A_1 contains a smallest element x. Now consider A_x, the subset of A in which the first component equals x; A_x may be considered a subset of T_{n-1} if its first component is suppressed, so by induction A_x contains a smallest element $(x, x_2, \ldots, x_n)$ which in fact is the smallest element of A.

(e) No, although properties (i) and (ii) are valid. If S contains at least two distinct elements a and b, the set (b), (a, b), (a, a, b), (a, a, a, b), (a, a, a, a, b), ... has no least element.

T can be well-ordered if we define $(x_1, \ldots, x_n) \prec (y_1, \ldots, y_m)$ whenever $n < m$, or $n = m$ and $(x_1, \ldots, x_n) \prec (y_1, \ldots, y_m)$ in T_n.

(f) Let S be well-ordered by $\prec$. If such an infinite sequence exists, the set A consisting of the members of the sequence fails to satisfy property (iii), for no element of the sequence can be smallest. Conversely if $\prec$ is a relation satisfying (i) and (ii) but not (iii), let A be a non-empty subset of S which has no smallest element. Since A is not empty, we can find x_1 in A; since x_1 is not the smallest element of A, there is x_2 in A for which $x_2 \prec x_1$; since x_2 is not the smallest element either, we can find $x_3 \prec x_2$; etc.

(g) Let A be the set of all x for which $P(x)$ is false. If A is not empty, it contains a smallest element, x_0. Hence $P(y)$ is true for all $y \prec x_0$. But this implies $P(x_0)$ is true, so x_0 is not in A (a contradiction). Therefore A must be empty, i.e. $P(x)$ is always true.

SECTION 1.2.2

1. There is none; if r is a positive rational, $r/2$ is smaller.

2. Not if infinitely many nines appear; in that case the decimal expansion of the number is $1 + .24000000\ldots$, according to Eq. (2).

3. $\frac{1}{343}$.

4. 4.

6. The decimal expansion of a number is unique, so $x = y$ if and only if $m = n$, and $d_i = e_i$ for $i = 1, 2, \ldots$. If $x \neq y$, one may compare m vs. n, d_1 vs. e_1, d_2 vs. e_2, etc., and when the first inequality occurs the larger one belongs to the larger of x, y.

7. One may use induction on x, first proving the laws for x positive, and then for x negative. Details are omitted here.

8. By trying $n = 0, 1, 2, \ldots$ we find the value of n for which $n^m \leq u < (n+1)^m$. Then assuming inductively $n, d_1, \ldots, d_{k-1}$ have been determined, d_k is the digit

such that

$$\left(n+\frac{d_1}{10}+\cdots+\frac{d_k}{10^k}\right)^m \le u < \left(n+\frac{d_1}{10}+\cdots+\frac{d_k}{10^k}+\frac{1}{10^k}\right)^m.$$

9. $((b^{p/q})^{u/v})^{qv} = (((b^{p/q})^{u/v})^v)^q = ((b^{p/q})^u)^q = ((b^{p/q})^q)^u = b^{pu}$, hence $(b^{p/q})^{u/v} = b^{pu/qv}$. This proves the second law. We prove the first law using the second: $b^{p/q}b^{u/v} = (b^{1/qv})^{pv}(b^{1/qv})^{qu} = (b^{1/qv})^{pv+qu} = b^{p/q+u/v}$.

10. If $\log_{10} 2 = p/q$, with p and q positive, then $2^q = 10^p$, which is absurd since the righthand side is divisible by 5 but the lefthand side isn't.

11. Infinitely many! No matter how many digits of x are given, we will not know whether $10^x = 1.99999\ldots$ or $2.00000\ldots$. There is nothing mysterious or paradoxical in this; a similar situation occurs in addition, if we are adding $.444444\ldots$ to $.555555\ldots$

12. They are the only values of $d_1, \ldots, d_8$ which satisfy Eq. (6).

13. (a) First prove by induction that if $y > 0$, $1 + ny \le (1+y)^n$. Then set $y = x/n$, and take n-th roots. (b) $x = b - 1$, $n = 10^k$.

14. Set $x = \log_b c$ in the second equation of (4) and take logarithms of both sides of this equation.

15. Prove it, by transposing "$\log_b y$" to the other side of the equation and using (10).

16. $\ln x/\ln 10$.

17. 5; 1; 1; 0; undefined.

18. No, $\log_8 x = \log_2 x/\log_2 8 = \frac{1}{3}\log_2 x$.

19. Yes, since $\log_2 x < (\log_{10} x)/.301 < 14/.301 < 47$.

20. They are reciprocals.

21. $(\ln\ln x - \ln\ln b)/\ln b$.

22. From the tables appearing in Appendix B, $\log_2 x = 1.442695 \ln x$; $\log_{10} x = .4342945 \ln x$. The error is $(1.442695 - 1.4342945)/1.442695 = 0.582\%$.

23. Take the figure of area $\ln y$, and divide its height by x while multiplying its length by x. This deformation preserves its area and makes it congruent to the piece left when $\ln x$ is removed from $\ln xy$: for the height at point $x + xt$ in the diagram for $\ln xy$ is $1/(x + xt) = \big(1/(1+t)\big)/x$.

24. Substitute 2 everywhere 10 appears.

27. Prove by induction on k that

$$x^{2^k}(1-\eta)^{2^{k+1}-1} \le 10^{2^k(n+b_1/2+\cdots+b_k/2^k)}x_k' \le x^{2^k}(1+\epsilon)^{2^{k+1}-1}$$

and take logarithms.

28. **E1.** Set $y \leftarrow 1$, $k \leftarrow 0$.

E2. If $x = 0$, stop.

E3. If $x < \log_b(1 + 2^{-k})$, go to E5.

E4. Set $x \leftarrow x - \log_b(1 + 2^{-k})$, $y \leftarrow y + 2^{-k}y$, go to E2.

E5. Increase k by 1, go to E2. ▮

The computational error arises when we set $x \leftarrow x - \log_b(1 + 2^{-k}) + \eta_j$ and $y \leftarrow y(1 + 2^{-k})(1 + \epsilon_j)$ at the j-th execution of step E4, for certain small errors η_j and ϵ_j. When the algorithm terminates, we have $y = b^x \Pi(1 + \epsilon_j) b^{\Sigma \eta_j}$. Further analysis depends on b and the computer word size. Notice that in both this case and in Ex. 26, it is possible to refine the error estimates somewhat if the base is e, since for most values of k the table entry $\ln(1 \pm 2^{-k})$ can be given with high accuracy: it equals $\pm 2^{-k} - \frac{1}{2} 2^{-2k} \pm \frac{1}{3} 2^{-3k} - \cdots$.

Note: Similar algorithms can be given for trigonometric functions; see J. E. Meggitt, *IBM J. Res. and Dev.* **6** (1962), 210–226, **7** (1963), 237–245.

29. e; 3; 4.

SECTION 1.2.3

1. $a_1 + a_2 + a_3$.

2. $\frac{1}{1} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{9} + \frac{1}{11}$; $\frac{1}{9} + \frac{1}{3} + \frac{1}{1} + \frac{1}{3} + \frac{1}{9}$.

3. The rule for $p(j)$ is violated; in the first place, the value 3 is assumed for no n^2, and in the second place the value 4 is assumed for *two* n^2.

4. $(a_{11}) + (a_{21} + a_{22}) + (a_{31} + a_{32} + a_{33})$
$= (a_{11} + a_{21} + a_{31}) + (a_{22} + a_{32}) + (a_{33})$.

5. It is only necessary to use the rule $a\sum_{R(i)} x_i = \sum_{R(i)} (ax_i)$:

$$\left(\sum_{R(i)} a_i\right)\left(\sum_{S(j)} b_j\right) = \sum_{R(i)} a_i \left(\sum_{S(j)} b_j\right) = \sum_{R(i)} \left(\sum_{S(j)} a_i b_j\right).$$

7. Use Eq. (3); the two limits are interchanged and the terms between a_0 and a_c must be transferred from one limit to the other.

8. Let $a_{(i+1)i} = +1$, and $a_{i(i+1)} = -1$, for all $i \geq 0$, and all other a_{ij} zero; let $R(i) = S(i) =$ "$i \geq 0$". The lefthand side is -1, the righthand side is $+1$.

10. No, the two applications of rule (d) assume $n \geq -1$.

11. $n + 1$.

12. $\frac{7}{6}(1 - 1/7^{n+1})$.

13. $m(n - m + 1) + \frac{1}{2}(n - m)(n - m + 1)$; or, $\frac{1}{2}\big(n(n + 1) - m(m - 1)\big)$.

14. $\big(m(n-m+1)+\frac{1}{2}(n-m)(n-m+1)\big)\big(r(s-r+1)+\frac{1}{2}(s-r)(s-r+1)\big)$.

15, 16. Key steps:

$$\sum_{0\leq j\leq n} jx^j = x \sum_{1\leq j\leq n} jx^{j-1} = x \sum_{0\leq j\leq n-1} (j+1)x^j$$

$$= x \sum_{0\leq j\leq n} jx^j - nx^{n+1} + x \sum_{0\leq j\leq n-1} x^j.$$

17. The number of elements in S.

18. $S'(j) =$ "$1 \leq j < n$". $R'(i, j) =$ "n is a multiple of i and $i > j$".

19. $a_n - a_{m-1}$.

20. $(b-1)\sum_{0\le k\le n}(n-k)b^k + n + 1 = \sum_{0\le k\le n} b^k$; this formula follows from (14) and the result of Ex. 16.

21. Analogous to (3), plus the stipulation that there exists an integer j_0 such that

$$\prod_{\substack{R(j)\\ |j|>j_0}} a_j \neq 0.$$

22. For (5), (7) just change $\sum$ to $\prod$. Also, we have

$$\prod_{R(i)}(b_ic_i) = \Big(\prod_{R(i)} b_i\Big)\Big(\prod_{R(i)} c_i\Big);\quad \Big(\prod_{R(j)} a_j\Big)\Big(\prod_{S(j)} a_j\Big) = \Big(\prod_{\substack{R(j)\\ \text{or } S(j)}} a_j\Big)\Big(\prod_{\substack{R(j)\\ \text{and } S(j)}} a_j\Big).$$

23. $0 + x = x$ and $1 \cdot x = x$. This makes many operations and equations simpler, e.g. rule (d) and its analogue in the previous exercise.

25. First step and last step o.k. Second step, uses i for two different purposes at once. Third step, should probably be $\sum_{1\le i\le n} n$.

26. Key steps:

$$\prod_{0\le i\le n}\Big(\prod_{0\le j\le n} a_ia_j\Big) = \prod_{0\le i\le n}\Big(a_i^{n+1}\prod_{0\le j\le n} a_j\Big) = \Big(\prod_{0\le i\le n} a_i^{n+1}\Big)\Big(\prod_{0\le i\le n}\Big(\prod_{0\le j\le n} a_j\Big)\Big)$$
$$= \Big(\prod_{0\le i\le n} a_i\Big)^{2n+2}.$$

The answer is

$$\Big(\prod_{0\le i\le n} a_i\Big)^{n+2}.$$

28. $(n+1)/2n$.

29. a) $\sum_{0\le k\le j\le i\le n} a_ia_ja_k$.

b) Let $S_r = \sum_{0\le i\le n} a_i^r$. Solution: $\frac{1}{3}S_3 + \frac{1}{2}S_1S_2 + \frac{1}{6}S_1^3$. The general solution to this problem, as the number of indices gets larger, may be found in Section 1.2.9, Eq. (34).

31. $n\sum_{1\le j\le n} a_jb_j - \Big(\sum_{1\le j\le n} a_j\Big)\Big(\sum_{1\le j\le n} b_j\Big)$.

33. This can be proved by induction on n, if we rewrite the formula as

$$\frac{1}{x_n - x_{n-1}}\left(\sum_{1\le j\le n}\frac{x_j^r(x_j - x_{n-1})}{\prod_{1\le k\le n,\ k\neq j}(x_j - x_k)} - \sum_{1\le j\le n}\frac{x_j^r(x_j - x_n)}{\prod_{1\le k\le n,\ k\neq j}(x_j - x_k)}\right).$$

Each of these sums now has the form of the original sum, except on $n-1$ elements, and the values turn out nicely by induction when $0 \le r \le n-1$. When $r = n$, consider the identity

$$0 = \sum_{1\le j\le n}\frac{\prod_{1\le k\le n}(x_j - x_k)}{\prod_{1\le k\le n,\ k\neq j}(x_j - x_k)} = \sum_{1\le j\le n}\frac{x_j^n - (x_1 + \cdots + x_n)x_j^{n-1} + P(x_j)}{\prod_{1\le k\le n,\ k\neq j}(x_j - x_k)}$$

where $P(x_j)$ is a polynomial of degree $n-2$; from the solution for $r = 0, 1, \ldots, n-1$ we obtain the desired answer.

Note: The formulas here are the basis for numerical methods concerning "divided differences." The following alternate method of proof, using complex variable theory, is less elementary but more elegant: By the residue theorem, the value of the given sum is

$$\frac{1}{2\pi i}\int_{|z|=R}\frac{z^r\,dz}{(z-x_1)\cdots(z-x_n)}$$

where $R > |x_1|, \ldots, |x_n|$. The Laurent expansion of the integrand converges uniformly on $|z| = R$; it is

$$z^{r-n}\left(\frac{1}{1-x_1/z}\right)\cdots\left(\frac{1}{1-x_n/z}\right)$$
$$= z^{r-n} + (x_1 + \cdots + x_n)z^{r-n-1} + (x_1^2 + x_1x_2 + \cdots)z^{r-n-2} + \cdots.$$

Integrating term by term, everything vanishes except the coefficient of z^{-1}. This method gives us the *general formula* for an arbitrary integer $r \geq 0$:

$$\sum_{\substack{j_1+\cdots+j_n=r-n+1\\ j_1,\cdots,j_n\geq 0}} x_1^{j_1}\ldots x_n^{j_n}.$$

34. If the reader has tried earnestly to solve this problem, *without* getting the answer, perhaps its purpose has been achieved. The temptation to regard the numerators as polynomials in x rather than as polynomials in k is almost overwhelming. It would undoubtedly be easier to prove the considerably more general result

$$\sum_{1\leq k\leq n}\frac{\prod_{1\leq r\leq n-1}(y_k - z_r)}{\prod_{1\leq r\leq n,\ r\neq k}(y_k - y_r)} = 1,$$

which is an identity in $2n - 1$ variables!

35. If $R(j)$ never holds, the value should be $-\infty$. The stated analogue of rule (a) is based on the identity $a + \max(b, c) = \max(a + b, a + c)$. Similarly if all a_i, b_j are *nonnegative*, we have

$$\sup_{R(i)} a_i \sup_{S(j)} b_j = \sup_{R(i)}\sup_{S(j)} a_ib_j.$$

Rules (b), (c) do not change; for rule (d) we get the simpler form

$$\sup\left(\sup_{R(j)} a_j, \sup_{S(j)} a_j\right) = \sup_{R(j)\text{ or }S(j)} a_j.$$

36. Subtract column one from columns $2, \ldots, n$. Add rows $2, \ldots, n$ to row one. The result is a triangular determinant.

37. Subtract column one from columns $2, \ldots, n$. Then subtract x_1 times row $k - 1$ from row k, for $k = n, n - 1, \ldots, 2$ (in that order). We now factor x_1 out of the first column and factor $x_k - x_1$ out of columns $k = 2, \ldots, n$, obtaining $x_1(x_2 - x_1)\ldots(x_n - x_1)$ times a Vandermonde determinant of order $n - 1$, so the process continues by induction.

Alternate proof, using "higher" mathematics: The determinant is a polynomial in the variables $x_1, \ldots, x_n$ of total degree $1 + 2 + \cdots + n$. It vanishes if $x_j = 0$ or if $x_i = x_j$ $(i < j)$, and the coefficient of $x_1^1x_2^2\ldots x_n^n$ is $+1$. These facts characterize its

value. In general, if two rows of a matrix become equal for $x_i = x_j$, their difference is usually divisible by $x_i - x_j$, and this observation often speeds the evaluation of determinants. (R. W. Floyd.)

38. Subtract column one from columns 2, ..., n, and factor out $(x_1 + y_1)^{-1} \ldots (x_n + y_1)^{-1}(y_1 - y_2) \ldots (y_1 - y_n)$ from rows and columns. Now subtract row one from rows 2, ..., n and factor out $(x_1 - x_2) \ldots (x_1 - x_n)(x_1 + y_2)^{-1} \ldots (x_1 + y_n)^{-1}$; we are left with the Cauchy determinant of order $n - 1$.

39. Let I = identity matrix, J = matrix of all ones. Since $J^2 = nJ$, we find immediately $(xI + yJ)\big((x + ny)I - yJ\big) = x(x + ny)I$.

40. $$\sum_{1\le t\le n} b_{it}x_j^t = x_j \prod_{\substack{1\le k\le n\\ k\ne i}} (x_k - x_j) \Big/ x_i \prod_{\substack{1\le k\le n\\ k\ne i}} (x_k - x_i) = \delta_{ij}.$$

41. This follows immediately from the observations about the relation of an inverse matrix to cofactors. It may also be interesting to give a direct proof here.

$$\sum_{1\le t\le n} \frac{1}{x_i + y_t} b_{tj} = \sum_{1\le t\le n} \frac{\prod_{k\ne t}(x_j + y_k - x)\prod_{k\ne i}(x_k + y_t)}{\prod_{k\ne j}(x_j - x_k)\prod_{k\ne t}(y_t - y_k)}$$

when $x = 0$. This is a polynomial of degree at most $n - 1$ in x. If we set $x = x_j + y_s$, $1 \le s \le n$, the terms are zero except when $s = t$, so the value of this polynomial is

$$\prod_{k\ne i}(-x_k - y_s) \Big/ \prod_{k\ne j}(x_j - x_k) = \prod_{k\ne i}(x_j - x_k - x) \Big/ \prod_{k\ne j}(x_j - x_k).$$

Since these polynomials of degree at most $n - 1$ agree at n distinct points x, they agree also for $x = 0$, hence

$$\sum_{1\le t\le n} \frac{1}{x_i + y_t} b_{tj} = \prod_{k\ne i}(x_j - x_k) \Big/ \prod_{k\ne j}(x_j - x_k) = \delta_{ij}.$$

42. $n/(x + ny)$.

43. $1 - \prod_{1\le k\le n}(1 - 1/x_k)$. This is easily verified if any $x_i = 1$, since the inverse of any matrix having a row or column all of ones must have elements whose sum is 1. If none of the x_i equals one, sum the elements of row i as in Ex. 44 and obtain $\prod_{k\ne i}(x_k - 1)/x_i \prod_{k\ne i}(x_k - x_i)$. We can now sum this on i using Ex. 33, with $r = 0$ (multiply numerator and denominator by $(x_i - 1)$).

44. We find

$$c_j = \sum_{1\le i\le n} b_{ij} = \prod_{1\le k\le n}(x_k + y_i) \Big/ \prod_{\substack{1\le k\le n\\ k\ne j}} (x_j - x_k),$$

after applying Ex. 33. And

$$\sum_{1\le j\le n} c_j = \sum_{1\le j\le n} \frac{(x_j^n + (y_1 + \cdots + y_n)x_j^{n-1} + \cdots)}{\prod_{1\le k\le n,\ k\ne j}(x_j - x_k)}$$

$$= (x_1 + x_2 + \cdots + x_n) + (y_1 + y_2 + \cdots + y_n).$$

45. Let $x_i = i$, $y_j = j - 1$. From Ex. 44, the sum of the elements of the inverse is $(1 + 2 + \cdots + n) + \big((n - 1) + (n - 2) + \cdots + 0\big) = n^2$. From Ex. 38, the

elements of the inverse are

$$b_{ij} = \frac{(-1)^{i+j}(i+n-1)!(j+n-1)!}{(i+j-1)(i-1)!^2(j-1)!^2(n-i)!(n-j)!}.$$

This quantity can be put into several forms involving binomial coefficients, for example

$$\frac{(-1)^{i+j}ij}{i+j-1}\binom{-i}{n}\binom{n}{i}\binom{-j}{n}\binom{n}{j}$$

$$= (-1)^{i+j}j\binom{i+j-2}{i-1}\binom{i+n-1}{i-1}\binom{j+n-1}{n-i}\binom{n}{j}.$$

From the latter formula we see that b_{ij} is not only an integer, it is divisible by i, j, n, $i+j-1$, $i+n-1$, and $j+n-1$. Perhaps the prettiest formula for b_{ij} is

$$(i+j-1)\binom{i+j-2}{i-1}^2\binom{-(i+j)}{n-i}\binom{-(i+j)}{n-j}.$$

The solution to this problem would be extremely difficult if we had not realized that a Hilbert matrix is a special case of a Cauchy matrix; the more general problem is much easier to solve than its special case! It is frequently wise to generalize a problem to its "inductive closure", i.e. to the smallest generalization such that all subproblems that arise in an attempted proof by mathematical induction belong to the same class. In this case, we see that cofactors of a Cauchy matrix are Cauchy matrices, but cofactors of Hilbert matrices are not Hilbert matrices. [For further information, see J. Todd, *J. Res. Nat. Bur. Stand.* **65** (1961), 19–22.]

46. For any integers $k_1, k_2, \ldots, k_m$, let $\epsilon(k_1, \ldots, k_m) = \text{sign}\,(\prod_{1\le i<j\le m}(k_j - k_i))$. If $(q_1, \ldots, q_m)$ is equal to $(k_1, \ldots, k_m)$ except for the fact that k_i and k_j have been interchanged, we have $\epsilon(q_1, \ldots, q_m) = -\epsilon(k_1, \ldots, k_m)$. Therefore we have the equation $\det(B_{k_1\ldots k_m}) = \epsilon(k_1, \ldots, k_m)\det(B_{j_1\ldots j_m})$, if $j_1 \le \cdots \le j_m$ are the numbers $k_1, \ldots, k_m$ rearranged into non-decreasing order. Now by definition of the determinant,

$$\det(AB) = \sum_{1\le q_1,\ldots,q_m\le m} \epsilon(q_1, \ldots, q_m)\Big(\sum_{1\le k\le n} a_{1k}b_{kq_1}\Big)\cdots\Big(\sum_{1\le k\le n} a_{mk}b_{kq_m}\Big)$$

$$= \sum_{1\le k_1,\ldots,k_m\le n} a_{1k_1}\ldots a_{mk_m} \sum_{1\le q_1,\ldots,q_m\le m} \epsilon(q_1, \ldots, q_m)b_{k_1q_1}\ldots b_{k_mq_m}$$

$$= \sum_{1\le k_1,\ldots,k_m\le n} a_{1k_1}\ldots a_{mk_m}\det(B_{k_1\ldots k_m})$$

$$= \sum_{1\le k_1,\ldots,k_m\le n} \epsilon(k_1, \ldots, k_m)a_{1k_1}\ldots a_{mk_m}\det(B_{j_1\ldots j_m})$$

[where the j's are related to the k's as above]

$$= \sum_{1\le j_1\le\ldots\le j_m\le n} \det(A_{j_1\ldots j_m})\det(B_{j_1\ldots j_m}).$$

Finally, if two j's are equal, $\det(A_{j_1\ldots j_m}) = 0$.

SECTION 1.2.4

1. 1, −2, −1, 0, 5.

2. $\lfloor x \rfloor$.

3. $\lfloor x \rfloor$ is the greatest integer less than or equal to x, by definition; therefore, $\lfloor x \rfloor$ is an integer, $\lfloor x \rfloor \leq x$, and $\lfloor x \rfloor + 1 > x$. The latter properties, plus the fact that when m, n are integers $n < m$ if and only if $n \leq m - 1$, lead to an easy proof of propositions (a) and (b). Similar arguments prove (c) and (d). Finally, (e) and (f) are just combinations of previous parts of this exercise.

4. $x - 1 < \lfloor x \rfloor \leq x$; so $-x + 1 > -\lfloor x \rfloor \geq -x$; hence the result.

5. $\lfloor x + \frac{1}{2} \rfloor$. The value of ($-x$ rounded) will be the same as $-$(x rounded), *except* when $x \bmod 1 = \frac{1}{2}$, when the negative value is rounded towards zero and the positive value is rounded away from zero.

6. (a) is true: $\lfloor \sqrt{x} \rfloor = n$ iff $n^2 \leq x < (n+1)^2$ iff $n^2 \leq \lfloor x \rfloor < (n+1)^2$ iff $\lfloor \sqrt{\lfloor x \rfloor} \rfloor = n$. Similarly, (b) is true. But (c) fails, e.g. for $x = 1.1$.

7. Apply Ex. 3 and Eq. (4). The inequality should be $\geq$ for ceilings, and then equality holds if and only if either x or y is an integer or $x \bmod 1 + y \bmod 1 \geq 1$.

8. 1, 2, 5, −100.

9. −1, 0, −2.

10. 0.1, 0.01, −0.09.

11. $x = y$.

12. All.

13. +1, −1.

14. 8.

15. Multiply both sides of Eq. (1) by z; if $y = 0$, the result is also easily verified.

17. As an example, consider the multiplication portion of law A: we have $a = b + qm$, $x = y + rm$ for some integers q, r; so $ax = by + (br + yq + qrm)m$.

18. We have $a - b = kr$ for some integer k, and also $kr \equiv 0$ (modulo s). Hence by law B, $k \equiv 0$ (modulo s), so $a - b = qsr$ for some integer q.

20. Multiply both sides of the congruence by a'.

21. There is at least one such representation, by the previously proved exercise. If there are two representations, $n = p_1 \ldots p_k = q_1 \ldots q_m$, we have $q_1 \ldots q_m \equiv 0$ (modulo p_1); so if none of the q's equals p_1 we could cancel them all by law B and obtain $1 \equiv 0$ (modulo p_1). The latter is impossible since p_1 is not equal to 1 (this is the principal reason we do not allow 1 as a prime number). So some q_j equals p_1, and $n/p_1 = p_2 \ldots p_k = q_1 \ldots q_{j-1} q_{j+1} \ldots q_m$. Either n is prime, when the result is clearly true, or by induction the two factorizations of n/p_1 are the same.

22. If $a = cd$, $m = nd$, then $an \equiv 0$ but $n \not\equiv 0$ (modulo m) if $d > 1$, $m \neq 0$.

24. Law A is always valid for addition and subtraction; Law C is always valid.

26. If b is not a multiple of p, then $b^2 - 1$ is, so one of the factors must be.

27. A number is relatively prime to p^e if and only if it is not a multiple of p. So we count those which are not multiples of p and get $\varphi(p^e) = p^e - p^{e-1}$.

28. If a, b are relatively prime to m, so is $(ab \bmod m)$, since any prime dividing the latter and m must divide a or b also. Now simply let $x_1, \ldots, x_{\varphi(m)}$ be the numbers relatively prime to m, and observe that $ax_1 \bmod m, \ldots, ax_{\varphi(m)} \bmod m$ are the same numbers in some order, etc.

29. We prove (b): if r, s are relatively prime and if k^2 divides rs, then p^2 divides rs for some prime p, so p divides r (say) and cannot divide s; so p^2 divides r. We see that $f(rs) = 0$ iff $f(r) = 0$ or $f(s) = 0$.

30. Let r, s be relatively prime. The idea is to prove that the $\varphi(rs)$ numbers relatively prime to rs are precisely the $\varphi(r)\varphi(s)$ distinct numbers $(sx_i + ry_j) \bmod (rs)$ where $x_1, \ldots, x_{\varphi(r)}$ and $y_1, \ldots, y_{\varphi(s)}$ are the corresponding values for r and s.

We then find $\varphi(10^6) = \varphi(2^6)\varphi(5^6) = (2^6 - 2^5)(5^6 - 5^5) = 400000$; $\varphi(p_1^{e_1} \ldots p_r^{e_r}) = (p_1^{e_1} - p_1^{e_1-1}) \ldots (p_r^{e_r} - p_r^{e_r-1})$; $\varphi(n) = n\prod_{p\backslash n, p \text{ prime}} (1 - 1/p)$. Another proof is in exercise 1.3.3–30.

31. The divisors of rs may be uniquely written in the form cd where c divides r and d divides s. Similarly, if $f(n) \geq 0$, we find the function $\max_{d\backslash n} f(d)$ is multiplicative. Cf. Ex. 1.2.3–35.

33. Either $n + m$ or $n - m + 1$ is even, so one of the quantities inside the brackets at the left is an integer, so equality holds in Ex. 7.

34. b must be an integer ≥ 2. (Set $x = b$). The sufficiency is proved as in Ex. 6. The same condition is necessary and sufficient for $\lceil \log_b x \rceil = \lceil \log_b \lceil x \rceil \rceil$.

More generally we have the following pretty generalization due to R. McEliece: Let f be a continuous, strictly increasing function defined on an interval A, and assume x in A implies that both $\lfloor x \rfloor$ and $\lceil x \rceil$ are in A. Then the relation $\lfloor f(x) \rfloor = \lfloor f(\lfloor x \rfloor) \rfloor$ holds for all x in A iff the relation $\lceil f(x) \rceil = \lceil f(\lceil x \rceil) \rceil$ holds for all x in A iff we have the following condition: "$f(x)$ is an integer implies x is an integer." The condition is obviously necessary, for if $f(x)$ is an integer and it equals $\lfloor f(\lfloor x \rfloor) \rfloor$ or $\lceil f(\lceil x \rceil) \rceil$ then x must equal $\lfloor x \rfloor$ or $\lceil x \rceil$. Conversely if e.g. $\lfloor f(\lfloor x \rfloor) \rfloor < \lfloor f(x) \rfloor$ then by continuity there is some y with $\lfloor x \rfloor < y \leq x$ for which $f(y)$ is an integer, and y cannot be an integer.

35.
$$\frac{x+m}{n} - 1 = \frac{x+m}{n} - \frac{1}{n} - \frac{n-1}{n} < \frac{\lfloor x \rfloor + m}{n} - \frac{n-1}{n} \leq \left\lfloor \frac{\lfloor x \rfloor + m}{n} \right\rfloor \leq \frac{x+m}{n};$$

apply Ex. 3. Use of Ex. 4 gives a similar result for the ceiling function. Both identities follow as a special case of McEliece's theorem in Ex. 34.

36. Assume first $n = 2t$.

$$\sum_{1 \leq k \leq n} \lfloor k/2 \rfloor = \sum_{1 \leq k \leq n} \lfloor (n+1-k)/2 \rfloor,$$

hence

$$\sum_{1 \leq k \leq n} \lfloor k/2 \rfloor = \frac{1}{2}\left(\sum_{1 \leq k \leq n} (\lfloor k/2 \rfloor + \lfloor (n+1-k)/2 \rfloor) \right)$$

$$= \frac{1}{2} \sum_{1 \leq k \leq n} \lfloor (2t+1)/2 \rfloor = t^2 = n^2/4.$$

(Cf. Ex. 33.) And if $n = 2t + 1$, we have $t^2 + \lfloor n/2 \rfloor = t^2 + t = n^2/4 - \frac{1}{4}$. For the second sum we get, similarly, $\lceil n(n+2)/4 \rceil$.

37. $$\sum_{0 \le k < n} \frac{mk + x}{n} = \frac{m(n-1)}{2} + x.$$

Let $\{y\}$ denote y mod 1; we must subtract

$$S = \sum_{0 \le k < n} \left\{ \frac{mk + x}{n} \right\};$$

S consists of d copies of the same sum, since if $t = n/d$,

$$\left\{ \frac{mk + x}{n} \right\} = \left\{ \frac{m(k+t) + x}{n} \right\}.$$

Let $u = m/d$; then

$$\sum_{0 \le k < t} \left\{ \frac{mk + x}{n} \right\} = \sum_{0 \le k < t} \left\{ \frac{x}{n} + \frac{uk}{t} \right\},$$

and since t and u are relatively prime this sum may be rearranged as

$$\left\{ \frac{x \bmod d}{n} \right\} + \left\{ \frac{x \bmod d}{n} + \frac{1}{t} \right\} + \cdots + \left\{ \frac{x \bmod d}{n} + \frac{t-1}{t} \right\}.$$

Finally, since $(x \bmod d)/n < 1/t$, the brackets in this sum may be removed and we have

$$S = d\left(\frac{t(x \bmod d)}{n} + \frac{t-1}{2} \right).$$

Applying Ex. 4, we get

$$\sum_{0 \le k < n} \left\lceil \frac{mk + x}{n} \right\rceil = \frac{(m+1)(n-1)}{2} - \frac{d-1}{2} + x + (-x) \bmod d.$$

This formula would become symmetric in m and n if it were extended over the range $0 \le k \le n$. (The symmetry can be explained by drawing the graph of the summand as a function of k, then reflecting about the line $y = x$.)

38. Take $m = 1$ in the preceding exercise. The ceiling function does not behave so nicely.

39. Proof of (f): Consider the more general identity $\prod_{0 \le k < n} 2 \sin \pi(x + k/n) = 2 \sin \pi n x$, which can be demonstrated as follows: Since $2 \sin \theta = (e^{i\theta} - e^{-i\theta})/i = (1 - e^{-2i\theta}) e^{i\theta - i\pi/2}$, the identity is a consequence of the two formulas

$$\prod_{0 \le k < n} (1 - e^{-2\pi(x + k/n)}) = 1 - e^{-2\pi n x} \quad \text{and} \quad \prod_{0 \le k < n} e^{\pi(x - (1/2) + (k/n)} = e^{\pi(nx - 1/2))}.$$

The latter is true since the function $x - \frac{1}{2}$ is replicative; and the former is true because we may set $z = 1$ in the factorization of the polynomial $z^n - \alpha^n = (z - \alpha)(z - \omega\alpha) \ldots (z - \omega^{n-1}\alpha)$, $\omega = e^{-2\pi i/n}$.

The identity in (i) is easily verified for $0 \le x < 1/n$, and we also find the function

$$g(nx) - g(x) - \cdots - g\left(x + \frac{n-1}{n}\right)$$

is periodic with period $1/n$.

41. We want $a_n = k$ when

$$\frac{k(k-1)}{2} < n \le \frac{k(k+1)}{2}.$$

The equation $k(k+1)/2 = r$ can be solved for k (the positive root):

$$k = \frac{-1+\sqrt{1+8r}}{2}.$$

Hence

$$\frac{-1+\sqrt{1+8n}}{2} \le k < \frac{-1+\sqrt{1+8n}}{2} + 1;$$

the answer is, by Ex. 3,

$$\left\lceil \frac{-1+\sqrt{1+8n}}{2} \right\rceil.$$

Another correct formula, which is not *obviously* the same, is $\lfloor (1+\sqrt{8n-7})/2 \rfloor$.

42. (a) Cf. Ex. 1.2.7–10. (b) The given sum is $n\lfloor \log_b n \rfloor - S$,

$$S = \sum_{\substack{1 \le k < n \\ k+1 \text{ is power of } b}} k = \sum_{1 \le t \le \log_b n} (b^t - 1) = (b^{\lfloor \log_b n \rfloor + 1} - b)/(b-1) - \lfloor \log_b n \rfloor.$$

43. $\lfloor \sqrt{n} \rfloor \left(n - \dfrac{(2\lfloor \sqrt{n} \rfloor + 5)(\lfloor \sqrt{n} \rfloor - 1)}{6} \right)$.

45. $\lfloor mj/n \rfloor = r$ if and only if

$$\left\lceil \frac{rn}{m} \right\rceil \le j < \left\lceil \frac{(r+1)n}{m} \right\rceil,$$

and we find the given sum is therefore

$$\sum_{0 \le r < m} f(r) \left(\left\lceil \frac{(r+1)n}{m} \right\rceil - \left\lceil \frac{rn}{m} \right\rceil \right).$$

The stated result follows by rearranging the latter sum, grouping the terms with a particular value of $\lceil rn/m \rceil$. The second formula is immediate by the substitution

$$f(x) = \binom{x+1}{k}.$$

46. $\sum_{0 \le r < \alpha m} \lceil rn/m \rceil (f(r-1) - f(r)) + \lceil \alpha n \rceil f(\lceil \alpha m \rceil - 1)$; prove like Ex. 45.

47. (a) The numbers $2, 4, \ldots, p-1$ are the even residues (modulo p); since $2kq = p\lfloor 2kq/p \rfloor + (2kq) \bmod p$, the number $(-1)^{\lfloor 2kq/p \rfloor}((2kq) \bmod p)$ will be an even

residue or an even residue minus p, and each even residue clearly occurs just once. Hence $(-1)^\sigma q^{(p-1)/2} 2 \cdot 4 \ldots (p-1) \equiv 2 \cdot 4 \ldots (p-1)$. (b) Let $q = 2$. If $p = 4n+1$, $\sigma = n$; if $p = 4n+3$, $\sigma = n+1$. Hence $(\frac{2}{p}) = 1, -1, -1, 1$ according as $p \bmod 8 = 1, 3, 5, 7$, respectively. (c) For $k < p/4$,

$$\lfloor (p-1-2k)q/p \rfloor = q - \lceil (2k+1)/p \rceil = q - 1 - \lfloor (2k+1)/p \rfloor \equiv \lfloor (2k+1)/p \rfloor \text{ (modulo 2).}$$

Hence we may replace the last terms $\lfloor (p-1)q/p \rfloor$, $\lfloor (p-3)q/p \rfloor$, ... by $\lfloor q/p \rfloor$, $\lfloor 3q/p \rfloor$, etc. (d) $\sum_{0 \le k < p/2} \lfloor kq/p \rfloor + \sum_{0 \le r < q/2} \lceil rp/q \rceil = \lceil p/2 \rceil (\lceil q/2 \rceil - 1) = (p+1)(q-1)/4$. Also $\sum_{0 \le r < q/2} \lceil rp/q \rceil = \sum_{0 \le r < q/2} \lfloor rp/q \rfloor + (q-1)/2$. The idea of this proof goes back to G. Eisenstein, *Journal f. d. reine und angewandte Math.* **28** (1844), 246–248; Eisenstein also gives several other proofs of this and other reciprocity laws in the same volume.

48. (a) This is clearly not always true when $n < 0$; when $n > 0$ it is easy to verify. (b) $\lfloor (n + 2 - \lfloor n/25 \rfloor)/3 \rfloor = \lceil (n - \lfloor n/25 \rfloor)/3 \rceil = \lceil (n + \lceil -n/25 \rceil)/3 \rceil = \lceil \lceil 24n/25 \rceil /3 \rceil =$ (cf. Ex. 35) $\lceil 8n/25 \rceil = \lfloor (8n+24)/25 \rfloor$.

SECTION 1.2.5

1. 52! For the curious, this number is 806 58175 17094 38785 71660 63685 64037 66975 28950 54408 83277 82400 00000 00000. (!)

2. $p_{nk} = p_{n(k-1)}(n - k + 1)$. After the first $n - 1$ objects have been placed, there is only one left, only one choice for the last object. But this does *not* mean that the last object is always the same in all permutations!

3. 53124, 35124, 31524, 31254, 31245; 42351, 41352, 41253, 31254, 31245.

4. There are 2568 digits. The leading digit is 4 (since $\log_{10} 4 = 2 \log_{10} 2 \approx .602$). The least significant digit is zero, and in fact by Eq. (8) the low order 249 digits are all zero! The exact value of 1000! was calculated by H. S. Uhler using a desk calculator and much patience over a period of several years, and the value appears in *Scripta Mathematica* **21** (1955), pp. 266–267. It begins with 402 38726 00770 . . .

5. $(39902)(97/96) \approx 416 + 39902 = 40318$.

6. $2^{18} \cdot 3^8 \cdot 5^4 \cdot 7^2 \cdot 11 \cdot 13 \cdot 17 \cdot 19$.

8. It is $m^n m!/\big((n+m)!/n!\big) = n!$ times $m^n/(m+1) \ldots (m+n)$. The latter quantity approaches one since $m/(m+k) \to 1$.

9. $\sqrt{\pi}$ and $-2\sqrt{\pi}$. (Ex. 10 used.)

10. Yes, except when $x = 0$ or a negative integer. For we have

$$(x+1) = x \lim_{m\to\infty} \frac{m^x m!}{x(x+1)\cdots(x+m)} \left(\frac{m}{x+m+1}\right).$$

11, 12.
$$\begin{aligned}\mu &= (a_k p^{k-1} + \cdots + a_1) + (a_k p^{k-2} + \cdots + a_2) + \cdots + a_k \\ &= a_k(p^{k-1} + \cdots + p + 1) + \cdots + a_1 \\ &= \big(a_k(p^k - 1) + \cdots + a_0(p^0 - 1)\big)/(p-1) \\ &= (n - a_k - \cdots - a_1 - a_0)/(p-1).\end{aligned}$$

13. For each n, $1 \le n < p$, determine n' as in Ex. 1.2.4–19. We have a unique such n' by law 1.2.4B; and $(n')' = n$. Therefore we can pair off the numbers in groups of two, provided $n' \neq n$. If $n' = n$, we have $n^2 \equiv 1$ (modulo p); hence, as in Ex. 1.2.4–26, $n = 1$ or $n = p - 1$. So $(p-1)! \equiv 1 \cdot 1 \cdot \ldots \cdot (-1)$, since $+1$ and $p-1$ are the only unpaired elements.

14. Of the numbers $1, 2, \ldots, n$ which are *not* multiples of p, there are $\lfloor n/p \rfloor$ complete sets of $p-1$ consecutive elements, each with a product congruent to (-1) (modulo p) by Wilson's Theorem. There are also a_0 left over, which are congruent to $a_0!$ (modulo p), so the contribution from those factors which are not multiples of p is $(-1)^{\lfloor n/p \rfloor} a_0!$. The contribution from the factors which *are* multiples of p is the same as the contribution in $\lfloor n/p \rfloor!$; this argument can therefore be repeated to get the desired formula.

15. $(n!)^3$. There are $n!$ terms. Each term has one from each row and each column, so each term has the value $(n!)^2$.

16. The terms do not approach zero, since the coefficients approach $1/e$.

17. Express the Gamma functions as limits by Eq. (15).

18. $$\prod_{n \ge 1} \frac{n}{(n-1/2)} \frac{n}{(n+1/2)} = \frac{\Gamma(\frac{1}{2})\Gamma(\frac{3}{2})}{\Gamma(1)\Gamma(1)} = 2\Gamma(\tfrac{3}{2})^2.$$

19. (a) Change of variable $t = mt$. (b) Integration by parts. (c) Induction.

20. [For completeness, we prove the stated inequality. Start with the easily verified inequality $1 + x \le e^x$; set $x = \pm t/n$ and raise to the n-th power to get $(1 \pm t/n)^n \le e^{\pm t}$. Hence

$$\begin{aligned} e^{-t} \ge (1 - t/n)^n &= e^{-t}(1 - t/n)^n e^t \ge e^{-t}(1 - t/n)^n (1 + t/n)^n \\ &= e^{-t}(1 - t^2/n^2)^n \ge e^{-t}(1 - t^2/n) \end{aligned}$$

by Ex. 1.2.1–9.]

Now the given integral minus $\Gamma_m(x)$ is

$$\int_m^\infty e^{-t} t^{x-1}\, dt + \int_0^m \left(e^{-t} - \left(\frac{1-t}{m}\right)^m \right) t^{x-1}\, dt.$$

As $m \to \infty$, the first of these approaches zero, since for large t, $t^{x-1} < e^{t/2}$; and the second is less than

$$\frac{1}{m} \int_0^m t^{x+1} e^{-t} < \frac{1}{m} \int_0^\infty t^{x+1} e^{-t}\, dt \to 0.$$

21. If $c(n, j, k_1, k_2, \ldots)$ denotes the appropriate coefficient, we find

$$\begin{aligned} c(n+1, j, k_1, \ldots) = {} & c(n, j-1, k_1 - 1, k_2, \ldots) \\ & + (k_1 + 1) c(n, j, k_1 + 1, k_2 - 1, k_3, \ldots) \\ & + (k_2 + 1) c(n, j, k_1, k_2 + 1, k_3 - 1, k_4, \ldots) + \cdots, \end{aligned}$$

by differentiation. The equations $k_1 + k_2 + \cdots = j$ and $k_1 + 2k_2 + \cdots = n$ are preserved in this induction relationship. We can easily factor $n!/k_1!(1!)^{k_1} k_2!(2!)^{k_2} \ldots$ out of each term appearing on the righthand side of the equation for $c(n+1, j, k_1, \ldots)$, and we are left with $k_1 + 2k_2 + 3k_3 + \cdots = n + 1$. (In the proof it is convenient to assume there are infinitely many k's, although clearly $k_{n+1} = k_{n+2} = \cdots = 0$.) For a table of these coefficients, see the reference at the end of Section 1.2.9.

The solution just given makes use of standard techniques of summation, but it does not give a satisfactory explanation of *why* the formula has this form, nor how it could have been discovered in the first place. Let us examine this question using a combinatorial argument. Write for convenience $w_j = D_u^j w$, $u_k = D_x^k u$. Then $D_x(w_j) = w_{j+1}u_1$ and $D_x(u_k) = u_{k+1}$. By these two rules and the rule for derivative of a product we find

$$\begin{aligned} D_x^1 w &= w_1 u_1 \\ D_x^2 w &= (w_2 u_1 u_1 + w_1 u_2) \\ D_x^3 w &= \big((w_3 u_1 u_1 u_1 + w_2 u_2 u_1 + w_2 u_1 u_2) + (w_2 u_1 u_2 + w_1 u_3)\big), \text{ etc.} \end{aligned}$$

Analogously we may set up a corresponding tableau of set partitions thus:

$$\begin{aligned} \mathfrak{D}^1 &= \{1\} \\ \mathfrak{D}^2 &= (\{2\}\{1\} + \{2, 1\}) \\ \mathfrak{D}^3 &= \big((\{3\}\{2\}\{1\} + \{3, 2\}\{1\} + \{2\}\{3, 1\}) + (\{3\}\{2, 1\} + \{3, 2, 1\})\big), \text{ etc.} \end{aligned}$$

Formally, if $a_1 a_2 \ldots a_j$ is a partition of the set $\{1, 2, \ldots, n-1\}$, define

$$\begin{aligned} \mathfrak{D}a_1 a_2 \ldots a_j = {} & (\{n\}a_1 a_2 \ldots a_j + (a_1 \cup \{n\})a_2 \ldots a_j + a_1(a_2 \cup \{n\}) \ldots a_j + \cdots \\ & + a_1 a_2 \ldots (a_j \cup \{n\}). \end{aligned}$$

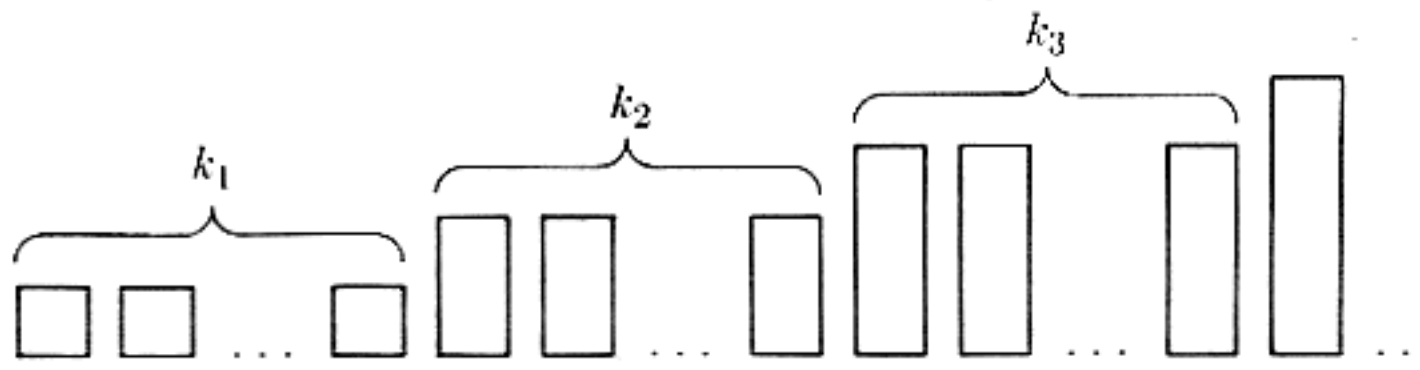
This rule is an exact parallel of the rule

$$\begin{aligned} D_x(w_j u_{r_1} u_{r_2} \ldots u_{r_j}) = {} & w_{j+1} u_1 u_{r_1} u_{r_2} \ldots u_{r_j} + w_j u_{r_1+1} u_{r_2} \ldots u_{r_j} \\ & + w_j u_{r_1} u_{r_2+1} \ldots u_{r_j} + \cdots + w_j u_{r_1} u_{r_2} \ldots u_{r_j+1}, \end{aligned}$$

if we let the term $w_j u_{r_1} u_{r_2} \ldots u_{r_j}$ correspond to a partition $a_1 a_2 \ldots a_j$ with r_t elements in a_t, $1 \le t \le j$. So there is a natural mapping from $\mathfrak{D}^n$ onto $D_x^n w$, and furthermore it is easy to see that $\mathfrak{D}^n$ includes each partition of the set $\{1, 2, \ldots, n\}$ exactly once. (Cf. Ex. 1.2.6–64.)

From these observations we find that if we collect like terms in $D_x^n w$, we obtain a sum of terms $c(k_1, k_2, \ldots) w_j u_1^{k_1} u_2^{k_2} \ldots$, where $j = k_1 + k_2 + \cdots$ and $n = k_1 + 2k_2 + \cdots$, and where $c(k_1, k_2, \ldots)$ is *the number of partitions of* $\{1, 2, \ldots, n\}$ *into* j *parts such that there are* k_t *parts having* t *elements.*

It remains to count these partitions. Consider an array of k_t boxes of capacity t:



The number of ways to put n different elements into these boxes is the multinomial coefficient

$$\binom{n}{1\ 1\ \ldots\ 1\ 2\ 2\ \ldots\ 2\ 3\ 3\ \ldots\ 3\ 4\ \ldots} = n!/1!^{k_1}2!^{k_2}3!^{k_3}\ldots;$$

to get $c(k_1, k_2, k_3, \ldots)$ we should divide this by $k_1!k_2!k_3!\ldots$ since the boxes in each group of k_t are indistinguishable from each other and may be permuted in $k_t!$ ways without affecting the set partition. (This solution is by R. McEliece.)

Faa di Bruno's formula has been generalized in several ways. For what is perhaps the most extensive generalization of the formula, and a list of references to other related work, see the paper by I. J. Good, *Annals of Mathematical Statistics* **32** (1961), 540–541.

22. The hypothesis that $\lim_{n\to\infty} (n+x)!/n!n^x = 1$ is valid for integers x; for example, if x is positive, the quantity is $(1 + 1/n)(1 + 2/n) \cdots (1 + x/n)$, which certainly approaches one. If we also assume that $x! = x(x-1)!$, the hypothesis leads us to conclude immediately that

$$1 = \lim_{n\to\infty} \frac{(n+x)!}{n!n^x} = x! \lim_{n\to\infty} \frac{(x+1)\ldots(x+n)}{n!n^x},$$

which is equivalent to the definition given in the text.

SECTION 1.2.6

1. n, since each combination leaves out one item.

2. 1.

3. $\binom{5}{1}\binom{2}{3}$. The actual number is 635013559600.

4. $2^4 5^2 7^2 17 \cdot 23 \cdot 41 \cdot 43 \cdot 47$.

5. $(10+1)^4 = 10000 + 4(1000) + 6(100) + 4(10) + 1$.

6.

$n = -3$:	1	-3	6	-10	15	-21	28	-36 ...
$n = -2$:	1	-2	3	-4	5	-6	7	-8 ...
$n = -1$:	1	-1	1	-1	1	-1	1	-1 ...

7. $\lfloor n/2 \rfloor$; or, alternatively, $\lceil n/2 \rceil$. It is clear from (3) that for smaller values the binomial coefficient is strictly increasing, and afterwards it decreases to zero.

8. The nonzero entries in each row read the same from left to right as from right to left.

9. One if n is positive or zero; zero if n is negative.

10. (a), (b), (f) follow immediately from (e), and (c), (d) follow from (a), (b), and Eq. (9). Thus it suffices to prove (e). Consider $\binom{n}{k}$ as a fraction, given by Eq. (3) with factors in numerator and denominator. The first $k \bmod p$ factors have no p's in the denominator, and in the numerator and denominator these terms are clearly congruent to the corresponding terms of

$$\binom{n \bmod p}{k \bmod p},$$

which differ by multiples of p. (When dealing with non-multiples of p we may work modulo p in both numerator and denominator, since if $a \equiv c$ and $b \equiv d$ and a/b, c/d are integers, then $a/b \equiv c/d$.) There remain $k - k \bmod p$ factors, which fall into $\lfloor k/p \rfloor$ groups of p consecutive values each. Each group contains exactly one multiple of p; the other $(p-1)$ factors in a group are congruent (modulo p) to $(p-1)!$ so they cancel in numerator and denominator. It remains to investigate the $\lfloor k/p \rfloor$ multiples of p in numerator and denominator; we divide each of these by p and are left with the binomial coefficient

$$\binom{\lfloor (n - k \bmod p)/p \rfloor}{\lfloor k/p \rfloor}.$$

If $k \bmod p \le n \bmod p$, this equals

$$\binom{\lfloor n/p \rfloor}{\lfloor k/p \rfloor}$$

as desired, but if $k \bmod p > n \bmod p$, this equals

$$\binom{\lfloor n/p \rfloor - 1}{\lfloor k/p \rfloor}.$$

However, the other factor

$$\binom{n \bmod p}{k \bmod p}$$

is then zero, so the formula is true in general. [See also N. J. Fine, *AMM* **54** (1947), 589–592.]

11. If

$$\begin{aligned} a &= a_r p^r + \cdots + a_0, \\ b &= b_r p^r + \cdots + b_0, \\ a + b &= c_r p^r + \cdots + c_0, \end{aligned}$$

the value (according to Ex. 1.2.5–12 and Eq. (5)) is

$$(a_0 + \cdots + a_r + b_0 + \cdots + b_r - c_0 - \cdots - c_r)/(p-1).$$

A carry decreases c_j by p and increases c_{j+1} by 1, giving a net change of $+1$ in this formula.

12. By either of the two previous exercises, n must be one less than a power of 2. More generally, $\binom{n}{k}$ is never divisible by the prime p, $0 \le k \le n$, if and only if $n = ap^m - 1$, $1 \le a < p$, $m \ge 0$.

14. $$24\binom{n+1}{5} + 36\binom{n+1}{4} + 14\binom{n+1}{3} + \binom{n+1}{2}$$
$$= \frac{n^5}{5} + \frac{n^4}{2} + \frac{n^3}{3} - \frac{n}{30} = \frac{n(n+1)(n+\frac{1}{2})(3n^2+3n-1)}{15}.$$

15. Induction and (9).

17. We may assume r, s are positive integers. Also

$$\sum_n \binom{r+s}{n} x^n = (1+x)^{r+s} = \sum_n \binom{r}{n} x^n \sum_k \binom{s}{k} x^k$$
$$= \sum_n \sum_k \binom{r}{n-k} x^{n-k} \binom{s}{k} x^k = \sum_n \left(\sum_k \binom{r}{n-k} \binom{s}{k} \right) x^n$$

for all x, so the coefficients of x^n must be identical.

21. The lefthand side is a polynomial of degree n, the righthand side is a polynomial of degree $m+n+1$. We have agreement at $n+1$ points. This is not enough to prove them equal (although when $m = 0$ it proves that the two sides are multiples of some polynomial; and indeed in the case $m = 0$ we find that the equation *is* an identity in s, since it is Eq. (11)).

22. Assume $n > 0$. The k-th term is

$$\frac{1}{n!}\binom{n}{k}\prod_{0<j<k}(r-tk-j)\prod_{0<j<n-k}(n-1-r+tk-j)$$
$$=\frac{(-1)^{k-1}}{n!}\binom{n}{k}\prod_{0<j<k}(-r+tk+j)\prod_{k\le j<n}(-r+tk+j)$$

and the two products give a polynomial of degree $(n-1)$ in k, so the sum over k is zero by Eq. (35).

24. The proof is by induction on n. If $n \le 0$ the identity is obvious. If $n > 0$, we prove it holds for $(r, n-r+nt+m, t, n)$, by induction on the integer $m \ge 0$, using the previous two exercises and the validity for $n-1$. This establishes the identity (r, s, t, n) for infinitely many s, and being a polynomial in s it holds for all s.

25. Using the ratio test and straightforward estimates for large values of k we can prove convergence. (Alternatively using complex variable theory we know the function is analytic in the neighborhood of $x = 1$.) We have

$$1=\sum_{k,j}(-1)^j\binom{k}{j}\binom{r-jt}{k}\frac{r}{r-jt}w^k=\sum_j(-1)^j\frac{r}{r-jt}\sum_k\binom{k}{j}\binom{r-jt}{k}w^k$$
$$=\sum_j\frac{r(-1)^j}{r-jt}\sum_k\binom{r-jt}{j}\binom{r-jt-j}{k-j}w^k$$
$$=\sum_j(-1)^jA_j(r,t)(1+w)^{r-jt-j}w^j.$$

Now let $x = 1/(1+w)$, $z = -w/(1+w)^{1+t}$. This proof is due to H. W. Gould [*AMM* **63** (1956), 84–91]. See also the simple but less elementary derivation by M. Skalsky, *AMM* **69** (1962), 404–405, and the more general formula in exercise 2.3.4.4–33.

26. We could start with the identity

$$\sum_j(-1)^j\binom{k}{j}\binom{r-jt}{k}=t^k$$

and proceed as above. Another way is to differentiate our formula with respect to z; we get

$$\sum_k kA_k(r,t)z^k=z\frac{d(x^r)}{dz}=\frac{(x^{t+1}-x^t)rx^r}{((t+1)x^{t+1}-tx^t)},$$

hence we can obtain the value of

$$\sum_k\left(1-\frac{t}{r}k\right)A_k(r,t)z^k.$$

27. For Eq. (26), multiply the series for $x^{r+1}/((t+1)x-t)$ by the series for x^s, get a series for $x^{r+s+1}/((t+1)x-t)$ in which coefficients of z may be equated to the coefficients arising from the series for $x^{(r+s)+1}/((t+1)x-t)$.

28. Denoting the lefthand side by $f(r, s, t, n)$, we find

$$\binom{r+s}{n} + t \cdot f(r-t-1, s+t, t, n-1) = f(r, s, t, n)$$

by considering the identity

$$\sum_k \binom{r+tk}{k}\binom{s-tk}{n-k}\frac{r}{r+tk} + \sum_k \binom{r+tk}{k}\binom{s-tk}{n-k}\frac{tk}{r+tk} = f(r, s, t, n).$$

29. $(-1)^k\binom{n}{k} \Big/ n! = (-1)^k/k!(n-k)! = (-1)^n \Big/ \prod_{\substack{0\le j\le n \\ j\ne k}} (k-j).$

30. Apply (7) and (19) to get

$$\sum_{k\ge 0}\binom{-m-2k-1}{n-m-k}\binom{2k+1}{k+1}\frac{(-1)^{n-m}}{2k+1} = \sum_{k\ge 1}\binom{-m-2k+1}{n-m-k+1}\binom{2k-1}{k}\frac{(-1)^{n-m}}{2k-1}.$$

Now if we add the term for $k = 0$ we can apply Eq. (26) with

$$r = -1, \quad s = m-2n-1, \quad t = -2, \quad n = n-m+1.$$

So we get the result

$$\binom{-m}{n-m+1}(-1)^{n-m+1} + \binom{-m+1}{n-m+1}(-1)^{n-m} = \binom{-m}{n-m}(-1)^{n-m} = \binom{n-1}{n-m}.$$

This result is the same as our previous formula, when n is positive, but when $n = 0$ the answer we have obtained is correct while $\binom{n-1}{m-1}$ is not. Our derivation has a further bonus, since the answer $\binom{n-1}{n-m}$ is valid for $n \ge 0$ and *all* integers m.

31. We have

$$\begin{aligned}
&\sum_k \sum_j \binom{m-r+s}{j}\binom{n+r-s}{n-k}\binom{r}{m+n-j}\binom{k}{j} \\
&\quad = \sum_j \sum_k \binom{m-r+s}{j}\binom{n+r-s}{n-k}\binom{r}{m+n-j}\binom{m-r+s-j}{k-j} \\
&\quad = \sum_j \binom{m-r+s}{j}\binom{r}{m+n-j}\binom{m+n-j}{n-j}.
\end{aligned}$$

Changing

$$\binom{m+n-j}{n-j} \quad \text{to} \quad \binom{m+n-j}{m}$$

and applying (20) again, we get

$$\sum_j \binom{m-r+s}{j}\binom{r}{m}\binom{r-m}{n-j} = \binom{r}{m}\binom{s}{n}.$$

32. Replace x by $-x$ in (40).

33, 34. We have

$$x^{\overline{n}} = n!\binom{x+n-1}{n}.$$

The equation may therefore be transformed into

$$\binom{x+y+n-1}{n} = \sum_k \binom{x+(1-z)k}{k}\binom{y-1+nz+(n-k)(1-z)}{n-k}\frac{x}{x+(1-z)k}$$

which is a case of (26).

35. For example, we prove the first formula:

$$\sum_k (-1)^{n-k}\left((n-1)\begin{bmatrix} n-1 \\ k \end{bmatrix} + \begin{bmatrix} n-1 \\ k-1 \end{bmatrix}\right) x^k$$

$$= -(n-1)(n-1)!\binom{x}{n-1} + x(n-1)!\binom{x}{n-1} = n!\binom{x}{n}.$$

36. By the binomial formula (assuming n is a nonnegative integer) we get 2^n and δ_{n0}, respectively.

37. When $n > 0$, 2^{n-1}. (The odd and even terms cancel, so each equals half the total sum.)

38. Let $\omega = e^{2\pi i/m}$. Then

$$\sum_{0\le j<m} (1+\omega^j)^n \omega^{-jk} = \sum_t \sum_{0\le j<m} \binom{n}{t} \omega^{j(t-k)}.$$

Now

$$\sum_{0\le j<m} \omega^{rj} = m\delta_{(r \bmod m)0}$$

(it is the sum of a geometric progression), so the righthand sum is

$$m \sum_{t \bmod m = k} \binom{n}{t}.$$

The original sum on the left is

$$\sum_{0\le j<m} (\omega^{-j/2} + \omega^{j/2})^n \omega^{j(n/2-k)} = \sum_{0\le j<m} \left(2\cos\frac{j\pi}{m}\right)^n \omega^{j(n/2-k)}.$$

Since the quantity is known to be real, we may take the real part and obtain the stated formula.

39. $n!$; 0 if $n \ge 2$, ± 1 in the other two cases. (The row sums in the second triangle are not so simple; we will find (Ex. 64) that this gives the number of ways to partition a set of n elements into disjoint sets, i.e. the number of equivalence relations.)

40. Proof of (c): By parts,

$$B(x+1, y) = \frac{t^x(1-t)^y}{y}\Big|_0^1 + \frac{x}{y}\int_0^1 t^{x-1}(1-t)^y\,dt.$$

Now use (b).

41. $m^x B(x, m+1) \to \Gamma(x)$ as $m \to \infty$, regardless of whether m runs through integer values or not (by monotonicity). Hence, $(m+y)^x B(x, m+y+1) \to \Gamma(x)$, and $(m/(m+y))^x \to 1$.

42. $1/kB(k, r-k+1)$.

43. $\int_0^1 dt/t^{1/2}(1-t)^{1/2} = 2\int_0^1 du/(1-u^2)^{1/2} = 2 \arcsin u|_0^1 = \pi$.

45. We have for large r,

$$\frac{1}{k\Gamma(k)} \sqrt{\frac{r}{r-k}} \frac{1}{e^k} \frac{(1-k/r)^k}{(1-k/r)^r} \to \frac{1}{\Gamma(k+1)}.$$

46. $\sqrt{\frac{1}{2\pi}\left(\frac{1}{x}+\frac{1}{y}\right)}\left(1+\frac{y}{x}\right)^x \left(1+\frac{x}{y}\right)^y \cdot \binom{2n}{n} \approx 4^n/\sqrt{\pi n}$.

48. This can be proved by induction, using the fact that

$$0 = \sum_k \binom{n}{k}(-1)^k = \sum_k \binom{n}{k} \frac{(-1)^k k}{k+x} + \sum_k \binom{n}{k} \frac{(-1)^k x}{k+x}.$$

Alternatively, we have

$$B(x, n+1) = \int_0^1 t^{x-1}(1-t)^n \, dt = \sum_k \binom{n}{k}(-1)^k \int_0^1 t^{x+k-1} \, dt.$$

(In fact, the stated sum equals $B(x, n+1)$ for non-integer n also, when the series converges.)

49. $\binom{r}{m} = \sum_k \binom{r}{k}\binom{-r}{m-2k}(-1)^{m+k}$, integer m.

50. The k-th summand is $\binom{n}{k}(-1)^{n-k}(x-kz)^{n-1}x$. Apply Eq. (35).

51. The righthand side is

$$\sum_k \binom{n}{n-k} x(x-kz)^{k-1} \sum_r \binom{n-k}{r}(x+y)^r(-x+kz)^{n-k-r}$$

$$= \sum_r \binom{n}{r}(x+y)^r \sum_k \binom{n-r}{n-r-k} x(x-kz)^{k-1}(-x+kz)^{n-k-r}$$

$$= \sum_r \binom{n}{r}(x+y)^r 0^{n-r} = (x+y)^n.$$

The same device may be used to prove Torelli's sum (Ex. 34) as well as the related formula

$$(x+y)^{\underline{n}} = \sum_k \binom{n}{k} x(x-kz-1)^{\underline{k-1}} (y+kz)^{\underline{n-k}},$$

where $x^{\underline{n}} = x(x-1)\cdots(x-n+1)$.

Another neat proof of Abel's formula comes from the fact that it is readily transformed into the more symmetric identity

$$\sum_k \binom{n}{k} x(x+kz)^{k-1}y(y+(n-k)z)^{n-k-1} = (x+y)(x+y+nz)^{n-1};$$

see Ex. 2.3.4.4–29.

Abel's theorem has been even further generalized by A. Hurwitz [*Acta Mathematica* **26** (1902), 199–203] as follows:

$$\sum x(x+\epsilon_1 z_1+\cdots+\epsilon_n z_n)^{\epsilon_1+\cdots+\epsilon_n-1}(y-\epsilon_1 z_1-\cdots-\epsilon_n z_n)^{n-\epsilon_1-\cdots-\epsilon_n} = (x+y)^n$$

where the sum is over all 2^n choices of $\epsilon_1, \ldots, \epsilon_n = 0$ or 1 independently. This is an identity in $x, y, z_1, \ldots, z_n$, and Abel's formula is the special case $z_1 = z_2 = \cdots = z_n$. Hurwitz's formula follows from the result in Ex. 2.3.4.4–30.

54. Insert minus signs in a checkerboard pattern as shown.

$$\begin{pmatrix} 1 & -0 & 0 & -0 \\ -1 & 1 & -0 & 0 \\ 1 & -2 & 1 & -0 \\ -1 & 3 & -3 & 1 \end{pmatrix}$$

This is equivalent to multiplying a_{ij} by $(-1)^{i+j}$. The result is the desired inverse, by Eq. (34).

55. Insert minus signs as in previous exercise in one triangle, get the inverse of the other. (Eq. (43).)

56. 012 013 023 123 014 024 124 034 134 234 015 025 125 035 135 235 045 145 245 345 016. With c fixed, a and b run through the combinations of c things two at a time; with c, b fixed, a runs through the combinations of b things one at a time. Similarly, we could express all numbers $n = \binom{a}{1} + \binom{b}{2} + \binom{c}{3} + \binom{d}{4}$ with $0 \le a < b < c < d$; the sequence begins 0123 0124 0134 0234 1234 0125 0135 0235 . . . .

59. $(n+1)\binom{n}{k}$.

60. $\binom{n+k-1}{k}$. This formula can be remembered since it is

$$\frac{n(n+1)\cdots(n+k-1)}{k(k-1)\cdots 1},$$

i.e. like Eq. (2) except the numbers in the numerator go up instead of down. A slick way to prove this formula is to note that we want to count the number of integer solutions $(a_1, \ldots, a_k)$ to the relations $1 \le a_1 \le a_2 \le \cdots \le a_k \le n$. This is the same as $0 < a_1 < a_2 + 1 < \cdots < a_k + k - 1 < n + k$; and the number of solutions to $0 < b_1 < b_2 < \cdots < b_k < n + k$ is the number of choices of k distinct things from the set $\{1, 2, \ldots, n+k-1\}$. (This trick is due to H. F. Scherk, *Journal für Math.* **3** (1828), 97; curiously it was also given by W. A. Förstemann in the same journal, vol. 13 (1835), 237, who said "One would almost believe this must have been known long ago, but I have found it nowhere, even though I have consulted many works in this regard."!) The formula may be derived easily with the use of generating functions (cf. Ex. 1.2.9–16).

61. If a_{nm} is the desired quantity, we have by (42), (43), $a_{nm} = na_{(n-1)m} + (-1)^n\delta_{nm}$. Hence the answer is 0 for $n < m$, and $(-1)^m n!/m!$ for $n \geq m$. The same formula is also easily obtained by inversion of (52).

62. Let $A(l, m, n)$ be the sum on the left. The proof proceeds in several steps:

a) $$A(l, m, n+1) = 2\sum (-1)^k \binom{2l}{l+k}\binom{2m}{m+k}\binom{2n+1}{n+k+1}.$$

[Replace $\binom{2n+2}{n+k+1}$ by $\binom{2n+1}{n+k+1} + \binom{2n+1}{n+k}$.]

b) $$(l+n+1)(2m+1)A(l, m, n+1)$$

$$= 2\sum (-1)^k \binom{2l}{l+k}\binom{2m+1}{m+k+1}\binom{2n+1}{n+k+1}(m+k+1)(l-k+n+k+1)$$

$$= 2\sum (-1)^k \binom{2l-1}{l-k-1}\binom{2m+1}{m+k+1}\binom{2n+1}{n+k+1}(m+k+1)(2l)$$

$$+ 2\sum (-1)^k \binom{2l}{l+k}\binom{2m+1}{m+k+1}\binom{2n+1}{n+k+1}(m+k+1)(n+k+1).$$

c) When $l > 0$, $$\sum (-1)^k \binom{2l-1}{l-k-1}\binom{2m+1}{m+k+1}\binom{2n+1}{n+k+1} = 0.$$

(Replace k by $-k-1$.)

d) Because of (c), the last derived quantity in (b) is symmetric in m and n. Hence $(l+n+1)(2m+1)A(l, m, n+1) = (l+m+1)(2n+1)A(l, m+1, n)$.

e) From (d), the result follows by induction on n. (*Note:* It is hard to say who originated this identity; the case $l = m = n$ was published by A. C. Dixon, *Messenger of Math.* **20** (1891), 79–80. See papers by P. A. MacMahon, *Quarterly Journal of Pure and Applied Math.* **33** (1902), 274–288, and John Dougall, *Proc. Edinburgh Math. Society* **25** (1906), 114–132.)

64. Let $f(n, m)$ be the number of partitions of $\{1, 2, \ldots, n\}$ into m parts. Clearly $f(1, m) = \delta_{1m}$. If $n > 1$, the partitionings are of two varieties: (a) The element n alone forms a set of the partition; there are $f(n-1, m-1)$ ways to construct partitions like this. (b) The element n appears together with another element; there are m ways to insert n into any m-partition of $\{1, 2, \ldots, n-1\}$, hence there are $mf(n-1, m)$ ways to construct partitions like this. We therefore conclude $f(n, m) = f(n-1, m-1) + mf(n-1, m)$, and by induction $f(n, m) = \left\{{n \atop m}\right\}$.

SECTION 1.2.7

1. 0; 1; 3/2.
2. Replace each term $1/(2^m + k)$ by the upper bound $1/2^m$.
3. $S_{2^m}^{(r)} \leq \sum_{0 \leq k \leq m} (1/2^r - 1)^k$. $2^{r-1}/(2^{r-1} - 1)$ is an upper bound.
4. (b) and (c).

5. 9.78760 60360 44382 . . .

6. Induction and Eq. 1.2.6–42.

7. $T(m+1, n) - T(m, n) = 1/(m+1) - 1/(mn+1) - \cdots - 1/(mn+n) < 1/(m+1) - 1/(mn+n) - \cdots - 1/(mn+n) = 1/(m+1) - n/(m+1) \leq 0$. The maximum value occurs at $m = n = 1$, and the minimum is approached when m and n get very large. By Eq. (3) the greatest lower bound is γ, which is never actually attained. A generalization of this result appears in *AMM* **70** (1963), 575–577.

8. By Stirling's approximation, $\ln n!$ is approximately $(n+\frac{1}{2})\ln n - n + \ln\sqrt{2\pi}$; also $\sum_{1\leq k\leq n} H_k$ is approximately $(n+1)\ln n - n(1-\gamma) + (\gamma+\frac{1}{2})$; the difference is approximately $\gamma n + \frac{1}{2}\ln n + .158$.

9. $-1/n$.

10. Break left side into two sums, change k to $k+1$ in second sum.

11. $2 - H_{n+1}/n - 1/(n+1)$.

12. 1.000 . . . is correct to over three hundred decimal places!

14. See section 1.2.3, example 2. The second sum is $\frac{1}{2}(H_{n+1}^2 - H_{n+1}^{(2)})$.

15. $\sum_{1\leq j\leq n} (1/j) \sum_{j\leq k\leq n} H_k$ can be summed by formulas in the text; the answer is $(n+1)H_n^2 - (2n+1)H_n + 2n$.

16. $H_{2n+1} - \frac{1}{2}H_n$.

17. Taking the denominator to be $(p-1)!$, which is a multiple of the true denominator but not a multiple of p, we must show only that the corresponding numerator, $(p-1)!/1 + (p-1)!/2 + \cdots + (p-1)!/(p-1)$, *is* a multiple of p. Modulo p, $(p-1)!/k \equiv (p-1)!k'$, where k' can be determined by the relation $kk' \bmod p = 1$. The set $\{1', 2', \ldots, (p-1)'\}$ is just the set $\{1, 2, \ldots, (p-1)\}$; so the numerator is congruent to $(p-1)!(1+2+\cdots+p-1) \equiv 0$. In fact the numerator is known to be a multiple of p^2 when $p > 3$; see Hardy and Wright, *The Theory of Numbers*, section 7.8.

18. If $n = 2^k m$ where m is odd, the sum equals $2^{2k}m_1/m_2$ where m_1 and m_2 are both odd. *AMM* **67** (1960), 924–925.

19. Only $n = 0$, $n = 1$. For $n \geq 2$, let $k = \lfloor \log_2 n \rfloor$. There is precisely one term whose denominator is 2^k, so $2^{k-1}H_n - \frac{1}{2}$ is a sum of terms involving only odd primes in the denominator. If H_n were an integer, $2^{k-1}H_n - \frac{1}{2}$ would have a denominator equal to 2.

20. Expand the integrand term by term. See also *AMM* **69** (1962), 239, and an article by H. W. Gould, *Mathematics Magazine* **34** (1961), 317–321.

21. $H_{n+1}^2 - H_{n+1}^{(2)}$.

22. $(n+2)(H_{n+1}^2 - H_{n+1}^{(2)}) - 2(n+1)H_n + 2n$.

23. $\Gamma'(n+1)/\Gamma(n+1) = 1/n + \Gamma'(n)/\Gamma(n)$, since $\Gamma(x+1) = x\Gamma(x)$. Hence $H_n = \gamma + \Gamma'(n+1)/\Gamma(n+1)$. The function $H_n - \gamma$ is called the *psi function* or the *digamma function*.

24. It is

$$x \lim_{n\to\infty} e^{(H_n - \ln n)x} \prod_{1\leq k\leq n} \left(\left(1+\frac{x}{k}\right)e^{-x/k}\right) = \lim_{n\to\infty} \frac{x(x+1)\cdots(x+n)}{n^x n!}.$$

Note: The generalization of H_n considered in the previous exercise is therefore equal to $H_x^{(r)} = \sum_{k\geq 0} \left(1/(k+1)^r - 1/(k+1+x)^r\right)$, when $r = 1$; the same idea can be used for larger values of r.

SECTION 1.2.8

1. F_{k+2}; the answer is $F_{14} = 377$ pairs.

2. $\ln(\phi^{1000}/\sqrt{5}) = 1000 \ln \phi - \frac{1}{2} \ln 5 = 480.40711$; $\log_{10} F_{1000}$ is $1/(\ln 10)$ times this, or $208.6\underline{4}$; F_{1000} is therefore a 209-digit number whose leading digit is 4.

4. 0, 1, 5; afterwards F_n increases too fast.

5. 0, 1, 12.

6. Induction. (The equation holds for *negative* n also, cf. Ex. 8.)

7. If d is a proper divisor of n, F_d divides F_n. Now F_d is greater than one and less than F_n provided d is greater than 2. The only non-prime number which has no proper factor greater than 2 is $n = 4$ (since if $d = 2$, $n/d \geq 2$). $F_4 = 3$ is the only exception.

8. $F_{-1} = 1$; $F_{-2} = -1$; $F_{-n} = (-1)^{n+1} F_n$ by induction on n.

9. Not (15). The others are valid, by an inductive argument which proves something true for $n - 1$ assuming it true for n and greater.

10. When n is even, it is greater, and when n is odd, it is less by Eq. (14).

11. Induction; cf. Ex. 9. This can also be observed to be a special case of Ex. 13(a).

12. If $\mathcal{G}(z) = \sum \mathcal{F}_n z^n$, $(1 - z - z^2)\mathcal{G}(z) = z + F_0 z^2 + F_1 z^3 + \cdots = z + z^2 G(z)$. Hence $\mathcal{G}(z) = G(z) + zG(z)^2$; by Eq. (17) we derive $\mathcal{F}_n = \big((3n+3)/5\big)F_n - (n/5)F_{n+1}$.

13. (a) $a_n = rF_{n-1} + sF_n$. (b) Since $(b_{n+2} + c) = (b_{n+1} + c) + (b_n + c)$, we may consider a new sequence $b'_n = b_n + c$. Apply part (a) to b'_n, and we obtain the answer $cF_{n-1} + (c+1)F_n - c$.

14. $a_n = F_{m+1}F_{n-1} + (F_{m+2} + 1)F_n - \binom{n}{m} - \binom{n+1}{m-1} - \cdots - \binom{n+m}{0}$.

15. $c_n = xa_n + yb_n + (1 - x - y)F_n$.

16. F_{n+1}. Induction, and $\binom{n+1-k}{k} = \binom{n-k}{k} + \binom{(n-1)-(k-1)}{k-1}$.

17. $(x^{n+k} - y^{n+k})(x^{m-k} - y^{m-k}) - (x^n - y^n)(x^m - y^m)$
$= (xy)^n (x^{m-n-k} - y^{m-n-k})(x^k - y^k)$.

Now set $x = \phi$, $y = \hat{\phi}$, and divide by $(\sqrt{5})^2$.

18. It is F_{2n+1}.

19. Let $u = \cos 72°$, $v = \cos 36°$. We have $u = 2v^2 - 1$; $v = 1 - 2\sin^2 18° = 1 - 2u^2$. Hence $u + v = 2(v^2 - u^2)$, i.e. $1 = 2(v - u)$, so $4v^2 - 2v - 1 = 0$. $v = \frac{1}{2}\phi$.

20. $F_{n+2} - 1$.

21. Multiply by $x^2 + x - 1$; the solution is $(x^{n+1}F_{n+1} + x^{n+2}F_n - x)/(x^2 + x - 1)$. If the denominator is zero, x is $1/\phi$ or $1/\hat{\phi}$; then the solution is

$$\big((n+1)x^n F_{n+1} + (n+2)x^{n+1}F_n - 1\big)/(2x+1).$$

22. F_{m+2n}; see next exercise with $t = 2$.

23. $$\frac{1}{\sqrt{5}}\sum_k \binom{n}{k}(\phi^k F_t^k F_{t-1}^{n-k}\phi^m - \hat{\phi}^k F_t^k F_{t-1}^{n-k}\hat{\phi}^m)$$

$$= \frac{1}{\sqrt{5}}\left(\phi^m(\phi F_t + F_{t-1})^n - \hat{\phi}^m(\hat{\phi}F_t + F_{t-1})^n\right) = F_{m+tn}.$$

24. F_{n+1} (expand by cofactors in first row).

25. $2^n\sqrt{5}\,F_n = (1+\sqrt{5})^n - (1-\sqrt{5})^n$.

26. By Fermat's theorem, $2^{p-1} \equiv 1$; now apply the previous exercise and Ex. 1.2.6–10(b).

27. It is true if $p = 2$. Otherwise, modulo p, $F_{p-1}F_{p+1} - F_p^2 = -1$; from the previous exercise and Fermat's theorem, $F_{p-1}F_{p+1} \equiv 0$. Only one of these can be a multiple of p since $F_{p+1} = F_p + F_{p-1}$.

28. $\hat{\phi}^n$. *Note:* the solution to the relation $a_{n+1} = Aa_n + B^n$, $a_0 = 0$, is

$$a_n = (A^n - B^n)/(A - B) \text{ if } A \neq B, \qquad a_n = nA^{n-1} \text{ if } A = B.$$

29.

$\left(\binom{n}{0}\right)$	$\left(\binom{n}{1}\right)$	$\left(\binom{n}{2}\right)$	$\left(\binom{n}{3}\right)$	$\left(\binom{n}{4}\right)$	$\left(\binom{n}{5}\right)$	$\left(\binom{n}{6}\right)$
1	0	0	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	0	0	0
1	2	2	1	0	0	0
1	3	6	3	1	0	0
1	5	15	15	5	1	0
1	8	40	60	40	8	1

(b) follows from (6).

30. By induction on m, the statement being obvious when $m = 1$:

a) $$\sum_k \left(\binom{m}{k}\right)(-1)^{\lceil (m-k)/2\rceil} F_{n+k}^{m-2}F_k = F_n \sum_k \left(\binom{m-1}{k-1}\right)(-1)^{\lceil (m-k)/2\rceil} F_{n+k}^{m-2} = 0.$$

b) $$\sum_k \left(\binom{m}{k}\right)(-1)^{\lceil (m-k)/2\rceil} F_{n+k}^{m-2}(-1)^k F_{m-k}$$

$$= F_n \sum_k \left(\binom{m-1}{k}\right)(-1)^{\lceil (m-1-k)/2\rceil} F_{n+k}^{m-2}(-1)^n = 0.$$

c) Since $(-1)^k F_{m-k} = F_{k-1}F_m - F_kF_{m-1}$, we conclude from (a), (b) that

$$\sum_k \left(\binom{m}{k}\right)(-1)^{\lceil (m-k)/2\rceil} F_{n+k}^{m-2}F_{k-1} = 0,$$

since $F_m \neq 0$.

d) Since $F_{n+k} = F_{k-1}F_n + F_kF_{n+1}$ the result follows from (a) and (c). This result may also be proved in slightly more general form by using the "q-binomial theorem" in Ex. 1.2.6–58. See also J. Riordan, *Duke Math J.* **29** (1962), 5–12.

31. Ex. 8 and 11.

32. Modulo F_n the Fibonacci sequence is $0, 1, \ldots, F_{n-1}, 0, F_{n-1}, -F_{n-2}, \ldots$.

33. One can use the properties of Chebyshev polynomials, if they are known. Directly, we find $\cos z = \frac{1}{2}(e^{iz} + e^{-iz}) = -i/2$. Then use the fact that $\sin(n+1)z + \sin(n-1)z = 2\sin(nz)\cos z$.

34. Prove that the only possible value for F_{k_1} is the largest Fibonacci number less than or equal to n; hence $n - F_{k_1}$ is less than F_{k_1-1}, and by induction there is a unique representation of $n - F_{k_1}$. The outline of this proof is quite similar to the proof of the unique factorization theorem.

36. We may consider the infinite string S_∞, since S_n $(n > 1)$ consists of the first F_n letters of S_∞. There are no double a's, no triple b's. S_n contains F_{n-2} a's, F_{n-1} b's. If we express $m - 1$ in the Fibonacci number system as in Ex. 34, the m-th letter of S_∞ is a if and only if $k_r = 2$. The k-th letter of S_∞ is b if and only if $\lfloor (k+1)\phi^{-1} \rfloor - \lfloor k\phi^{-1} \rfloor = 1$; the number of b's in the first k letters is therefore $\lfloor (k+1)\phi^{-1} \rfloor$.

37. [*Fibonacci Quart.* **1** (Dec. 1963), 9–12.] Consider the Fibonacci number system of Ex. 34; if $n = F_{k_1} + \cdots + F_{k_r} > 0$ in that system, let $\mu(n) = F_{k_r}$. Let $\mu(0) = \infty$. We find that (A) If $n > 0$, $\mu(n - \mu(n)) > 2\mu(n)$. *Proof:* $\mu(n - \mu(n)) = F_{k_r-1} \geq F_{k_r+2} > 2F_{k_r}$ since $k_r \geq 2$. (B) If $0 < m < F_k$, $\mu(m) \leq 2(F_k - m)$. *Proof:* Let $\mu(m) = F_j$; $m \leq F_j + F_{j+2} + \cdots + F_{k-1-(k-1-j) \bmod 2} = -F_{j-1} + F_{k-(k-1-j) \bmod 2} \leq -\frac{1}{2}F_j + F_k$. (C) If $0 < m < \mu(n)$, $\mu(n - \mu(n) + m) \leq 2(\mu(n) - m)$. *Proof:* This follows from (B). (D) If $0 < m < \mu(n)$, $\mu(n - m) \leq 2m$. *Proof:* Set $m = \mu(n) - m$ in (C).

Now we will prove that if there are n chips, and if at most q may be taken in the next turn, there is a winning move iff $\mu(n) \leq q$. *Proof:* (a) If $\mu(n) > q$ all moves leave a position n', q' with $\mu(n') \leq q'$. [This follows from (D), above.] (b) If $\mu(n) \leq q$, we can either win on this move (if $q \geq n$) or we can make a move which leaves a position n', q' with $\mu(n') > q'$. [This follows from (A) above, our move is to *take* $\mu(n)$ *chips*.] It can be seen that the set of all winning moves, if $n = F_{k_1} + \cdots + F_{k_r}$, is to remove $F_{k_j} + \cdots + F_{k_r}$, for some j with $1 \leq j \leq r$, provided that $j = 1$ or $F_{k_j-1} > 2(F_{k_j} + \cdots + F_{k_r})$.

If $n = 1000$, the Fibonacci representation is $987 + 13$; the *only* lucky move to force a victory is to take 13 chips. The first player can always win unless n is a Fibonacci number.

The solution to considerably more general games of this type has been obtained by A. Schwenk (to be published).

39. $(3^n - (-2)^n)/5$.

SECTION 1.2.9

1. $1/(1-2z)+1/(1-3z)$.

2. Follows from (6), since $\binom{n}{k} = n!/k!(n-k)!$.

3. $G'(z) = \ln\big(1/(1-z)\big)/(1-z)^2 + 1/(1-z)^2$. From this and the significance of $G(z)/(1-z)$, we have $\sum_{1\le k\le n-1} H_k = nH_n - n$; this agrees with Eq. 1.2.7–8.

4. Put $t = 0$.

5. The coefficient of z^k is, by (11), (22),

$$(n-1)!\sum_{0\le j<k}\left\{{j \atop n-1}\right\}\binom{k}{j}.$$

Now apply Eqs. 1.2.6–42 and 1.2.6–48.

A simpler method uses differentiation and 1.2.6–42.

6. $(\ln(1/(1-z)))^2$; the derivative is twice the generating function for the harmonic numbers; the sum is therefore $2H_{n-1}/n$.

8. $1/(1-z)(1-z^2)(1-z^3)\cdots$. [This is historically one of the first applications of generating functions. For an interesting account of L. Euler's eighteenth-century researches concerning this generating function, see G. Polya, *Induction and Analogy in Mathematics* (Princeton: Princeton University Press, 1954), Chapter 6.]

9. $\frac{1}{24}S_1^4+\frac{1}{4}S_1^2S_2+\frac{1}{8}S_2^2+\frac{1}{3}S_1S_3+\frac{1}{4}S_4$.

10. $G(z)=(1+x_1z)\cdots(1+x_nz)$. Taking logarithms as in the derivation of Eq. (34), we have the same formulas except (24) replaces (25), and the answer is exactly the same except $S_2, S_4, S_6, \ldots$ are replaced by $-S_2, -S_4, -S_6$, etc. We have $b_1=S_1$, $b_2=\frac{1}{2}S_1^2-\frac{1}{2}S_2$, $b_3=\frac{1}{6}S_1^3-\frac{1}{2}S_1S_2+\frac{1}{3}S^3$, $b_4=\frac{1}{24}S_1^4-\frac{1}{4}S_1^2S_2+\frac{1}{8}S_2^2+\frac{1}{3}S_1S_3-\frac{1}{4}S_4$. (Cf. Ex. 9.) The numbers b_m are called the *elementary symmetric functions* of the x_j, and the formulas derived in this exercise are called *Newton's identities*.

11. We find $z^2G'(z)+zG(z)=G(z)-1$. The solution to this differential equation is $G(z)=(-1/z)e^{-1/z}(E_1(-1/z)+C)$, where $E_1(x)=\int_x^\infty e^{-t}\,dt/t$ and C is a constant. This function is very ill-behaved in the neighborhood of $z=0$ and $G(z)$ has no power series expansion. Indeed, since $\sqrt[n]{n!}\approx n/e$ is not bounded, the generating function does not converge in this case; it is, however, an asymptotic expansion for the above function, when $z<0$.

12. $$\sum_{m,n\ge0}a_{mn}w^mz^n=\sum_{m,n\ge0}\binom{n}{m}w^mz^n=\sum_{n\ge0}(1+w)^nz^n=1/(1-z-wz).$$

13. $$\int_n^{n+1}e^{-st}f(t)\,dt=\frac{a_0+\cdots+a_n}{s}(e^{-sn}-e^{-s(n+1)}).$$

Adding these together, we find $\mathbf{L}f(s)=G(e^{-s})/s$.

14. Cf. Ex. 1.2.6–38.

15. $G_n(z)=G_{n-1}(z)+zG_{n-2}(z)$, so we find $H(w)=1/(1-w-zw^2)$. Hence, ultimately, we find

$$G_n(z)=\left(\left(\frac{1+\sqrt{1+4z}}{2}\right)^{n+1}-\left(\frac{1-\sqrt{1+4z}}{2}\right)^{n+1}\right)\Big/\sqrt{1+4z}.$$

16. $$G_{nr}(z)=(1+z+\cdots+z^r)^n=\left(\frac{1-z^{r+1}}{1-z}\right)^n.$$

The case $r=\infty$ is of interest.

17. $$\sum_k \binom{-w}{k}(-z)^k = \sum_k \frac{w(w+1)\cdots(w+k-1)}{k(k-1)\cdots 1} z^k = \sum_{n,k} \begin{bmatrix} k \\ n \end{bmatrix} z^k w^n / k!.$$

(Alternatively, write it as $e^{w \ln(1/(1-z))}$ and expand first by powers of w.)

18. (a) For fixed n and varying r, the generating function is

$$G_n(z) = (1+z)(1+2z)\cdots(1+nz) = z^{n+1}\left(\frac{1}{z}\right)\left(\frac{1}{z}+1\right)\left(\frac{1}{z}+2\right)\cdots\left(\frac{1}{z}+n\right)$$
$$= \sum_k \begin{bmatrix} n+1 \\ k \end{bmatrix} z^{n+1-k}$$

by Eq. (27). Hence the answer is

$$\begin{bmatrix} n+1 \\ n+1-r \end{bmatrix}.$$

(b) Similarly, the generating function is

$$\frac{1}{1-z}\cdot\frac{1}{1-2z}\cdot\cdots\cdot\frac{1}{1-nz} = \sum_k \begin{Bmatrix} k \\ n \end{Bmatrix} z^{k-n}$$

by Eq. (28), so the answer is

$$\begin{Bmatrix} n+r \\ n \end{Bmatrix}$$

19. $$\sum_{n\geq 0}\left(\frac{1}{n+p/q}-\frac{1}{n+1}\right)x^{p+nq}$$
$$= x^{p-q}\ln(1-x^q) - \sum_{1\leq k\leq q} \omega^{-kp}\ln(1-\omega^k x) = S_1 + S_2 + S_3,$$

where $\omega = e^{2\pi i/q}$ and where S_1, S_2, S_3 are defined below. Now

$$\lim_{x\to 1-} S_1 = \lim_{x\to 1-} x^{p-q}\ln\left(\frac{1-x^q}{1-x}\right) = \ln q;$$
$$\lim_{x\to 1-} S_2 = \lim_{x\to 1-} (x^{p-q}-1)\ln(1-x) = 0;$$

and

$$\lim_{x\to 1-} S_3 = -\sum_{0<k<q} \omega^{-kp}\ln(1-\omega^k).$$

From the identity

$$\ln(1-e^{i\theta}) = \ln\left(2e^{i(\theta-\pi)/2}\cdot\frac{e^{i\theta/2}-e^{-i\theta/2}}{2i}\right) = \ln 2 + \tfrac{1}{2}i(\theta-\pi) + \ln\sin\frac{\theta}{2},$$

we may write the latter sum as $S_4 + S_5$ where

$$S_4 = -\sum_{0<k<q} \omega^{-kp}\ln\sin\frac{k}{q}\pi = -\sum_{0<k<q/2} (\omega^{-kp}+\omega^{-(q-k)p})\ln\sin\frac{k}{q}\pi$$
$$= -2\sum_{0<k<q/2} \cos\frac{2\pi pk}{q}\ln\sin\frac{k}{q}\pi;$$

and

$$S_5 = -\sum_{0<k<q} \omega^{-kp}\left(\ln 2 - \frac{i\pi}{2} + \frac{ik\pi}{q}\right) = \ln 2 - \frac{i\pi}{2} - \frac{i\pi}{(\omega^{-p}-1)}.$$

Finally,

$$\frac{-i}{2} - \frac{i}{(\omega^{-p}-1)} = -\frac{i}{2}\left(\frac{1+\omega^p}{1-\omega^p}\right) = \frac{i}{2}\left(\frac{\omega^{p/2}+\omega^{-p/2}}{\omega^{p/2}-\omega^{-p/2}}\right) = \tfrac{1}{2}\cot\frac{p}{q}\pi.$$

SECTION 1.2.10

1. $1/n$; this is the probability that $X[n]$ is the largest.

2. $G''(1) = \sum k(k-1)p_k$, $G'(1) = \sum kp_k$.

3. min 0, ave 6.49, max 999, dev 2.42. (Note that $H_n^{(2)}$ is approximately $\pi^2/6$; see Eq. 1.2.7–7.)

4. $\binom{n}{k} p^k q^{n-k}$.

5. Mean is $36/5 = 7.2$; standard deviation is $6\sqrt{2}/5 \approx 1.697$.

7. The probability that $A = k$ is p_{mk}. For we may consider the values to be $1, 2, \ldots, m$. Given any partitioning of the n positions into m disjoint sets, there are $m!$ ways to assign the numbers $1, \ldots, m$ to these sets. Algorithm M treats these values as if only the rightmost element of each set were present; so, p_{mk} is the average for any fixed partitioning. For example, if $n = 5$, $m = 3$, one partition is

$$\{X[1], X[4]\}\,\{X[2], X[5]\}\,\{X[3]\};$$

the arrangements possible are 12312, 13213, 21321, 23123, 31231, 32132. In every partition we get the same percentage of arrangements with $A = k$.

On the other hand, if more information is given the probability distribution changes. If $n = 3$, $m = 2$, the above argument considers the six possibilities 122, 212, 221, 211, 121, 112; if we know there are two 2's and one 1, then only the first three of these possibilities is to be considered. This interpretation is not consistent with the statement of the exercise, however.

8. $M(M-1)\cdots(M-n+1)/M^n = M!/(M-n)!M^n$. The larger M is, the closer this probability gets to one.

9. Let q_{nm} be the probability that exactly m distinct values occur; then from the recurrence

$$q_{nm} = \frac{M-m+1}{M} q_{(n-1)(m-1)} + \frac{m}{M} q_{(n-1)m}$$

we deduce that

$$q_{nm} = M! \left\{ {n \atop m} \right\} \Big/ (M-m)!M^n.$$

See also Ex. 1.2.6–64.

10. This is $q_{nm}p_{mk}$ summed over all m, i.e.

$$\frac{1}{M^n} \sum_m \binom{M}{m} \left\{ {n \atop m} \right\} \left[{m \atop k+1} \right].$$

There does not appear to be a simple formula for the average, which is

$$\left(\frac{1}{M^n}\sum_m \binom{M}{m}\left\{{n \atop m}\right\} m!H_m\right) - 1.$$

11. The first identity is obvious by writing out the power series for e^{kt}. For the second, let $u = 1 + M_1 t + M_2 t^2/2! + \cdots$; when $t = 0$ we have $u = 1$ and $D_t^k u = M_k$. Also, $D_u^j(\ln u) = (-1)^{j-1}(j-1)!/u^j$.

12. Since this is a product, we add the semi-invariants of each term. If $H(z) = z^n$, $H(e^t) = e^{nt}$, so we find $k_1 = n$ and all others are zero. Therefore, mean$(G_1) =$ $n +$ mean (G), and all other semi-invariants are unchanged. (This accounts for the name "semi-invariant.")

13.
$$G_n(z) = \frac{\Gamma(n+z)}{\Gamma(z+1)n!} = \frac{1}{\Gamma(z+1)}\,\frac{(n+z)^z}{n+z}\,e^{-z}\left(1+\frac{z}{n}\right)^n\left(1+O\left(\frac{1}{n}\right)\right)$$
$$= \frac{n^{z-1}}{\Gamma(z+1)}\left(1+O\left(\frac{1}{n}\right)\right).$$

Let $z = z_n = \exp(t/\sigma_n)$. Now $z_n \to 1$, so the $\Gamma(z+1)$ term may be ignored as $n \to \infty$. The limit now is

$$\exp\left(-t\mu_n/\sigma_n + (e^{t/\sigma_n} - 1)\ln n\right)$$
$$= \exp\left(t\left(\frac{\ln n - \mu_n}{\sigma_n}\right) + \frac{t^2 \ln n}{2\sigma_n^2} + O\left(\frac{1}{\ln n}\right)\right) \to \exp\left(\frac{t^2}{2}\right).$$

14. $e^{-tpn/\sqrt{pqn}}(q + pe^{t/\sqrt{pqn}})^n = (qe^{-tp/\sqrt{pqn}} + pe^{tq/\sqrt{pqn}})^n$. Expand the exponentials in power series, get $(1 + t^2/2n + O(n^{-3/2}))^n = \exp\left(n\ln(1 + t^2/2n + O(n^{-3/2}))\right) =$ $\exp\left(t^2/2 + O(n^{-1/2})\right) \to \exp(t^2/2)$.

15. $G_n(z)$ in Eq. (8). A generating function for probabilities may always be interpreted as the average value of a quantity, in this way.

16. (a) $\sum_{k\geq 0} e^{-\mu}(\mu z)^k/k! = e^{\mu(z-1)}$. (b) $\ln e^{\mu(e^t-1)} = \mu(e^t - 1)$, so all semi-invariants equal μ. (c) $\exp(-tnp/\sqrt{np}) \times \exp\left(np(t/\sqrt{np} + t^2/2np + O(n^{-3/2})\right) =$ $\exp\left(t^2/2 + O(n^{-1/2})\right)$.

17. (a) The coefficients of $f(z)$, $g(z)$ are non-negative and $f(1) = g(1) = 1$. Clearly $h(z)$ shares these same characteristics since $h(1) = g(f(1))$ and the coefficients of h are polynomials in those of f, g with non-negative coefficients. (b) Let $f(z) = \sum p_k z^k$ where p_k is the probability that some event yields a "score" of k. Let $g(z) = \sum q_k z^k$ where q_k is the probability that the event described by f happens exactly k times (each occurrence of the event being independent of the others). Then $h(z) = \sum r_k z^k$, where r_k is the probability that the sum of the scores of the events that occurred is equal to k. (This is easy to see if we observe that $f(z)^k = \sum s_t z^t$, where s_t is the probability that a total score t is obtained in k independent occurrences of the event.) Example: If f gives the probabilities that a man has k male offspring, and if g gives the probabilities that there are k males in the n-th generation, then h gives the probabilities that there are k males in the $(n+1)$st generation, assuming independence. (c) mean (h) = mean (g) mean (f); var (h) = var (g) mean$^2(f)$ + mean (g) var (f).

18. Consider the choice of $X[1], \ldots, X[n]$ as a process in which we first place all the n's, then place all the $(n-1)$'s among these n's, ..., finally place the ones among the rest. As we place the r's among the numbers $r+1, \ldots, n$, the number of local maxima from right to left increases by one if and only if we put an r at the extreme right. This happens with probability $k_r/(k_r + k_{r+1} + \cdots + k_n)$.

SECTION 1.2.11.1

1. Zero.

2. Each O symbol represents a different approximate quantity; since the lefthand side may be $f(n) - (-f(n)) = 2f(n)$, the best we can say is $O(f(n)) - O(f(n)) = O(f(n))$. To prove the latter, note that if $|x_n| \le M_1|f(n)|$ and $|y_n| \le M_2|f(n)|$, then $|x_n - y_n| \le |x_n| + |y_n| \le (M_1 + M_2)|f(n)|$. (Signed, J. H. Quick, student.)

3. $n(\ln n) + \gamma n + O(\sqrt{n} \ln n)$.

4. $|x_n| \le \max(x_1/f(1), \ldots, x_{n_0}/f(n_0), M)|f(n)|$.

5. (a) Take $M = |c_0|/r^m + |c_1|/r^{m-1} + \cdots + |c_m|$. (Cf. the text following Eq. (3).) (b) Disproof: Let $P(x) = 1$, $m = 1$; then $|1| > Mx$ when $x < 1/M$; the condition therefore fails for all choices of M.

6. A variable number, n, of O-symbols has been replaced by a single O-symbol, falsely implying that a single value of M will suffice for each term $|kn| \le Mn$. The given sum is actually $O(n^3)$, as we know. The last equality, $\sum_{1 \le k \le n} O(n) = O(n^2)$, is perfectly valid.

7. If x is positive, the power series tells us $e^x > x^{m+1}/(m+1)!$, hence the ratio of e^x/x^m is unbounded by any M.

8. Replace n by e^n and apply the previous exercise.

9. If $|f(x)| \le M|x|^m$, $e^{f(x)} \le e^{M|x|^m} = 1 + |x|^m(M + M^2|x|^m/2! + M^3|x|^{2m}/3! + \cdots) \le 1 + |x|^m(M + M^2r^m/2! + M^3r^{2m}/3! + \cdots)$.

10. "If $f(x) = O(x^m)$, $|x| \le r$, there exists a positive number r' such that $\ln(1+f(x)) = O(x^m)$, $|x| \le r'$." *Proof:* Take r', r'' such that $|f(x)| \le r'' < 1$ when $|x| \le r'$. Then $|\ln(1+f(x))| \le |f(x)| + \frac{1}{2}|f(x)|^2 + \frac{1}{3}|f(x)|^3 + \cdots \le |x|^m M(1 + \frac{1}{2}r'' + \frac{1}{3}r''^2 + \cdots)$.

11. Apply Eq. (11) with $m = 1$, $x = \ln n/n$. This is justified since $\ln n/n$ remains bounded by some positive value r (it approaches zero for large n).

12. $\ln a + (\ln a)^2/2n + (\ln a)^3/6n^2 + O(n^{-3})$.

SECTION 1.2.11.2

1. $x = (B_0 + B_1 x + B_2 x^2/2! + \cdots)e^x - (B_0 + B_1 x + B_2 x^2/2! + \cdots)$; apply Eq. 1.2.9–11.

2. The function $B_{m+1}(\{x\})$ must be continuous, for the integration by parts.

3. $|R_{2k}| \leq \left|\frac{B_{2k}}{(2k)!}\right| \int_1^n |f^{(2k)}(x)|\, dx.$

4. $$\sum_{0\le k<n} k^m = \frac{1}{1+m}x^m + \sum_{1\le k\le m} \frac{B_k}{k!}\frac{m!}{(m-k+1)!}n^{m-k+1}$$
$$= \frac{1}{m+1}B_{m+1}(n) - \frac{1}{m+1}B_{m+1}.$$

5. It follows that

$$K = \sqrt{2}\lim_{n\to\infty}\frac{2^{2n}(n!)^2}{\sqrt{n}(2n)!};$$

$$K^2 = \lim_{n\to\infty}\frac{2}{n}\frac{n^2(n-1)^2\cdots(1)^2}{(n-\frac{1}{2})^2(n-\frac{3}{2})^2\cdots(\frac{1}{2})^2} = 4\frac{2\cdot2\cdot4\cdot4\cdot}{1\cdot3\cdot3\cdot5\cdot}\cdots = 2\pi.$$

6. Assume $c > 0$ and consider $\sum_{0\le k<n}\ln(k+c)$. We find

$$\ln\big(c(c+1)\cdots(c+n-1)\big) = (n+c)\ln(n+c) - c\ln c - n - \tfrac{1}{2}\ln c$$
$$+ \sum_{1<k\le m}\frac{B_k(-1)^k}{k(k-1)}\left(\frac{1}{(n+c)^{k-1}} - \frac{1}{c^{k-1}}\right) + R_{mn}.$$

Also

$$\ln(n-1)! = (n-\tfrac{1}{2})\ln n - n + \sigma + \sum_{1<k\le m}\frac{B_k(-1)^k}{k(k-1)}\left(\frac{1}{n^{k-1}}\right) - \frac{1}{m}\int_n^\infty \frac{B_m(\{x\})\,dx}{x^m}.$$

Now $\ln\Gamma_{n-1}(c) = c\ln(n-1) + \ln(n-1)! - \ln\big(c\cdots(c+n-1)\big)$; substituting and letting $n\to\infty$, we get

$$\ln\Gamma(c) = -c + (c-\tfrac{1}{2})\ln c + \sigma + \sum_{1<k\le m}\frac{B_k(-1)^k}{k(k-1)c^{k-1}} - \frac{1}{m}\int_0^\infty \frac{B_m(\{x\})\,dx}{(x+c)^m}.$$

This shows that $\Gamma(c+1) = ce^{\ln\Gamma(c)}$ has the same expansion we derived for $c!$.

7. $A\cdot n^{n^2/2+n/2+1/12}e^{-n^2/4}$ where A is a constant. (It is "Glaisher's constant" 1.2824271...) To obtain this result, apply Euler's summation formula to $\sum_{1\le k<n} k\ln k$. A more accurate formula is obtained if we multiply the above answer by

$$\exp\big(-B_4/2\cdot3\cdot4n^2 - \cdots - B_{2t}/(2t-2)(2t-1)(2t)n^{2t-2} + O(1/n^{2t})\big).$$

This formula makes it possible to calculate Glaisher's constant to six decimal places if we let $t = 3$, $n = 4$.

SECTION 1.2.11.3

1. Integrate by parts.

2. Substitute series for e^{-t} in integral.

3. See Eq. 1.2.9–11 and Ex. 1.2.6–48.

4. $1 + 1/u$ is bounded as a function of v, since it goes to zero as v goes from r to infinity. Replace it by M and the resulting integral is Me^{-rx}.

5. $f''(x) = f(x)\left(\frac{(n+\frac{1}{2})(n-\frac{1}{2})}{x^2} - \frac{(2n-1)}{x} + 1\right)$

has constant sign for $0 \le x \le n-1$, so $|R| \le \frac{1}{12}|f'(n-1)| + \frac{1}{12}\int_{n-1}^{n}|f''(x)|\,dx$.

6. It is $n^{n+\beta}\exp\left((n+\beta)(\alpha/n - \alpha^2/2n^2 + O(n^{-3})\right)$, etc.

7. The integrand as a power series in x^{-1} has the coefficient of x^{-n} as $O(u^{2n})$. After integration, terms in x^{-3} are $Cu^7/x^3 = O(x^{-5/4})$, etc. To get $O(x^{-2})$ in the answer, we can discard terms u^n/x^m with $4m - n \ge 9$ (cf. next exercise). Thus expanding $\exp(-u^2/2x)\exp(u^3/3x^2)\ldots$ leads ultimately to the answer

$$yx^{1/4} - \frac{y^3}{6}x^{-1/4} + \frac{y^5}{40}x^{-3/4} + \frac{y^4}{12}x^{-1} - \frac{y^7}{336}x^{-5/4} - \frac{y^6}{36}x^{-3/2} + \left(\frac{y^9}{3456} - \frac{y^5}{20}\right)x^{-7/4} + O(x^{-2}).$$

8. The integrand can be expanded into terms of the form $c_{mn}u^m/x^n$. These terms integrate into $O\big((m+1)r - n\big)$. [Note that if $r > \frac{1}{2}$ the series for $e^{-u^2/2x}$ integrates into series that diverge for large x, hence the assumption $r > \frac{1}{2}$ is necessary.] Multiplying two terms $(u^{m_1}/x^{n_1})(u^{m_2}/x^{n_2})$, if $(m_1+1)r - n_1 \le -s$, we have

$$\begin{aligned}(m_1+m_2+1)r - (n_1+n_2) &= (m_1+1)r - n_1 + (m_2+1)r - n_2 - r \\ &\le -s + r - r = -s.\end{aligned}$$

Therefore we may expand $\exp(-u^2/2x)\exp(u^3/3x^2)\ldots$ and discard all terms with $(m+1)r - n \le -s$ *before* multiplying together the factors; and all terms $\exp(u^p/px^{p-1})$ with $(p+1)r - p + 1 \le -s$ may be suppressed, i.e. those with $p > \lceil (s+2r)/(1-r)\rceil$.

9. We may assume $p \ne 1$, since $p = 1$ is given by Theorem A. We also assume $p \ne 0$ since that case is trivial.

Case 1. $p < 1$. Substitute $t = px(1-u)$ and then $v = -\ln(1-u) - pu$. We have $dv = \big((1-p+pu)/(1-u)\big)\,du$, so the transformation is monotone for $0 \le u \le 1$, and we obtain an integral of the form

$$\int_0^\infty xe^{-xv}\,dv\left(\frac{1-u}{1-p+pu}\right).$$

The parenthesized quantity is

$$\frac{1}{1-p}\left(1 - \frac{v}{(1-p)^2} + \cdots\right).$$

The answer is therefore

$$\frac{p}{1-p}(pe^{1-p})^x\frac{e^{-x}x^x}{\Gamma(x+1)}\left(1 - 1/(p-1)^2x + O(x^{-2})\right).$$

Case 2. $p > 1$. This is $1 - \int_{px}^{\infty}(\ )$. In the latter integral, substitute $t = px(1+u)$, then $v = pu - \ln(1+u)$, and proceed as in Case 1. The answer turns out to be the same formula as Case 1, plus one. Note that $pe^{1-p} < 1$ so $(pe^{1-p})^x$ is *very* small.

10. $\dfrac{p}{p-1}(pe^{1-p})^x e^{-x} x^x \left(1 - e^{-\nu} - \dfrac{e^{-\nu}(e^{\nu} - 1 - y - y^2/2)}{x(p-1)^2} + O(x^{-2})\right)$.

11. First, $xQ_x(n) + R_{1/x}(n) = n!(x/n)^n e^{n/x}$. (The case $x = -1$ is interesting here!) We get

$$Q_x(n) = \frac{1}{1-x} - \frac{x}{(1-x)^3 n} + O(n^{-2}),$$

$$R_x(n) = \frac{1}{1-x} - \frac{x}{(1-x)^3 n} + O(n^{-2}), \quad \text{if } x < 1;$$

$$Q_x(n) = \frac{1}{x} n! \left(\frac{x}{n}\right)^n e^{nx} - \frac{1}{x-1} + \frac{1}{(x-1)^3 n} + O(n^{-2}),$$

$$R_x(n) = n! \left(\frac{x}{n}\right)^n e^{nx} - \frac{1}{x-1} + \frac{x}{(x-1)^3 n} + O(n^{-2}), \quad \text{if } x > 1.$$

These formulas are quite easily verified when $|x| < 1$ and the relation between $Q_x(n)$ and $R_{1/x}(n)$ extends this to $|x| > 1$. The case $x = -1$ remains; this requires more delicate maneuverings with limits. For further details about the asymptotic expansion, and its connection with Stirling numbers of the second kind, see L. Carlitz, *Proc. Amer. Math. Soc.* **16** (1965), 248–252.

12. $\dfrac{1}{\sqrt{2}} \gamma(\frac{1}{2}, \frac{1}{2}x^2)$.

15. Expanding the integrand by the binomial theorem, we find the integral is $1 + Q(n)$.

16. Write $Q(k)$ as a sum, and interchange order of summation using Eq. 1.2.6–49.

SECTION 1.3.1

1. Four; each byte would then contain $3^4 = 81$ different values.

2. Five, since five bytes is always adequate but four is not.

3. (0:2); (3:3); (4:4); (5:5).

4. Presumably index register 4 contains a value greater than or equal to 2000, so that after indexing a valid memory address results.

5. "`DIV -80,3(0:5)`" or simply "`DIV -80,3`".

6. (a) Sets rA to | − | 5 | 1 | 200 | 15 |. (b) Sets rI2 to −200. (c) Sets rX to | + | 0 | 0 | 5 | 1 | ? |. (d) Undefined, since we can't load such a big value into an index register. (e) Sets rX to | − | 0 | 0 | 0 | 0 | 0 |.

7. Let the magnitude |rAX| before the operation be n, and let the magnitude of V be d. After the operation the magnitude of rA is $\lfloor n/d \rfloor$, and the magnitude of rX is $n \bmod d$. The sign of rX afterwards is the previous sign of rA; the sign of rA afterwards is + if the previous signs of rA and V were the same, and it is − otherwise.

8. rA ← | + | 0 | 617 | 0 | 1 |; rX ← | − | 0 | 0 | 0 | 1 | 0 |.

9. ADD, SUB, DIV, NUM, JOV, JNOV, INCA, DECA, INCX, DECX.

10. CMPA, CMP1, CMP2, CMP3, CMP4, CMP5, CMP6, CMPX. (Also FCMP, floating point.)

11. MOVE, LD1, LD1N, INC1, DEC1, ENT1, ENN1.

12. INC3 0,3.

13. "JOV 1000" makes no difference except time. "JNOV 1001" makes a different setting of rJ in most cases. "JNOV 1000" makes an extraordinary difference, since it may lock the computer in an infinite loop.

14. NOP with anything; ADD, SUB with F = (0:0); HLT (depending on how you interpret the statement of the exercise); any shift with address and index zero; MOVE with F = 0; JSJ with address equal to the location of the instruction plus one; any of the INC or DEC instructions with address and index zero; ENTi 0,i for $1 \le i \le 6$; SLC or SRC with address a multiple of 10.

15. 70; 80; 120. (record size times 5)

16. (a) STZ 0; ENT1 1; MOVE 0(49); MOVE 0(50). If the byte size were known to equal 100, only one MOVE instruction would have been necessary, but we are not allowed to make assumptions about the byte size. (b) Use 100 STZ's.

17. (a)
```
        STZ   0,2
        DEC2  1
        J2NN  3000
```

(b)
```
              STZ   0
              ENT1  1
              JMP   3004
(3003)  MOVE  0(63)
(3004)  DEC2  63
        J2NN  3003
        INC2  63
        ST2   3008(4:4)
(3008)  MOVE  0
```

(Using assembly language, a slightly faster program which uses "bytesize minus 1" instead of 63 could be written; see the following section.)

18. (If you have correctly followed the instructions, an overflow will occur on the ADD, with minus zero in register A afterward.) Answer: Overflow is set on, comparison is set EQUAL, rA is set to | − | 30 | 30 | 30 | 30 | 30 |, rX is set to | − | 31 | 30 | 30 | 30 | 30 |, rI1 is set to +3, and memory locations 0001, 0002 are set to zero.

19. $24u = (2+1+2+2+1+1+1+2+2+1+2+2+3+1+1)u$.

20.
```
(3990)  ENT1  0
        MOVE  3995      (standard F for MOVE is 1)
        MOVE  0(49)
(3993)  MOVE  0(50)
        JMP   3993
(3995)  HLT   0
```

21. (a) Not unless it can be set to zero by external means (see the "GO-button", Ex. 26), since a program can set rJ = N only by jumping from location $N - 1$.

(b)
```
                LDA   -1,4
                LDX   3004
                STX   -1,4
                JMP   -1,4
        (3004)  JMP   3005
        (3005)  STA   -1,4
```

22. *Minimum time:* If b is the byte size, the assumption that $|X^{13}| < b^5$ implies that $X^2 < b$, so X^2 can be contained in one byte. The following ingenious solution due to G. MacGinitie makes use of this fact.

```
(3000)  LDA   2000
        STA   3500(0:1)
        MUL   2000
        STX   3501(0:1)
        SRC   1
        MUL   3501
        STA   3502
        MUL   3500
        MUL   3502
        MUL   3502
        SLAX  2
        HLT   0
(3500)  NOP   0
(3501)  NOP   0
(3502)  NOP   0
```

	rA					rX			
X^2	0	0	0	0	0	0	0	0	0
X^4		0	0	0	0	0	0	0	0
X^4		0	0	0	0	0	0	0	0
X^5			0	0	0	0	0	0	0
X^9					0	0	0	0	0
0	0	X^{13}					0	0	0
X^{13}					0	0	0	0	0

space = 15; time = $62u$.

At least five multiplications are "necessary" (see Chapter 4); however, see the much better solution on the following page.

Minimum space:

```
(3000)  ENT4  12
        LDA   2000
(3002)  MUL   2000
        SLAX  5
        DEC4  1
        J4P   3002
        HLT   0
```

space = 7; time = $171u$.

True minimum time: As R. W. Floyd points out, the conditions imply $|X| \leq 6$, so the minimum execution time is achieved by referring to a table:

```
(3000)   LD1   2000
         LDA   3500,1
         HLT   0
(3494)   (−6)[13]
(3495)   (−5)[13]
   ⋮
(3506)   (+6)[13]
```

space = 16, time = $4u$.

23. The following solution by R. D. Dixon appears to satisfy all the conditions:

```
(3000)   ENT1   4
(3001)   LDA    200
         SRA    0,1
         SRAX   1
         DEC1   1
         J1NN   3001
         SLAX   5
         HLT    0
```

24. (a) `DIV 3500`, where 3500 = | + | 1 | 0 | 0 | 0 | 0 |.

(b) `SRC 4; SRA 1; SLC 5`.

25. Some ideas: (a) Obvious things like faster memory, more input-output devices. (b) The I field could be used for J-register indexing, and/or multiple indexing (specify two different index registers), and/or "indirect addressing" (see Exs. 2.2.2–3, 4, 5). (c) Index registers and J register can be extended to a full five bytes; this means locations with higher addresses can be referred to only by indexing, but this is not so intolerable if multiple indexing is available as in (b). (d) An interrupt capability can be added (when certain conditions occur, all registers are stored in special locations—say locations $-1, -2, \ldots$—and a jump is made to a control program which later restarts the original program by using a new operation code, see Ex. 1.4.4–18). (e) A "real time clock" could be added, in a negative memory address. (f) "Logical operations" could be added to binary versions of `MIX` (see Ex. 2.5–28 and Chapter 7). (g) An "execute" command, meaning to perform the instruction at location M, could be another variant of C = 5.

26. The following routine is the shortest found so far for which the transfer card does not depend on the byte size. It is tempting to use a (2:5) field to get at cols. 7–10 of the card, but this cannot be done since $2 \cdot 8 + 5 = 21$. Since this routine requires only 28 instructions, it can be adapted for paper tape.

To make the program easier to follow, it is presented here in symbolic language, anticipating the following section of the text.

The transfer card has the format `TRANSOnnnn` in columns 1–10, where nnnn is the place to jump to start execution.

```
                                                          characters
      BUFF    EQU   28          Buffer area is 28–43      punched on card:
              ORIG  0
00    TEMP1   IN    16(16)      Read in second card.          0 06
01    READ    IN    BUFF(16)    Read next card.               Y 06
02            LD1   0(0:0)      [ENT1 0]                        I
03            ENTA  0                                         B=
04            JBUS  *(16)       Wait for read to finish       D 04
05            LD2N  BUFF+1(1:1) — (count + 30)                Z IQ
06            STZ   BUFF+1(1:1) Clear (1:1) so (2:5)          Z I3
07            LDX   BUFF+1        can be used.                Z EN
08    TEMP    NUM   0                                         E
09            STA   TEMP1       Starting location             EU
10            ENTA  30,2        — count                       OBB=
11    LOOP    STA   TEMP(1:1)                                 H IU
12            LD3   TEMP1                                     EJ
13            JAZ   0,3         Transfer card                 CA.
14            ENTA  1,3         Increase TEMP1.               ACB=
15            STA   TEMP1                                     EU
```

```
16            LDA   BUFF+3,1(5:5)                           1A-H
17            DECA  25                                      V A=
18            STA   0,3         Store sign.                   CEU
19            LDA   BUFF+2,1                                  OAEH
20            LDX   BUFF+3,1                                  1AEN
21            NUM                                             E
22            STA   0,3(1:5)    Store magnitude.              CLU
23            MOVE  0,1(2)      [INC1 2!]                     ABG
24            LDA   TEMP(1:1)                                 H IH
25            DECA  1           Decrease count.               A A=
26            JAP   LOOP                                      J B.
27            JMP   READ        Ready for new card            A  9
```

SECTION 1.3.2

1. ENTX 1000; STX X.

2. The STJ instruction in line 03 resets this address. (It is conventional to denote the address of such instructions by "*", both because it is simple to write, and because it provides a recognizable test of an error condition in a program, where a subroutine has not been entered properly because of some oversight.)

3. Read in 100 words from tape unit zero; exchange the maximum of these with the last one; exchange the maximum of the remaining 99 with the last of those; etc. Eventually the 100 words will become completely sorted into ascending sequence and the result is then written onto tape unit one.

4. Nonzero locations:

3000:	+	0000	00	18	35
3001:	+	2051	00	05	09
3002:	+	2050	00	05	10
3003:	+	0001	00	00	49
3004:	+	0499	01	05	26
3005:	+	3016	00	01	41
3006:	+	0002	00	00	50
3007:	+	0002	00	02	51
3008:	+	0000	00	02	48
3009:	+	0000	02	02	55
3010:	−	0001	03	05	04
3011:	+	3006	00	01	47
3012:	−	0001	03	05	56
3013:	+	0001	00	00	51
3014:	+	3008	00	06	39
3015:	+	3003	00	00	39
3016:	+	1995	00	18	37
3017:	+	2035	00	02	52
3018:	−	0050	00	02	53
3019:	+	0501	00	00	53
3020:	−	0001	05	05	08

3021:	+	0000	00	01	05
3022:	+	0000	04	12	31
3023:	+	0001	00	01	52
3024:	+	0050	00	01	53
3025:	+	3020	00	02	45
3026:	+	0000	04	18	37
3027:	+	0024	04	05	12
3028:	+	3019	00	00	45
3029:	+	0000	00	02	05
0000:	+				2
1995:	+	06	09	19	22	23
1996:	+	00	06	09	25	05
1997:	+	00	08	24	15	04
1998:	+	19	05	04	00	17
1999:	+	19	09	14	05	22
2024:	+				2035
2049:	+				2010
2050:	+				3
2051:	−				499

(the latter two may be interchanged, with corresponding changes to 3001 and 3002)

6. (a) If n is not prime, by definition n has a divisor d with $1 < d < n$. If $d > \sqrt{n}$, n/d is a divisor with $1 < n/d < \sqrt{n}$. (b) If N is not prime, N has a *prime* divisor d with $1 < d \leq \sqrt{n}$. The algorithm has verified that N has no prime divisors $\leq p =$ PRIME[K]; also $\mathtt{N} = p\mathtt{Q} + \mathtt{R} < p\mathtt{Q} + p \leq p^2 + p < (p+1)^2$. Any prime divisor of N is therefore $> p + 1 > \sqrt{\mathtt{N}}$.

Strictly speaking, we must also prove that there will be a sufficiently large prime less than N when N is prime, so that when $\mathtt{K} \leftarrow \mathtt{K} + 1$ we know PRIME[K] has already been stored. This follows from "Bertrand's postulate," i.e., if p is prime there is a larger prime less than $2p$.

7. (a) It refers to the location of line 29. (b) The program would then fail; line 14 would refer to line 15 instead of line 25; line 24 would refer to line 15 instead of line 12.

8. Prints 100 lines. If the 12000 characters on these lines were arranged end to end, they would reach quite far and would consist of five blanks followed by five A's

followed by ten blanks followed by five A's followed by fifteen blanks . . . followed by $5k$ blanks followed by five A's followed by $5(k+1)$ blanks . . . until 12000 characters have been printed. The second-last line ends with AAAAA and 35 blanks; the final line is all blank. The total effect is one of OP art, as in OP-code.

9. In the table, the (4:4) field is the maximum F setting; (1:2) is the location of a checking routine.

```
B       EQU   1(4:4)
BMAX    EQU   B-1
TABLE   NOP   GOOD(BMAX)
        ADD   FLOAT(5:5)
        SUB   FLOAT(5:5)
        MUL   FLOAT(5:5)
        DIV   FLOAT(5:5)
        HLT   GOOD
        SRC   GOOD
        MOVE  MEMORY(BMAX)
        LDA   FIELD(5:5)
        LD1   FIELD(5:5)
...
        STZ   FIELD(5:5)
        JBUS  MEMORY(19)
        IOC   GOOD(19)
        IN    MEMORY(19)
        OUT   MEMORY(19)
        JRED  MEMORY(19)
        JLE   MEMORY
        JANP  MEMORY
...
        JXNP  MEMORY
        ENNA  GOOD
...
        ENNX  GOOD
        CMPA  FLOAT(5:5)
        CMP1  FIELD(5:5)
...
        CMPX  FIELD(5:5)
```

```
BEGIN   LDA   INST
        CMPA  VALID(3:3)
        JG    BAD                    I field > 6?
        LD1   INST(5:5)
        DEC1  64                     C field > 64?
        J1NN  BAD
        CMPA  TABLE+64,1(4:4)        F field > F max?
        JG    BAD
        LD1   TABLE+64,1(1:2)        Jump to special
        JMP   0,1                      routine.
FLOAT   CMPA  VALID(4:4)             F = 6 allowed on
        JE    GOOD                     arithmetic op
FIELD   ENTA  0
        LDX   INST(4:4)              This is a tricky
        DIV   =9=                      way to check
        STX   *+1(0:2)                 for a valid
        INCA  0                        partial field.
        CMPA  =5=
        JG    BAD
MEMORY  LDX   INST(3:3)
        JXNZ  GOOD                   If I = 0,
        LDX   INST(0:2)              ensure the
        JXN   BAD                    address is a
        CMPX  =3999=                 valid memory
        JLE   GOOD                   location.
        JMP   BAD
VALID   CMPX  3999,6(6)
```

10. The catch to this problem is that there may be several places in a row (column) where the minimum (maximum) occurs, and each is a potential saddle point.

Solution 1: In this solution we make a list of all columns in which the row minimum occurs, then check for a column maximum for each column in the list.

rX ≡ current max or min; rI1 traces through the matrix (goes from 72 down to zero unless a saddle point is found); rI2 ≡ column index of rI1; rI3 ≡ size of list of minima. Notice that in all cases the terminating condition for a loop is that an index register is ≤ 0.

```
* SOLUTION 1
A10     EQU   1008
LIST    EQU   1000
START   ENT1  72        Begin at lower right column.
ROWMIN  ENT2  8         Now rI1 is at 8th column of row.
2H      LDX   A10,1     Candidate for row minimum
        ENT3  0         List empty
```

```
4H      INC3  1
        ST2   LIST,3      Put column index in list.
1H      DEC1  1           Go left one.
        DEC2  1
        J2Z   COLMAX      Done with row?
3H      CMPX  A10,1
        JL    1B          Is rX still minimum?
        JG    2B          New minimum?
        JMP   4B          Another appearance of minimum.
COLMAX  LD2   LIST,3      Get column from list.
        INC2  64
1H      CMPX  A10,2
        JL    NO          Is row min < col element?
        DEC2  8
        J2P   1B          Done with column?
YES     INC1  A10+8,2     Yes; rI1 ← address of saddle.
        HLT
NO      DEC3  1           Is list empty?
        J3P   COLMAX      No; try again.
        J1P   ROWMIN      Have all rows been tried?
        HLT               Yes; rI1 = 0, no saddle. ▮
```

Solution 2: The introduction of Mathematics gives a different algorithm.

Theorem. *Let* $R(i) = \min_j a_{ij}$, $C(j) = \max_i a_{ij}$. *The element* $a_{i_0j_0}$ *is a saddle point if and only if* $R(i_0) = \max_i R(i) = C(j_0) = \min_j C(j)$.

Proof: If $a_{i_0j_0}$ is a saddle point, then for any fixed i, $R(i_0) = C(j_0) \geq a_{ij_0} \geq R(i)$; so $R(i_0) = \max_i R(i)$. Similarly $C(j_0) = \min_j C(j)$. Conversely, assuming the given condition, $R(i_0) \leq a_{i_0j_0} \leq C(j_0) = R(i_0)$ implies $a_{i_0j_0} = R(i_0)$ so we have a saddle point. ▮

(It may be of interest to note that we always have the inequality

$$\max_i \min_j a_{ij} = \min_j a_{i_0j} \leq \min_j \max_i a_{ij}, \text{ for some } i_0;$$

so there is no saddle point iff $\max R(i) < \min C(j)$, i.e. all the R's are less than all the C's.)

A program based on this theorem finds the smallest column maximum and then searches for an equal row minimum. (Phase 1: rI1 is col index; rI2 runs through matrix. Phase 2: rI1 is possible answer; rI2 runs through matrix; rI3 is row index; rI4 is column index.)

```
* SOLUTION 2
A10     EQU   1008
CMAX    EQU   1000
PHASE1  ENT1  8           Start at column 8.
3H      ENT2  64,1
        JMP   2F
1H      CMPX  A10,2       rX still minimum?
        JGE   *+2
```

```
2H      LDX   A10,2      New maximum in column
        DEC2  8
        J2P   1B
        STX   CMAX+8,2   Store column maximum.
        J2Z   1F         First time?
        CMPA  A10,2      rA still min max?
        JLE   *+2
1H      LDA   A10,2
        DEC1  1          Move left a column.
PHASE2  ENT3  64         At this point rA = min C(j)
3H      ENT2  8,3        Prepare to search a row.
        ENT4  8
1H      CMPA  A10,2      Is a[i,j] ≥ min C(j)?
        JG    NO         No saddle in this row
        JL    2F
        CMPA  CMAX,4     a[i,j] = C(j)?
        JNE   2F
        ENT1  A10,2      Possible saddle point
2H      DEC4  1          Move left in row.
        DEC2  1
        J4P   1B
        HLT              Saddle point found
NO      DEC3  8
        ENT1  0          Try another row.
        J3P   3B
        HLT              rI1 = 0; no saddle. ∎
```

We leave it to the reader to invent a still better solution in which "phase 1" records all possible rows which are candidates for the row search in "phase 2". It is not necessary to search all rows, just those i_0 for which there exists j_0 with $a_{i_0 j_0} = C(j_0) = \min_j C(j)$. Usually this is only one row.

In some trial runs with elements selected at random from {0,1,2,3,4}, solution 1 required approximately $730u$ to run, while solution 2 took about $540u$. Given a matrix of all zeroes, solution 1 found a saddle point in $137u$, solution 2 in $524u$.

11. Assume an $m \times n$ matrix. (a) By the theorem in the answer to Ex. 10, all saddle points of a matrix have the same value, so (under our assumption of distinct elements) there is at most one saddle point. By symmetry the desired probability is mn times the probability that a_{11} is a saddle point. This latter is $1/(mn)!$ times the number of permutations with $a_{12} > a_{11}, \ldots, a_{1n} > a_{11}, a_{11} > a_{21}, \ldots, a_{11} > a_{m1}$; this is $1/(m+n-1)!$ times the number of permutations of $m+n-1$ elements in which the first is greater than the next $(m-1)$ and less than the remaining $(n-1)$, namely $(m-1)!(n-1)!$. The answer is therefore

$$mn(m-1)!(n-1)!/(m+n-1)! = (m+n)\Big/\binom{m+n}{n}.$$

In our case this is $17/\binom{17}{8}$, only one chance in 1430. (b) Under the second assumption, an entirely different method must be used since there can be multiple saddle points; in fact either a whole row or whole column must consist entirely of saddle points. The

probability equals the probability that there is a saddle point with value zero plus the probability that there is a saddle point with value one. The former is the probability that there is at least one column of zeroes; the latter is the probability that there is at least one row of ones. The answer is $(1 - (1 - 2^{-m})^n) + (1 - (1 - 2^{-n})^m)$; in our case, 92474479623403623 1/18446744073709551616, about 1 in 19.9. An approximate answer is $n2^{-m} + m2^{-n}$.

13.

```
*CRYPTANALYST PROBLEM (CLASSIFIED)
UNIT    EQU   19          Input unit number
SIZE    EQU   14          Input record size
TABLE   EQU   1000        Table of counts
        ORIG  TABLE         (initially zero
        CON   -1            except entries for
        ORIG  TABLE+46      blank space and
        CON   -1            asterisk)
        ORIG  2000
BUF1    ORIG  *+SIZE      First buffer area
        CON   -1          "Sentinel" at end of buffer
        CON   *+1         Each buffer refers to other
BUF2    ORIG  *+SIZE      Second buffer
        CON   -1          "Sentinel"
        CON   BUF1        Reference to first buffer
BEGIN   IN    BUF1(UNIT)  Input first record.
        ENT6  BUF2
1H      IN    0,6(UNIT)   Input next record.
        LD6   SIZE+1,6    During this input, prepare
        ENT5  0,6           to process previous one.
2H      LDX   0,5         Five chars → rX.
        JXN   1B          End of record?
1H      SLAX  1
        STA   *+1(2:2)    Next char → rI1.      } main
        ENT1  0                                 } loop,
        LDA   TABLE,1                           } should
        JAN   2F          Is character special? } run as
        INCA  1           Update table entry.   } fast as
        STA   TABLE,1                           } possible
1H      JXP   1B          Any non-blanks in rX? }
        INC5  1
        JMP   2B
2H      J1Z   1B          Skip over a blank.
        ENT1  1           Asterisk detected.
2H      LDA   TABLE,1
        JANP  1F          Skip zero answers.
        CHAR              Convert to decimal.
        JBUS  *(19)       Wait for typewriter complete.
        ST1   CHAR(1:1)
        STA   CHAR(4:5)
```

```
         STX   FREQ
         OUT   ANS(19)        Type one answer.
1H       CMP1  =60=
         INC1  1              Up to 60 character
         JL    2B               codes counted
         HLT
ANS      ALF                  Output buffer
         ALF
CHAR     ALF   C  NN
FREQ     ALF   NNNNN
         ORIG  ANS+14         Rest of buffer is blank
         END   BEGIN          Literal constant =60= here. ▮
```

For this problem, buffering of *output* is not desirable since it could save at most $7u$ of time per line output, and this is quite insignificant compared to the time required to output the line itself. For information about letter frequencies, see Charles P. Bourne and Donald F. Ford, "A study of the statistics of letters in English words," *Information and Control* **4** (1961), 48–67.

14. To make the problem more challenging, the following solution uses as much *trickery* as possible, in order to reduce execution time. Can the reader squeeze out any more microseconds?

```
         *DATE OF EASTER
EASTER   STJ   EASTX
         STX   Y
         ENTA  0              E1.
         DIV   =19=
         INCX  1
         STX   G(0:2)
         LDA   Y              E2.
         MUL   =1//100+1=     (see
         INCA  1                below)
         STA   C(1:4)
         MUL   =3//4+1=       E3.
         STA   XPLUS12(0:2)
         LDA   =8(1:1)=
         MUL   C              rA = 8C
         INCA  680            680 = 5 + 27 · 25
         MUL   =1//25+1=      rA = Z + 32
XPLUS12  DECA  0
         STA   1F(0:2)        Z + 20 − X
         LDA   Y              E4.
         MUL   =1//4+1=
         ADD   Y
         SUB   XPLUS12(0:2)
         INCA  5
         STA   DPLUS3
G        ENTA  0              E5.
```

```
          MUL   =11=
1H        INCX  0
          DIV   =30=
          JXNN  *+2              see Ex. 15
          INCX  30
          CMPX  =24=
          JE    1F
          CMPX  =25=
          JNE   2F
          LDA   G(0:2)
          DECA  11
          JANP  2F
1H        INCX  1
2H        DECX  20               E6. (24 — N)
          CMPX  =3=
          JLE   *+2
          DECX  30
          STX   N(0:2)
          LDAN  N(0:2)           E7.
          ADD   DPLUS3
          SRAX  5
          DIV   =7=
          SLAX  5
N         INCA  0                31 — N
          JANN  1F               E8.
          CHAR
          LDA   APRIL
          JMP   2F
1H        DECA  31
          CHAR
          LDA   MARCH
2H        JBUS  *(18)
          STA   MONTH
          STX   DAY(1:2)
          LDA   Y
          CHAR
          STX   YEAR
          OUT   ANS(18)          Print
EASTX     JMP   *
MARCH     ALF   MARCH
APRIL     ALF   APRIL
ANS       ALF
DAY       ALF   DD
MONTH     ALF   MMMMM
          ALF   ,
YEAR      ALF   YYYYY
C         CON   0                C times byte size
          ORIG  *+20
```

```
BEGIN   LDX   =1950=          "driver"
        JMP   EASTER             routine,
        LDX   Y                  uses the
        INCX  1                  above
        CMPX  =2000=             subroutine
        JLE   EASTER+1
        HLT
        END   BEGIN              ▮
```

A rigorous justification for the change from division to multiplication in several places can be based on the fact that the number in rA is not too large. (Cf. Section 12.4.1.) The program works with all byte sizes.

16. Work with scaled numbers, $R_n = 10^n r_n$. $R_n(1/m) = R$ iff $10^n/(R + \frac{1}{2}) < m \le 10^n/(R - \frac{1}{2})$; thus we find $m_h = \lfloor 2 \cdot 10^n/(2R - 1) \rfloor$.

```
*SUM OF HARMONIC SERIES
BUF     ORIG  *+24
START   ENT2  0
        ENT1  3          [5 — n]
        ENTA  20
OUTER   MUL   =10=
        STX   CONST      [2 · 10^n]
        DIV   =2=
        ENTX  2
        JMP   1F
INNER   STA   R
        ADD   R
        DECA  1
        STA   TEMP       [2R — 1]
        LDX   CONST
        ENTA  0
        DIV   TEMP
        INCA  1
        STA   TEMP       [m_h + 1]
        SUB   M
        MUL   R
        SLAX  5
        ADD   S
        LDX   TEMP
1H      STA   S          Partial sum
        STX   M
        LDA   M
        ADD   M
        STA   TEMP
        LDA   CONST
        ADD   M          Compute
        SRAX  5          ⌊(2 · 10^n + m)/2m⌋.
        DIV   TEMP
```

```
        JAP    INNER    R > 0?
        LDA    S        Answer
        CHAR
        SLAX   0,1      Neat formatting
        SLA    1
        INCA   40       Decimal point
        STA    BUF,2
        STX    BUF+1,2
        INC2   3
        DEC1   1
        LDA    CONST
        J1NN   OUTER
        OUT    BUF(18)
        HLT
        END    START    ∎
```

The output is

```
    0006.16        0008.449        0010.7509        0013.05362
```

in $65595u$ plus output time.

18.

```
FAREY   STJ    9F       Assume rI1 contains n
        ENTA   0        and assume n > 1.
        STA    X        x_0 ← 0.
        ENTX   1
        STX    Y        y_0 ← 1.
        STX    X+1      x_1 ← 1.
        ST1    Y+1      y_1 ← n.
        ENT2   2        k ← 2.
1H      LDX    Y-2,2
        INCX   0,1
        ENTA   0
        DIV    Y-1,2
        STA    TEMP     ⌊(y_{k-2} + n)/y_{k-1}⌋
        MUL    Y-1,2
        SLAX   5
        SUB    Y-2,2
        STA    Y,2      y_k
        LDA    TEMP
        MUL    X-1,2
        SLAX   5
        SUB    X-2,2
        STA    X,2      x_k
        CMPA   Y,2      Test if x_k < y_k.
        INC2   1        k ← k + 1.
        JL     1B       If so, continue.
9H      JMP    *        Exit from subroutine. ∎
```

19. (a) Induction. (b) Let $k \geq 0$ and let the righthand side of (*) be denoted by X, Y. By part (a) we have $\gcd(X, Y) = 1$ and $X/Y > x_{k+1}/y_{k+1}$. So if

$$X/Y \neq x_{k+2}/y_{k+2},$$

we have, by definition, $X/Y > x_{k+2}/y_{k+2}$. But this implies

$$\frac{1}{Yy_{k+1}} = \frac{Xy_{k+1} - Yx_{k+1}}{Yy_{k+1}} = \frac{X}{Y} - \frac{x_{k+1}}{y_{k+1}}$$
$$= \left(\frac{X}{Y} - \frac{x_{k+2}}{y_{k+2}}\right) + \left(\frac{x_{k+2}}{y_{k+2}} - \frac{x_{k+1}}{y_{k+1}}\right) \geq \frac{1}{Yy_{k+2}} + \frac{1}{y_{k+1}y_{k+2}}$$
$$= \frac{Y + y_{k+1}}{Yy_{k+1}y_{k+2}} > \frac{n}{Yy_{k+1}y_{k+2}} \geq \frac{1}{Yy_{k+1}}.$$

For more of the interesting properties of the Farey series, and its history, see G. H. Hardy and E. M. Wright, *The Theory of Numbers*, Oxford, chapter 3.

20.

```
*TRAFFIC SIGNAL PROBLEM
BSIZE    EQU   1(4:4)          Bytesize
2BSIZE   EQU   2(4:4)          Twice bytesize
DELAY    STJ   1F               If rA contains n,
         DECA  6                  this subroutine
         DECA  2                  waits max (n, 7)u
         JAP   *-1                exactly, not including
         JAN   *+2                the jump
         NOP                      to the subroutine
1H       JMP   *
FLASH    STJ   2F          4    This subroutine
         ENT2  8           5      flashes the
         LDA   =49991=     7      appropriate DON'T
1H       JMP   DELAY       8      WALK light
         DECX  0,1         9    Turn light off.
         LDA   =49996=     2
         JMP   DELAY       3
         INCX  0,1         4    "DON'T WALK"
         DEC2  1           1
         J2Z   1F          2    Repeat eight times.
         LDA   =49993=     4
         JMP   1B          5
1H       LDA   =399992=         Set amber 2u after exit.
         JMP   DELAY       5
2H       JMP   *           6
WAIT     JNOV  *                Del Mar green until tripped
TRIP     INCX  BSIZE            DON'T WALK on Del Mar
         ENT1  2BSIZE
         JMP   FLASH
         LDX   BAMBER          Amber on boulevard
         LDA   =799995=
         JMP   DELAY       3    Wait 8 seconds
         LDX   AGREEN      5    Green for avenue
         LDA   =799996=
         JMP   DELAY            Wait 8 seconds.
         INCX  1                DON'T WALK on Berk'ly
```

```
            ENT1  2
            JMP   FLASH          Do flash cycle.
            LDX   AAMBER         Amber on avenue
            JOV   *+1            Cancel redundant trip.
            LDA   =499994=
            JMP   DELAY          Wait 5 seconds.
BEGIN       LDX   BGREEN         Green on boulevard
            LDA   =1799994=
            JMP   DELAY          Wait at least 18
            JMP   WAIT             seconds.
AGREEN      ALF   CABA           Green for avenue
AAMBER      ALF   CBBB           Amber for avenue
BGREEN      ALF   ACAB           Green for boulevard
BAMBER      ALF   BCBB           Amber for boulevard
            END   BEGIN          ∎
```

22.

```
*JOSEPHUS PROBLEM
N    EQU   24
M    EQU   11
X    ORIG  *+N
OH   ENT1  N-1          1          Set each cell to
     STZ   X+N-1        1            number of next man
     ST1   X-1,1        N - 1        in the sequence.
     DEC1  1            N - 1
     J1P   *-2          N - 1
     ENT1  0            1
     ENTA  1            1
1H   ENT2  M-2          N - 1      (assume M > 2)
     LD1   X,1          (M - 2)(N - 1)  Count around
     DEC2  1            (M - 2)(N - 1)    the circle.
     J2P   *-2          (M - 2)(N - 1)
     LD2   X,1          N - 1      rI1 ≡ lucky man
     LD3   X,2          N - 1      rI2 ≡ doomed man
     CHAR               N - 1      rI3 ≡ next man
     STX   X,2(4:5)     N - 1      Store execution number.
     NUM                N - 1
     INCA  1            N - 1
     ST3   X,1          N - 1      Take man from circle.
     ENT1  0,3          N - 1
     CMPA  =N=          N - 1
     JL    1B           N - 1
     CHAR               1          One man left
     STX   X,1(4:5)     1          (he is clobbered too).
     OUT   X(18)        1          Print answer.
     HLT                1
     END   0B                      ∎
```

The last man is in position 11. The total time before output is $(4(N-1)(M+3)+8)u$. Is there a better method?

1. (1 2 4)(3 6 5).

2. $a \leftrightarrow c$, $c \leftrightarrow f$; $b \leftrightarrow d$. The generalization to arbitrary permutations is clear.

3. $\begin{pmatrix} a & b & c & d & e & f \\ d & b & f & c & a & e \end{pmatrix}$.

4. $(a\ d\ c\ f\ e)$.

5. 12. (Cf. Ex. 20.)

6. The total time is increased by $4u$ for every blank word with the preceding nonblank word a "(", plus $5u$ for every blank word with the preceding nonblank word a name. Initial blanks and blanks between cycles do not affect the execution time. The position of blanks has no effect whatever on Program B.

7. $X = 2$, $Y = 29$, $M = 5$, $N = 7$, $U = 3$, $V = 1$. Total, by Eq. (18), $2165u$.

8. Yes; we would then keep the inverse of the permutation, i.e. x_i goes to x_j iff $T[j] = i$. (The final cycle form would then be constructed from the T table from right to left.)

9. No. For example, given (6) as input, Program A will produce "(ADG)(CEB)" as output, while Program B produces "(CEB)(DGA)". The answers are equivalent but not identical, due to the non-uniqueness of the cycle notation. The first element chosen for a cycle is the (a) leftmost available name, or (b) last available distinct name to be encountered from right to left, with Program A or B, respectively.

10. (1) Kirchhoff's law yields $A = 1 + C - D$; $B = A + J + P - 1$; $C = B - (P - L)$; $E = D - L$; $F = E + G - H$; $G = F - (J - H)$; $Q = Z$; $W = S$. (2) Interpretations: B = number of words of input $= 16X - 1$; C = number of nonblank words $= Y$; $D = C - M$; $E = D - M$; G = number of unsuccessful comparisons in names table search; $H = N$; $K = M$; $Q = N$; $R = U$; $S = R - V$; $T = N - V$ since each other name gets tagged. (3) Summing up, we have $(5G + 17Y + 80X + 18N - 21M + 10U - 17V)u$, which is somewhat better than Program A since G is certainly less than $16NX$. The time in the stated case is $962u$.

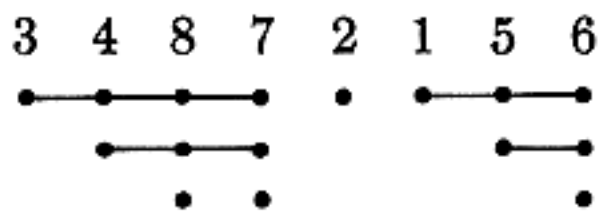
11. "Reflect" it; e.g., the inverse of $(acf)(bd)$ is $(db)(fca)$.

12. (a) The value in cell $L + mn - 1$ is fixed by the transposition, so we may omit it from consideration. Otherwise if $x = n(i - 1) + (j - 1) < mn - 1$, the value in $L + x$ should go to cell $L + (mx) \bmod N = L + \big(mn(i - 1) + m(j - 1)\big) \bmod N = L + m(j - 1) + (i - 1)$, since $mn \equiv 1 (\text{modulo } N)$ and $0 \leq m(j - 1) + (i - 1) < N$. (b) If one bit in each memory cell is available (e.g. the sign, or the least significant bit of a floating-point value), we can "tag" elements as we move them, using an algorithm like algorithm I. Thus, (a) set $x \leftarrow N - 1$; (b) if CONTENTS$(L + x)$ has been tagged, go to (f), otherwise tag it and set $y \leftarrow$ CONTENTS$(L + x)$; (c) set $x \leftarrow (mx) \bmod N$; (d) exchange $y \leftrightarrow$ CONTENTS$(L + x)$; (e) if y is untagged, tag it and return to (d); (f) decrease x by 1 and if $x > 0$ return to (b). Reference: *JACM* **5** (1958), 383–384. If there is no room for a tag bit, tag bits can perhaps be kept in an auxiliary table, or else a list of representatives of all non-singleton cycles can be used: For each divisor d of N, we can transpose those elements which are multiples of d separately, since m is prime to N. The length of the cycle containing x, when gcd $(x, N) = d$, is the smallest integer $r > 0$ such that $m^r \equiv 1$ (modulo N/d). For each d, we want to find $\varphi(N/d)/r$ representatives, one from each of these cycles. Some number-theoretic methods are available for this purpose, but they are not simple enough to be really

satisfactory; if too much space is not required we are probably best off storing a list of cycle representatives somewhere in memory by simply using a tagging method to compute these representatives when the program is first loaded. Reference: Gordon Pall and Esther Seiden, *Math. Comp.* **14** (1960), 189–192. See also J. Boothroyd, *CACM* **10** (1967), 292–3.

13. Show by induction that, at the beginning of step J2, $X[i] = +j$ if and only if $j > m$ and j goes to i under π; $X[i] = -j$ iff i goes to j under π^{k+1}, where k is the smallest non-negative integer such that π^k takes i into a number $\leq m$.

14. Writing the *inverse* of the given permutation in canonical cycle form and dropping parentheses, the quantity $A - N$ is the sum of the number of consecutive elements greater than a given element and immediately to its right. For example, if the original permutation is (165)(3784), the canonical form of the inverse is (3487)(2)(156); set up the array



and the quantity A is the number of "dots," 16. The number of dots below the k-th element is the number of right-to-left minima in the first k elements (i.e. there are 3 dots below 7 in the above, since there are 3 right-to-left minima in 3487). Hence the average is $H_1 + H_2 + \cdots + H_n = (n+1)H_n - n$.

15. If the first character of the linear representation is 1, the last character of the canonical representation is 1. If the first character of the linear representation is $m > 1$, then "$\ldots 1m \ldots$" appears in the canonical representation. So the only solution is the permutation of order 1.

16. 1324,4231,3214,4213,2143,3412,2413,1243,3421,1324, ...

17. (a) The probability that the cycle is an m-cycle is $n!/m$ divided by $n!H_n$, so $p_m = 1/mH_n$. The average length is $p_1 + 2p_2 + 3p_3 + \cdots = \sum_{1\leq m\leq n} (m/mH_n) = n/H_n$. (b) Since the total number of m-cycles is $n!/m$, the total number of appearances of elements in m-cycles is $n!$. Each element appears as often as any other, by symmetry, so k appears $n!/n$ times in m-cycles. In *this* case, therefore, $p_m = 1/n$ for all k and m; the average is

$$\sum_{1\leq m\leq n} m/n = (n+1)/2.$$

18. See Ex. 22(e).

19. $|P_{n0} - n!/e| = 1/(n+1)! - 1/(n+2)! + \cdots$, an alternating series of decreasing magnitudes, which is less than $1/(n+1)! \leq \frac{1}{2}$.

20. Each m-cycle can be independently written in m ways; there are $\alpha_1 + \alpha_2 + \cdots$ cycles in all, which can be permuted among one another; and so the answer is

$$(\alpha_1 + \alpha_2 + \cdots)!1^{\alpha_1} \cdot 2^{\alpha_2} \cdot 3^{\alpha_3} \cdots .$$

21. $1/(\alpha_1!1^{\alpha_1}\alpha_2!2^{\alpha_2}\cdots)$ if $n = \alpha_1 + 2\alpha_2 + \cdots$; zero otherwise.

Proof: Write out α_1 1-cycles, α_2 2-cycles, etc., in a row, with empty positions; for example if $\alpha_1 = 1$, $\alpha_2 = 2$, $\alpha_3 = \alpha_4 = \cdots = 0$, we would have "(–) (– –) (– –)".

Fill the empty positions in all $n!$ possible ways; we obtain each permutation of the desired form exactly $\alpha_1!1^{\alpha_1}\alpha_2!2^{\alpha_2}\ldots$ times.

22. (a) If $k_1 + 2k_2 + \cdots = n$, the probability in (ii) is $\prod_{j\geq 0} f(w, j, k_j)$ which is assumed to equal $(1 - w)w^n/k_1!1^{k_1}k_2!2^{k_2}\ldots$; hence

$$\Big(\prod_{j\geq 0} f(w, j, k_j)\Big)^{-1} \prod_{j\geq 0} f(w, j, k_j + \delta_{jm}) = \frac{f(w, m, k_m + 1)}{f(w, m, k_m)} = \frac{w^m}{m(k_m + 1)}.$$

Therefore by induction

$$f(w, m, k) = \frac{1}{k!}\left(\frac{w^m}{m}\right)^k f(w, m, 0).$$

Condition (i) now implies that

$$f(w, m, k) = \frac{1}{k!}\left(\frac{w^m}{m}\right)^k e^{-w^m/m}.$$

[*Note:* Hence α_m is chosen with a "Poisson" distribution, see Ex. 1.2.10–16.]

$$\text{(b)} \quad \sum_{\substack{k_1+2k_2+\ldots=n \\ k_1,k_2,\ldots\geq 0}} \Big(\prod_{j\geq 0} f(w, j, k_j)\Big) = (1 - w)w^n \sum_{\substack{k_1+2k_2+\ldots=n \\ k_1,k_2,\ldots\geq 0}} P(n; k_1, k_2, \ldots)$$
$$= (1 - w)w^n.$$

Hence the probability that $\alpha_1 + 2\alpha_2 + \cdots \leq n$ is $(1 - w)(1 + w + \cdots + w^n) = 1 - w^{n+1}$.

(c) The average of ϕ is

$$\sum_{n\geq 0}\Big(\sum_{k_1+2k_2+\ldots=n} \phi(k_1, k_2, \ldots) \text{ prob } (\alpha_1 = k_1, \alpha_2 = k_2, \ldots)\Big)$$
$$= (1 - w)\sum_{n\geq 0} w^n\Big(\sum_{k_1+2k_2+\ldots=n} \phi(k_1, k_2, \ldots)/k_1!1^{k_1}k_2!2^{k_2}\ldots\Big).$$

(d) Let $\phi(\alpha_1, \alpha_2, \ldots) = \alpha_2 + \alpha_4 + \alpha_6 + \cdots$. Since the α's are independently chosen, the average value of the linear combination ϕ is the sum of the average values of $\alpha_2, \alpha_4, \alpha_6, \ldots$; the average value of α_m is

$$\sum_{k\geq 0} kf(w, m, k) = \sum_{k\geq 1} \frac{1}{(k-1)!}\left(\frac{w^m}{m}\right)^k e^{-w^m/m} = \frac{w^m}{m}.$$

Therefore the average value of ϕ is

$$\frac{w^2}{2} + \frac{w^4}{4} + \cdots = (1 - w)(\tfrac{1}{2}w^2 + \tfrac{1}{2}w^3 + (\tfrac{1}{2} + \tfrac{1}{4})w^4 + \cdots).$$

The desired answer is

$$\sum_{\substack{0\leq k\leq n \\ k \text{ even}}} \frac{1}{k} = \tfrac{1}{2}H_{\lfloor n/2\rfloor}.$$

(e) Let z be a real number; let $\phi(\alpha_1, \alpha_2, \ldots) = z^{\alpha_m}$. The average value of ϕ is

$$\sum_{k\geq 0} f(w, m, k)z^k = \sum_{k\geq 0} \frac{1}{k!}\left(\frac{w^m z}{m}\right)^k e^{-w^m/m} = e^{w^m(z-1)/m} = \sum_{j\geq 0} \frac{w^{mj}}{j!}\left(\frac{z-1}{m}\right)^j$$
$$= (1-w)\sum_{n\geq 0} w^n \left(\sum_{0\leq j\leq n/m} \frac{1}{j!}\left(\frac{z-1}{m}\right)^j\right)$$
$$= (1-w)\sum_{n\geq 0} w^n G_{nm}(z).$$

Hence

$$G_{nm}(z) = \sum_{0\leq j\leq n/m} \frac{1}{j!}\left(\frac{z-1}{m}\right)^j;\ p_{nmk} = \frac{1}{m^k k!} \sum_{0\leq j\leq n/m-k} \frac{(-1/m)^j}{j!};$$

(min 0, ave $1/m$, max n/m, dev $\sqrt{1/m}$).

23. The constant λ is $\int_0^1 e^{\mathrm{li}(u)}\,du$, where $\mathrm{li}(x) = \int_0^x dt/(\ln t)$. See *Transactions of the American Math. Society* **121** (1966), 340–357; many other results are proved in this paper, in particular the average length of the *shortest* cycle is approximately $\ln n/e^{\gamma}$. Further terms of the asymptotic representation of l_n are not yet known. William C. Mitchell has calculated a high-precision value of λ = .62432 99885 43550 87099 29363 8310 . . . ; no relation between λ and classical mathematical constants is known.

25. One proof, by induction on N, is based on the fact that when the Nth element is a member of s of the sets it contributes exactly

$$\binom{s}{0} - \binom{s}{1} + \binom{s}{2} - \cdots = (1-1)^s = \delta_{s0}$$

to the sum. Another proof, by induction on M, is based on the fact that the number of elements that are in S_M but not in $S_1 \cup \cdots \cup S_{M-1}$ is

$$\|S_M\| - \sum_{1\leq j<M} \|S_j \cap S_M\| + \sum_{1\leq j<k<M} \|S_j \cap S_k \cap S_M\| - \cdots.$$

26. Let $N_0 = N$ and let

$$N_k = \sum_{1\leq j_1<\cdots<j_k\leq M} \|S_{j_1} \cap \cdots \cap S_{j_k}\|.$$

Then the desired formula is

$$N_r - \binom{r+1}{r} N_{r+1} + \binom{r+2}{r} N_{r+2} - \cdots.$$

This may be proved from the principle of inclusion and exclusion itself, or by using the formula

$$\binom{r}{r}\binom{s}{r} - \binom{r+1}{r}\binom{s}{r+1} + \cdots = \binom{s}{r}\binom{s-r}{0} - \binom{s}{r}\binom{s-r}{1} + \cdots = \delta_{sr}$$

as in exercise 25.

27. Let S_j be the multiples of m_j in the stated range and let $N = am_1 \ldots m_t$. Then

$$\|S_j \cap S_k\| = N/m_j m_k, \text{ etc.},$$

so the answer is

$$N - N \sum_{1 \le j \le t} \frac{1}{m_j} + N \sum_{1 \le j < k \le t} \frac{1}{m_j m_k} - \cdots = N\left(1 - \frac{1}{m_1}\right) \cdots \left(1 - \frac{1}{m_t}\right).$$

This provides another proof of exercise 1.2.4–30, if we let $m_1, \ldots, m_t$ be the primes dividing N.

1. Calling sequence: `JMP MAXN`; or, `JMP MAX100` if $n = 100$.
   Entry conditions: For the `MAXN` entrance, rI3 $= n$; assume $n \geq 1$.
   Exit conditions: Same as in (4).

2.
```
MAX50   STJ    EXIT
        ENT3   50
        JMP    2F
```

3. Entry conditions: n = rI1 if rI1 > 0; otherwise $n = 1$.
   Exit conditions: rA and rI2 as in (4); rI1 unchanged; rI3 = min (0, rI1); rJ = `EXIT` + 1; CI unchanged if $n = 1$, otherwise CI is greater, equal, or less, according as the maximum is greater than $X[1]$, equal to $X[1]$ and rI2 > 1, or equal to $X[1]$ with rI2 $= 1$. (The analogous exercise for subroutine (9) would of course be somewhat more complicated.)

4.
```
SMAX1   ENT1   1      r = 1
SMAX    STJ    EXIT   general r
        JMP    2F     continue as before
...
        DEC3   0,1    decrease by r
        J3P    1B
EXIT    JMP    *      exit.
```

Calling sequence: `JMP SMAX`; or, `JMP SMAX1` if $r = 1$.
Entry conditions: rI3 $= n$, assumed positive; for the `SMAX` entrance, rI1 $= r$, assumed positive.
Exit conditions: rA $= \max_{0 \leq k < n/r}$ `CONTENTS`(X + $n - kr$) = `CONTENTS`(X + rI2); rI3 $= ((n - 1) \bmod r) + 1 - r$.

5. Any other register can be used. For example,

Calling sequence:
```
ENTA *+2
JMP MAX100
```
Entry conditions: None.
Exit conditions: Same as in Ex. 1.

The code is the same as (1) except the first instruction would be

"`MAX100 STA EXIT(0:2)`".

6. (Solution by Joel Goldberg.)

```
MOVE   STJ    1F
       STJ    2F
       STJ    3F
       STA    4F
       ST2    5F(0:2)
1H     LDA    *(0:3)
       STA    *+1(0:3)
       ENTA   *
2H     LD2N   *(4:4)
       J2Z    1F
       DECA   0,2
       STA    2F(0:2)
2H     LDA    *,2
       STA    0,1
       INC1   1
       INC2   1
       J2N    2B
1H     LDA    4F
5H     ENT2   *
3H     JMP    *
4H     CON    0
```

SECTION 1.4.2

1. If one coroutine calls the other only once, it is nothing but a subroutine; so we need an application in which each coroutine calls the other in at least two distinct places. Even then, it is often easy to set some sort of switch or use some property of the data, so that upon entry to a fixed place within one coroutine it is possible to branch to one of two desired places—so again, nothing more than a subroutine would be required. Coroutines become correspondingly more useful as the number of references between them grows larger.

2. The first character found by `IN` would be lost.

3. *Almost* true, since "`CMPA =10=`" within `IN` is then the only comparison instruction of the program, and since the code for "." is 40. But if the final period were preceded by a replication digit, the test would go unnoticed. (*Note:* The most efficient program would probably remove lines 40, 44, and 48, and would insert "`CMPA PERIOD`" between lines 26 and 27. If the state of the comparison indicator is to be used across coroutines, however, it must be recorded as part of the coroutine characteristics in the documentation of the program.)

4. (a) On the IBM 650, using SOAP assembly language, we would have the calling sequences "`LDD A`" and "`LDD B`"; and linkage "`A STD BX AX`" and "`B STD AX BX`" (with the two linkage instructions preferably in core). (b) On the IBM 709, using common assembly languages, the calling sequences would be "`TSX A,4`" or "`TSX B,4`"; the linkage instructions would be

```
A   SXA  BX,4          B   SXA  AX,4
AX  AXT  1-A1,4        BX  AXT  1-B1,4
    TRA  1,4               TRA  1,4
```

(c) On the CDC 1604, the calling sequences would be "return jump" (`SLJ 4`) to `A` or `B`, and the linkage would be, e.g.,

```
B:  SLJ  A1;  ALS  0
A:  SLJ  B1;  SLJ  B.
```

(d) Most other machines are similar to one of these three. For example, System/360 would be analogous to the 709, or we could use `BALR r,r` in short coroutines.

5. "`STA HOLDAIN; LDA HOLDAOUT`" between `OUT` and `OUTX`, and "`STA HOLDAOUT; LDA HOLDAIN`" between `IN` and `INX`.

6. Within `A` write "`JMP AB`" to activate `B`, "`JMP AC`" to activate `C`. Similarly locations `BA`, `BC`, `CA`, and `CB` would be used within `B` and `C`. The linkage is:

```
AB  STJ  AX          BC  STJ  BX          CA  STJ  CX
BX  JMP  B1          CX  JMP  C1          AX  JMP  A1
CB  STJ  CX          AC  STJ  AX          BA  STJ  BX
    JMP  BX              JMP  CX              JMP  AX
```

(*Note:* With n coroutines, $2(n-1)n$ cells would be required for the linkage. If n is large, a "centralized" routine for linkage could of course be used; a method with $3n+2$ cells would not be hard to invent. But in practice the faster method above requires just $2m$ cells, where m is the number of pairs (i, j) such that coroutine i jumps to coroutine j. When there are many coroutines each independently jumping to others, we usually have a situation in which the sequence of control is under external influence, as discussed in Section 2.2.5.)

SECTION 1.4.3.1

1. It is used only twice, both times immediately followed by a call on MEMORY, so it would be slightly more efficient to make it a special entrance to the MEMORY subroutine, and also to make it put $-R$ in rI2.

2.
```
SHIFT   J5N     ADDRERROR
        DEC3    5
        J3P     FERROR
        LDA     AREG
        LDX     XREG
        LD1     1F,3(4:5)
        ST1     2F(4:5)
        J5Z     CYCLE
```

```
2H      SLA   1
        DEC5  1
        J5P   2B
        JMP   STOREAX
        SLA   1
        SRA   1
        SLAX  1
        SRAX  1
        SLC   1
1H      SRC   1
```

3.

```
MOVE    J3Z   CYCLE
        JMP   MEMORY
        SRAX  5
        LD1   I1REG
        LDA   SIGN1
        JAP   *+2
        J1NZ  MEMERROR
        CMP1  =BEGIN=
        JGE   MEMERROR
        STX   0,1
        LDA   CLOCK
        INCA  2
        STA   CLOCK
        INC1  1
        ST1   I1REG
        INC5  1
        DEC3  1
        JMP   MOVE
```

4. Just insert "`IN 0(16)`" and "`JBUS *(16)`" between lines 03 and 04. (Of course on another computer this would be considerably different since it would be necessary to convert to `MIX` character code.)

5. Central control time is $34u$, plus $15u$ if indexing is required; the `GETV` subroutine takes $52u$, plus $5u$ if $L \neq 0$; extra time to do the actual loading is $11u$ for `LDA` or `LDX`, $13u$ for `LDi`, $21u$ for `ENTA` or `ENTX`, $23u$ for `ENTi` (add $2u$ to the latter two times if `M` $= 0$). Summing up, we have a total time of $97u$ for `LDA` and $55u$ for `ENTA`, plus $15u$ for indexing, and plus $5u$ or $2u$ in certain other circumstances. It would seem that simulation in this case is causing roughly a 50:1 ratio in speeds. (Results of a test run which involved $178u$ of simulated time required $8422u$ of actual time, a 47:1 ratio.)

7. Execution of `IN` or `OUT` sets a variable associated with the appropriate input device to the time when transmission is desired. The "`CYCLE`" control routine interrogates these variables on each cycle, to see if `CLOCK` has exceeded either (or both) of them; if so, the transmission is carried out and the variable is set to "infinity". (When more than two I/O units must be handled in this way, there will be so many variables it will be preferable to keep them in a sorted list using linked memory techniques; see Section 2.2.5.)

1. Change lines 49 and 50 to the following sequence:

```
LEAVE   STX   3F
        ST1   4F
        LD1   JREG
        LDA   -1,1
        LDX   1F
        STX   -1,1
        JMP   -1,1
1H      JMP   *+1
        STA   -1,1
2H      LD1   4F
        LDX   3F
        LDA   AREG
LEAVEX  JSJ   *
3H      CON   0
4H      CON   0
```

The operator "JSJ" here is, of course, particularly crucial.

2.

```
. . . . . lines 01–04
        ST1   I1REG
. . . . . lines 05–07
BUF     LD1   BUF2
        STA   1,1
. . . . . lines 08–12
        STA   2,1
        LDA   AREG
        STA   3,1
        STX   4,1
        LDA   I1REG
        STA   5,1
        ST2   6,1
        ST3   7,1
        ST4   8,1
        ST5   9,1
        ST6   10,1
        LDA   JREG
        STA   11,1
        STZ   12,1
        ENTA  1
        JNOV  1F
        STA   12,1
        LDA   BIG
        ADD   BIG
1H      STZ   13,1
        ST1   BUF(0:2)
. . . . . lines 13–30
```

```
        LD1    I1REG
. . . . . . . . . . . lines 31–34
        ST1    I1REG
. . . . . . . . . . . lines 35–49
        LD1    I1REG
. . . . . . . . . . . lines 50–53
BIG     CON    50(1:1)
BUF1    CON    *+14
        ORIG   *+13
BUF2    CON    BUF1
        ORIG   *+13
```

Note: Since only 13/100 of each tape record gets useful information, this program is not using tape very efficiently, and since the tape time is the limiting factor the speed of tracing could also be improved if we put several 13-word groups into each tape record. The program could be readily modified to have 14 buffers, with a tape record written only after seven buffers are full (linking the buffers together and using a sentinel); this would make the tape length and execution time $\frac{1}{7}$ as long, but it would require much more memory.

3. Tape is faster; and the editing of this information into characters while tracing would consume far too much space.

4, 5. A true trace, as desired in Ex. 6, would not be obtained, since restriction (a) mentioned in the text is violated. Early examination leads to speculation that the first attempt to trace CYCLE would cause a loop back to ENTER.

6. Suggestion: keep a table of values of each memory location within the trace area that has been changed by the outer program.

SECTION 1.4.4

1. (a) No, the input operation may not yet be complete. (b) No, the input operation may be going just a little faster, and this is much too risky.

2.
```
        ENT1  2000
        JBUS  *(6)
        MOVE  1000(50)
        MOVE  1050(50)
        OUT   2000(6)
```

3.
```
WORDOUT STJ   1F
        INC5  1
        LDX   0,5
        JXZ   2F
        OUT   -100,5(V)
        LD5   1,5
        ENT1  0,5
        MOVE  -1,1(50)
        MOVE  -1,1(50)
        ST5   CURRENT(0:2)
```

```
2H      STA   0,5
1H      JMP   *
* BUFFER AREAS
        CON   0
OUTBUF1 ORIG  *+100
        CON   -1
        CON   *+2
        CON   0
OUTBUF2 ORIG  *+100
        CON   -1
        CON   OUTBUF1
```

At the beginning of the program, give the instruction "`ENT5 OUTBUF1-1`". At the end of the program, put

```
CURRENT  OUT  *(V)
         OUT  OUTBUF1(V)
         IOC  0(V)
```

Write out last record. (optional; writes an extra record in case of later input buffering, and rewinds the tape)

4. If the calculation time exactly equals the I/O time (which is the most favorable situation), both the computer and peripheral device running simultaneously will take half as long as if they ran separately. Formally, let C be the calculation time for the entire program, and let T be the total I/O time required; then the best possible running time with buffering is $\max(C, T)$, while the running time without buffering is $C + T$; and of course $\frac{1}{2}(C + T) \leq \max(C, T) \leq C + T$. However, there are some devices which have a "shutdown penalty" which causes an extra amount of time to be lost if too long an interval occurs between references to that unit; in such a case, better than 2:1 ratios are possible.

5. Best ratio is $(n + 1):1$.

6.
```
IN     INBUF1(U)        IN     INBUF2(U)
ENT6   INBUF2+99   or   ENT6   INBUF1+99
```

(possibly preceded by `IOC 0(U)` to rewind the tape just in case it is necessary).

7. One way is to use coroutines:

```
INBUF1   ORIG  *+100
         CON   *+1
INBUF2   ORIG  *+100
         CON   INBUF1
1H       LDA   INBUF2+100,6
         JMP   MAIN
         INC6  1
         J6N   1B
WORDIN1  IN    INBUF2(U)
         ENN6  100
2H       LDA   INBUF1+100,6
         JMP   MAIN
         INC6  1
         J6N   2B
```

```
                IN      INBUF1(U)
                ENN6    100
                JMP     1B
WORDIN          STJ     MAINX
WORDINX         JMP     WORDIN1
MAIN            STJ     WORDINX
MAINX           JMP     *
```

Adding a few more instructions to take advantage of special cases will make this routine actually faster than (4).

8. At the time shown in Fig. 23, the two red buffers have been filled with line images, and the one indicated by NEXTR is being printed. At the same time, the program is computing between RELEASE and ASSIGN. When the program ASSIGNs, the green buffer indicated by NEXTG becomes yellow; NEXTG moves clockwise and the program begins to fill the yellow buffer. When the output operation is complete, NEXTR moves clockwise, the buffer that has just been printed turns green, and the remaining red buffer begins to be printed. Finally, the program RELEASEs the yellow buffer and it too is ready for subsequent printing.

9, 10, 11:

time	*action* ($N = 1$)	*action* ($N = 2$)	*action* ($N = 4$)
0	ASSIGN(BUF1)	ASSIGN(BUF1)	ASSIGN(BUF1)
1000	RELEASE, OUT BUF1	RELEASE, OUT BUF1	RELEASE, OUT BUF1
2000	ASSIGN(wait)	ASSIGN(BUF2)	ASSIGN(BUF2)
3000		RELEASE	RELEASE
4000		ASSIGN(wait)	ASSIGN(BUF3)
5000			RELEASE
6000			ASSIGN(BUF4)
7000			RELEASE
8000			ASSIGN(wait)
8500	BUF1 assigned, output stops	BUF1 assigned, OUT BUF2	BUF1 assigned, OUT BUF2
9500	RELEASE, OUT BUF1	RELEASE	
10500	ASSIGN(wait)	ASSIGN(wait)	
15500			RELEASE

and so on. Total time when $N = 1$ is $110000u$; when $N = 2$ it is $89000u$; when $N = 3$ it is $81500u$; and when $N \geq 4$ it is $76000u$.

12. The following code should be inserted before "LD5 -1,5" in program B:

```
STA     2F
LDA     3F
CMPA    15,5(5:5)
LDA     2F
```

Then the instruction "JMP 1B" should be changed to

```
JNE     1B
JMP     COMPUTE
JMP     *-1         [or JMP COMPUTEX]
```

```
2H  CON   0
3H  ALF       .    ∎
```

13.

```
    JRED  CONTROL(U)
    J6NZ  *-1        ∎
```

14. If $N = 1$ the process would loop indefinitely; otherwise the construction will have the effect that there are two yellow buffers. This can be useful if the computational program wants to refer to two buffers at once, although it ties up buffer space. In general, the excess of ASSIGNs over RELEASEs should be nonnegative and not greater than N.

15.

```
U         EQU   0
V         EQU   1
BUF1      ORIG  *+100
BUF2      ORIG  *+100
BUF3      ORIG  *+100
TAPECPY   ENT1  99
          IN    BUF1(U)
1H        IN    BUF2(U)
          OUT   BUF1(V)
          IN    BUF3(U)
          OUT   BUF2(V)
          IN    BUF1(U)
          OUT   BUF3(V)
          DEC1  3
          J1P   1B
          OUT   BUF1(V)
          HLT
          END   TAPECPY   ∎
```

This is a special case of the algorithm indicated in Fig. 26.

18. Partial solution: in the algorithms below, t is a variable which is set to 0 when the I/O device is active, and $t = 1$ when it is idle.

Algorithm A (ASSIGN, a normal state subroutine).
This algorithm is unchanged from Algorithm 1.4.4A.

Algorithm R (RELEASE, a normal state subroutine).

R1. Increase n by one.
R2. If $t = 0$, cause an interrupt (using the INT operator) which should go to step B2. ∎

Algorithm B (Buffer control routine, which processes interrupts).

B1. If $n = 0$, set $t \leftarrow 0$ and restart main program.
B2. Set $t \leftarrow 1$, and initiate I/O from the buffer area specified by NEXTR.
B3. Restart the main program; an "I/O Complete" condition will interrupt to step B4.
B4. Advance NEXTR to the next clockwise buffer.
B5. Decrease n by one, and go to step B1. ∎

1. (a) TAG(NEXT(TOP)) = TAG(NEXT(242)) = TAG(386) = 0. (b) Λ.

2. Whenever V is a link variable (else CONTENTS(V) makes no sense) whose value is not Λ. It is wise to *avoid* using LOC in contexts like this.

3. Set NEWCARD ← TOP, and if TOP ≠ Λ set TOP ← NEXT(TOP).

4. C1. Set X ← LOC(TOP). (For convenience we make the reasonable assumption that TOP = NEXT(LOC(TOP)). This assumption is compatible with program (5).)

C2. If NEXT(X) ≠ Λ, set X ← NEXT(X) and repeat this step.

C3. Set NEXT(X) ← NEWCARD, NEXT(NEWCARD) ← Λ, TAG(NEWCARD) ← 1. ∎

5. D1. Set X ← LOC(TOP), Y ← TOP. (See step C1 above. By hypothesis, Y ≠ Λ. Throughout the algorithm which follows, X trails one step behind Y in the sense that Y = NEXT(X).)

D2. If NEXT(Y) ≠ Λ, set X ← Y, Y ← NEXT(Y), and repeat this step.

D3. (Now NEXT(Y) = Λ, so Y points to the bottom card; also X points to the next-to-last card.) Set NEXT(X) ← Λ, NEWCARD ← Y. ∎

6. (b) and (d). *Not* (a)! CARD is a node, not a link to a node.

7. Sequence (a) gives NEXT(LOC(TOP)), which in this case is identical to TOP; sequence (b) is correct. Note the following important distinction, when X is a link variable: In an algorithm, the symbol X refers to the value of variable X; but in assembly language it refers to the *location* of X. There is no need for confusion; consider the analogous example when X is a numeric variable: To bring X into register A, we write LDA X, not ENTA X.

8. Let rA ≡ N, rI1 ≡ X.

```
ENTA   0           N ← 0.
LD1    TOP         X ← TOP.
J1Z    *+4         Is X = Λ?
INCA   1           N ← N + 1.
LD1    0,1(NEXT)   X ← NEXT(X).
J1NZ   *-2                ∎
```

9. Let rI2 ≡ X.

```
PRINTER  EQU   18          Unit number for printer
TAG      EQU   1:1
NEXT     EQU   4:5         Definition of fields
NAME     EQU   0:5
PBUF     ALF   PILE        Message printed in case
         ALF   EMPTY         pile is empty
         ORIG  PBUF+24
BEGIN    LD2   TOP         Set X ← TOP.
         J2Z   2F          Is the pile empty?
1H       LDA   0,2(TAG)    rA ← TAG(X).
         ENT1  PBUF        Get ready for MOVE instruction.
         JBUS  *(PRINTER)  Wait until printer is ready.
         JAZ   *+3         Is TAG = 0 (is card face up)?
```

```
         MOVE  PAREN(3)          No: set parentheses.
         JMP   *+2
         MOVE  BLANKS(3)         Yes: set blanks.
         LDA   1,2(NAME)         rA ← NAME(X).
         STA   PBUF+1
         LD2   0,2(NEXT)         Set X ← NEXT(X).
2H       OUT   PBUF(PRINTER)     Print the line.
         J2NZ  1B                If X ≠ Λ, repeat the print loop.
DONE     HLT
PAREN    ALF       (
BLANKS   ALF
         ALF   )
         ALF                     ▮
```

SECTION 2.2.1

1. Yes (consistently insert all items at one of the two ends).

2. To obtain 325641, do SSSXXSSXSXXX (in the notation of the following exercise). The order 154623 cannot be achieved, since 2 can precede 3 only if it is removed from the stack before 3 has been inserted.

3. An admissible sequence is one in which the number of X's never exceeds the number of S's if we read from the left to the right.

Two different admissible sequences must give a different result, since if the two sequences agree up to a point where one has S and the other has X, the latter sequence outputs a symbol which cannot possibly be output before the symbol just inserted by the S of the former sequence.

4. This problem is equivalent to many other interesting problems, such as the enumeration of binary trees, the number of ways to insert parentheses into a formula, and the number of ways to divide a polygon into triangles, and it appeared as early as 1759 in notes by Euler and Segner (see Section 2.3.4.6). For further references, see A. Erdélyi and I. M. H. Etherington, *Edinburgh Mathematical Notes*, **32** (1940), 1–12.

The following elegant solution is due to D. André (1878): There are obviously $\binom{2n}{n}$ sequences of S's and X's that contain n of each. It remains to evaluate the number of *inadmissible* sequences (which contain the right number of S's and X's but which violate the other condition). In any inadmissible sequence, locate the first X for which the X's outnumber the S's. Then in the partial sequence leading up to and including this X, replace all X's by S and all S's by X. The result is a sequence with $(n+1)$ S's and $(n-1)$ X's. Conversely for every sequence of the latter type we can reverse the process and find the inadmissible sequence of the former type which leads to it. For example, the sequence XXSXSSSXXSX must have come from SSXSXXXXXSX. This correspondence shows that the number of inadmissible sequences is $\binom{2n}{n-1}$. Hence $a_n = \binom{2n}{n} - \binom{2n}{n-1}$.

Using the same idea, we can solve the more general "ballot problem" of probability theory, which essentially is the enumeration of all partial admissible sequences with a given number of S's and X's. For the history of the ballot problem and some generalizations, see papers by A. Dvoretzky and T. Motzkin, *Duke Math. Journal* **14** (1947),

305–313; L. Takács, *Journal of the American Statistical Association* **57** (1962), 327; John Riordan, *Annals of Math. Statistics* **35** (1964), 369–379; and exercise 2.3.4.4–32.

We present here a new method for solving the ballot problem with the use of double generating functions, since this method lends itself to the solution of more difficult problems such as Ex. 11.

Let g_{nm} be the number of sequences of S's and X's of length n, in which the number of X's never exceeds the number of S's if we count from the left, and in which there are m more S's than X's in all. Then $a_n = g_{(2n)0}$. Obviously g_{nm} is zero unless $m + n$ is even. The recurrence relation satisfied by these numbers is easily found to be

$$g_{(n+1)m} = g_{n(m-1)} + g_{n(m+1)},\ m \geq 0,\ n \geq 0;\ g_{0m} = \delta_{0m}.$$

We set up the double generating function $G(x, z) = \sum_{n,m} g_{nm}x^m z^n$, and let $g(z) = G(0, z)$. The recurrence relation above transforms into

$$\left(x + \frac{1}{x}\right) G(x, z) = \frac{1}{x} g(z) + \frac{1}{z}\left(G(x, z) - 1\right), \quad \text{i.e.} \quad G(x, z) = \frac{zg(z) - x}{z(x^2 + 1) - x}.$$

This equation unfortunately tells us nothing if we set $x = 0$, but we can proceed by factoring the denominator as $z(1 - r_1(z)x)(1 - r_2(z)x)$ where

$$r_1(z) = \frac{1}{2z}(1 + \sqrt{1 - 4z^2}), \qquad r_2(z) = \frac{1}{2z}(1 - \sqrt{1 - 4z^2}).$$

(Note that $r_1 + r_2 = 1/z$; $r_1 r_2 = 1$.) We now proceed heuristically; the problem is to find some value of $g(z)$ such that $G(x, z)$ as given by the formula above has an infinite power series expansion in x and z. Note that $r_2(z)$ has such an expansion, and $r_2(0) = 0$; and for fixed z, the value $x = r_2(z)$ causes the denominator of $G(x, z)$ to vanish. This suggests that we might choose $g(z)$ so that the numerator also vanishes when $x = r_2(z)$, i.e. take $zg(z) = r_2(z)$. The equation for $G(x, z)$ now simplifies to

$$G(x, z) = \frac{r_2(z)}{z(1 - r_2(z)x)} = \sum_{n \geq 0} (r_2(z))^{n+1} x^n z^{-1}.$$

Since this is a power series expansion which satisfies the original equation, we must have found the right choice of $g(z)$.

The coefficients of $g(z)$ are the solution to our problem. Actually we can go further and derive a simple form for all the coefficients of $G(x, z)$: By the binomial theorem,

$$r_2(z) = \sum_{k \geq 0} z^{2k+1} \binom{2k+1}{k} \frac{1}{2k+1}.$$

Let $w = z^2$ and $r_2(z) = zf(w)$. Then

$$f(w) = \sum_{k \geq 0} A_k(1, -2) w^k$$

in the notation of Ex. 1.2.6–25; hence

$$f(w)^r = \sum_{k \geq 0} A_k(r, -2) w^k.$$

We now have

$$G(x, z) = \sum_{n,m} A_m(n, -2)x^n z^{2m+n},$$

so the general solution is

$$g_{(2n)(2m)} = \binom{2n+1}{n-m}\frac{2m+1}{2n+1} = \binom{2n}{n-m} - \binom{2n}{n-m-1};$$

$$g_{(2n+1)(2m+1)} = \binom{2n+2}{n-m}\frac{2m+2}{2n+2} = \binom{2n+1}{n-m} - \binom{2n+1}{n-m-1};$$

$$a_n = \binom{2n+1}{n}\frac{1}{2n+1}.$$

5. If $j < k$ and $p_j < p_k$, we must have taken p_j off the stack before p_k was put on; if $p_j > p_k$, we must have left p_k on the stack until after p_j was put on. Combining these two rules, the condition $i < j < k$ and $p_j < p_k < p_i$ is impossible since it means p_j must go off before p_k and after p_i, yet p_i appears after p_k.

Conversely, the desired permutation can be obtained by using the algorithm "For $j = 1, 2, \ldots, n$ input zero or more items (as many as necessary) until p_j first appears in the stack, then output p_j." This algorithm can fail only if we reach a j for which p_j is not at the top of the stack but it is covered by some element p_k for $k > j$. Since the contents of the stack is always monotone increasing, we have $p_j < p_k$. This element p_k could have gotten there only if it is less than p_i for some $i < j$.

6. Only the trivial one, $1\ 2 \ldots n$, by the nature of a queue.

7. An input-restricted deque which first outputs n must simply put the values $1\ 2 \ldots n$ on the deque in order as its first n operations. An output-restricted deque which first outputs n must put the values $p_1\ p_2 \ldots p_n$ on its deque as its first n operations. Therefore we find the answers (a) 4132 (b) 4213 (c) 4231. By trial, these answers are unique.

8. When $n = 4$, no; when $n = 5$, there are four: 51324, 51342, 52314, 52341.

9. By operating in reverse, we can get the inverse of any input-restricted permutation with an output-restricted deque, and conversely. This sets up a one-to-one correspondence between the two sets of permutations.

10. (i) There should be n X's and n combined S's and Q's. (ii) The number of X's must never exceed the combined number of S's and Q's, if we read from the left. (iii) Whenever the number of X's equals the combined number of S's and Q's (reading from the left), the next character must be a Q. (iv) The two operations XQ must never be adjacent in this order.

Clearly rules (i) and (ii) are necessary. The extra rules (iii) and (iv) are added to remove ambiguity, since S is the same as Q when the scroll is empty, and since XQ can always be replaced by QX. Thus, any obtainable sequence corresponds to at least one admissible sequence.

To show that two admissible sequences give different permutations, consider sequences which are identical up to a point, and then one sequence has an S while the other has an X or Q. Since by (iii) the deque is not empty, clearly different permutations (relative to the order of the element moved on by S) are obtained by the two

sequences. The remaining case is where sequences A, B agree up to a point and then sequence A has Q, sequence B has X. Sequence B may have further X's at this point, and by (iv) they must be followed by an S, so again the permutations are different.

11. Proceeding as in Ex. 4, we let g_{nm} be the number of partial admissible sequences of length n, leaving m elements on the deque, *not* ending in the symbol X; h_{nm} is defined analogously, for those sequences that *do* end with X. We have $g_{(n+1)m} = 2g_{n(m-1)}$ $(+h_{n(m-1)}$ if $m > 1)$; $h_{(n+1)m} = g_{n(m+1)} + h_{n(m+1)}$. Define $G(x, z)$ and $H(x, z)$ analogously to the definition in Ex. 4; we have

$$G(x, z) = xz + 2x^2z^2 + 4x^3z^3 + (8x^4 + 2x^2)z^4 + (16x^5 + 8x^3)z^5 + \cdots;$$

$$H(x, z) = z^2 + 2xz^3 + (4x^2 + 2)z^4 + (8x^3 + 6x)z^5 + \cdots.$$

If $h(z) = H(0, z)$, we find

$$\frac{1}{z} G(x, z) = 2xG(x, z) + x\big(H(x, z) - h(z)\big) + x,$$

$$\frac{1}{z} H(x, z) = \frac{1}{x} G(x, z) + \frac{1}{x} \big(H(x, z) - h(z)\big).$$

Consequently

$$G(x, z) = \frac{xz\big(x - z - xh(z)\big)}{x - z - 2x^2z + xz^2}.$$

As in Ex. 4, we try choosing $h(z)$ so the numerator cancels with a factor of the denominator. We find $G(x, z) = xz/\big(1 - 2r_2(z)\big)$ where

$$r_2(z) = \frac{1}{4z} \big(z^2 + 1 - \sqrt{(z^2 + 1)^2 - 8z^2}\big).$$

Using the convention $b_0 = 1$, the desired generating function comes to

$$\tfrac{1}{2}(3 - z - \sqrt{1 - 6z + z^2}) = 1 + z + 2z^2 + 6z^3 + 22z^4 + 90z^5 + \cdots.$$

By differentiation we find a recurrence relation that is handy for calculation: $nb_n = 3(2n - 3)b_{n-1} - (n - 3)b_{n-2}$, $n \geq 2$.

12. If $0 < \alpha < 1$, the coefficient of z^n in $\sqrt{1 - z}\,\sqrt{1 - \alpha z}$ is

$$(-1)^n \binom{\frac{1}{2}}{n} \left(\sqrt{1 - \alpha} + O\left(\frac{1}{n}\right)\right)$$

which by Stirling's approximation is asymptotically

$$-\frac{1}{2}\sqrt{\frac{1 - \alpha}{\pi n^3}}.$$

Now $1 - 6z + z^2 = \big(1 - (3 + \sqrt{8})z\big)\big(1 - (3 - \sqrt{8})z\big)$. Substitute $w = (3 + \sqrt{8})z$ and apply the above formulas; we find $a_n \sim 4^n/\sqrt{\pi n^3}$; $b_n \sim c(3 + \sqrt{8})^n n^{-3/2}$, where $c = \frac{1}{2}\sqrt{(3\sqrt{2} - 4)/\pi} \approx 0.139$.

1. M — 1 (*not* M). If we allowed M items, as (6) and (7) do, it would be impossible to distinguish an empty queue from a full one by examination of R and F, since only M possibilities can be detected. It is better to give up one storage cell than to make the program overly complicated!

2. Delete from rear: if R = F then UNDERFLOW; Y ← X[R]; if R = 1 then R ← M, otherwise R ← R — 1. Insert at front: (Assume that initially R = F = 1 instead of R = F = 0.) Set X[F] ← Y; if F = 1 then F ← M, otherwise F ← F — 1; if F = R then OVERFLOW.

3. (a) LD1 I; LDA BASE,7:1. This takes 5 cycles instead of 4 or 8 as in (8).

(b) *Solution 1:* LDA BASE,2:7 where each base address is stored with $I_1 = 0$, $I_2 = 1$. *Solution 2:* If it is desired to store the base addresses with $I_1 = I_2 = 0$, we could write LDA X,7:1 where location X contains NOP BASE,2:7. The second solution takes one more cycle, but allows the base table to be used with any index registers.

(c) This is equivalent to "LD4 X(0:2)", and takes the same execution time.

4. (i) NOP *,7. (ii) LDA X,7:7. (iii) This is impossible, because 7:7 has been disallowed in an indirect location; see Ex. 5. (iv) LDA X,7:1 with the auxiliary constants

```
X   NOP   *+1,7:2
    NOP   *+1,7:3
    NOP   *+1,7:4
    NOP   0,5:6
```

Execution time is 6 units. (v) INC6 X,7:6 where X contains NOP 0,6:6.

5. (a) Consider the instruction ENTA 1000,7:7 with the memory configuration

location	ADDRESS	I_1	I_2
1000:	1001	7	7
1001:	1004	7	1
1002:	1002	2	2
1003:	1001	1	1
1004:	1005	1	7
1005:	1006	1	7
1006:	1008	7	7
1007:	1002	7	1
1008:	1003	7	2

and with rI1 = 1, rI2 = 2. We find that 1000,7,7 = 1001,7,7,7 = 1004,7,1,7,7 = 1005,1,7,1,7,7 = 1006,7,1,7,7 = 1008,7,7,1,7,7 = 1003,7,2,7,1,7,7 = 1001,1,1,2,7,1,7,7 = 1002,1,2,7,1,7,7 = 1003,2,7,1,7,7 = 1005,7,1,7,7 = 1006,1,7,1,7,7 = 1007,7,1,7,7 = 1002,7,1,1,7,7 = 1002,2,2,1,1,7,7 = 1004,2,1,1,7,7 = 1006,1,1,7,7 = 1007,1,7,7 = 1008,7,7 = 1003,7,2,7 = 1001,1,1,2,7 = 1002,1,2,7 = 1003,2,7 = 1005,7 = 1006,1,7 = 1007,7 = 1002,7,1 = 1002,2,2,1 = 1004,2,1 = 1006,1 = 1007. (A perhaps faster way to do this derivation by hand would be to evaluate successively the addresses specified in locations 1002, 1003, 1007, 1008, 1005, 1006, 1004, 1001, 1000 in this order, but it would seem that a computer would need to go about the evaluation essentially as shown.) The author tried out several fancy schemes for changing the contents of

memory while evaluating the address, yet designed so that everything would be restored again by the time the final address has been obtained. Similar algorithms appear in Section 2.3.5. However, these attempts were unfruitful and it appears there is just not enough room to store the necessary information.

(b, c) Let H, C be auxiliary registers and let N be a counter. To get the effective address M, for the instruction in location L, do the following:

A1. [Initialize.] Set $H \leftarrow 0$, $C \leftarrow L$, $N \leftarrow 0$. (C will be the "current" location, H is used to add together the contents of various index registers, and N measures the "depth" of indirect addressing.)

A2. [Examine address.] Set $M \leftarrow \texttt{ADDRESS}(C)$. If $I_1(C) = j$, $1 \le j \le 6$, set $M \leftarrow M + rIj$. If $I_2(C) = j$, $1 \le j \le 6$, set $H \leftarrow H + rIj$. If $I_1(C) = I_2(C) = 7$, set $N \leftarrow N + 1$, $H \leftarrow 0$.

A3. [Indirect?] If either $I_1(C)$ or $I_2(C)$ equals 7, set $C \leftarrow M$ and go to A2. Otherwise set $M \leftarrow M + H$, $H \leftarrow 0$.

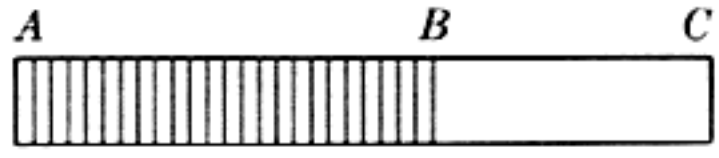
A4. [Reduce depth.] If $N > 0$, set $C \leftarrow M$, $N \leftarrow N - 1$, and go to A2. Otherwise M is the desired answer. ∎

This algorithm will handle any situation correctly except those in which $I_1 = 7$ and $1 \le I_2 \le 6$ *and* the evaluation of the address in ADDRESS involves a case with $I_1 = I_2 = 7$. The effect is as if I_2 were zero. To understand the operation of algorithm A, consider the notation of part (a); the state "L,7,7,7,7,7,1,2,3,5" is represented in the above algorithm by C or M = L, N = 5 (the number of 7's), and H = (rI1) + (rI2) + (rI3) + (rI5) (the post-indexing). In a solution to part (b) of this exercise, the counter N will always be either 0 or 1.

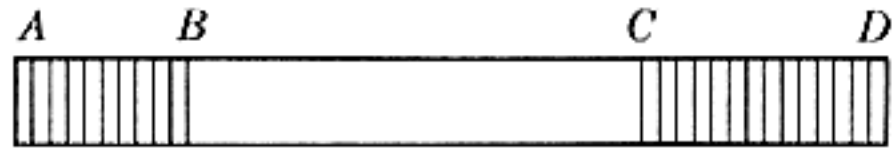
6. (c) causes OVERFLOW. (e) causes UNDERFLOW, and if the program resumes it causes OVERFLOW on the final I_2.

7. No, since TOP[i] must be greater than OLDTOP[i].

8. With a stack, the useful information appears at one end with the vacant information at the other:



where A = BASE[j], B = TOP[j], C = BASE[$j+1$]. With a queue or deque, the useful information appears at the ends with the vacant information somewhere in the middle:



or in the middle with the vacant information at the ends:

where A = BASE[j], B = REAR[j], C = FRONT[j], D = BASE[$j+1$]. The two cases are distinguished by the conditions $B \le C$, $B \ge C$, respectively. The algorithms are therefore to be modified in an obvious way so as to widen or narrow the gaps of vacant

information. (Thus in case of overflow, i.e. when $B = C$, we make empty space between B and C by moving one part and not the other.)

9. Given any sequence specification $a_1, a_2, \ldots, a_m$ there is one move operation required for every pair (j, k) such that $j < k$ and $a_j > a_k$. The number of such pairs is therefore the number of moves required. Now imagine all n^m specifications written out, and for each of the $\binom{m}{2}$ pairs (j, k) with $j < k$ count how many specifications have $a_j > a_k$. Clearly this is $\binom{n}{2}$, the number of choices for a_j and a_k, times n^{m-2}, the number of ways to fill in the remaining places. Hence the total number of moves among all specifications is $\binom{m}{2}\binom{n}{2}n^{m-2}$. Divide this by n^m to get the average, Eq. (12).

10. As in Ex. 9 we find the expected value is

$$\binom{m}{2}\sum_{1\le j<k\le n} p_j p_k = \frac{1}{2}\binom{m}{2}((p_1+\cdots+p_n)^2-(p_1^2+\cdots+p_n^2))$$
$$= \frac{1}{2}\binom{m}{2}(1-(p_1^2+\cdots+p_n^2)).$$

For this model, it makes *absolutely no difference* what the relative order of the lists is! (A moment's reflection explains why; if we consider all possible permutations of a given sequence $a_1, \ldots, a_m$ we find the total number of moves summed over all these permutations depends only on the number of pairs of distinct elements $a_j \neq a_k$.)

11. Counting as before, we find the expected number is

$$\frac{1}{n^m}\binom{n}{2}\sum_{0\le s<m}\sum_{r\ge t}\binom{s}{r}(n-1)^{s-r}n^{m-s-2}(m-s-1).$$

Here s respresents $j-1$ in the terminology of the above answer, and r is the number of entries in $a_1, a_2, \ldots, a_s$ which equal a_j. This formula can be simplified, e.g. by writing generating functions which correspond to it, until we arrive at Eq. (13). Is there a simpler way yet to give the answer? Apparently not, since the generating function is

$$\sum_m E_{tm}z^m = \binom{n}{2}\frac{1}{(1-nz)^3}\left(\frac{z}{1-(n-1)z}\right)^t.$$

12. If $m = 2k$, the average is 2^{-2k} times

$$\binom{2k}{0}2k+\binom{2k}{1}(2k-1)+\cdots+\binom{2k}{k}k+\binom{2k}{k+1}(k+1)+\cdots+\binom{2k}{2k}2k.$$

The latter sum is

$$\binom{2k}{k}k+2\left(\binom{2k-1}{k}2k+\cdots+\binom{2k-1}{2k-1}2k\right)=\binom{2k}{k}k+4k\cdot\tfrac{1}{2}\cdot 2^{2k-1}.$$

A similar argument may be used when $m = 2k+1$. The answer is

$$\frac{m}{2}+\frac{m}{2^{m+1}}\binom{m}{m/2}$$

when m is even,

$$\frac{m}{2}+\frac{m}{2^m}\binom{m-1}{(m-1)/2}$$

when m is odd.

14. Let $k_j = n/m + \sqrt{n}\,x_j$. (This idea was suggested by N. G. de Bruijn.) Stirling's approximation implies that

$$n^{-m}\frac{m!}{k_1!\ldots k_n!}\max(k_1,\ldots k_n)$$
$$= \sqrt{2\pi}^{1-n} n^{n/2}\left(\frac{m}{n}+\sqrt{m}\max(x_1,\ldots,x_n)\right)$$
$$\times \exp\left(-\frac{n}{2}(x_1^2+\cdots+x_n^2)\right)\sqrt{m}^{1-m}\left(1+O\left(\frac{1}{\sqrt{m}}\right)\right),$$

when $k_1+\cdots+k_n = m$ and when the x's are uniformly bounded. The sum of the latter quantity over all nonnegative $k_1,\ldots,k_n$ satisfying this condition is an approximation to a Riemann integral; we may deduce that the asymptotic behavior of the sum is $a_n!(m/n) + c_n\sqrt{m} + O(1)$, where

$$a_n = \sqrt{2\pi}^{1-n} n^{n/2}\int_{x_1+\ldots+x_n=0}\exp\left(-\frac{n}{2}(x_1^2+\cdots+x_n^2)\right)dx_2\ldots dx_n,$$

$$c_n = \sqrt{2\pi}^{1-n} n^{n/2}\int_{x_1+\ldots+x_n=0}\max(x_1,\ldots,x_n)$$
$$\times \exp\left(-\frac{n}{2}(x_1^2+\cdots+x_n^2)\right)dx_2\ldots dx_n,$$

since it is possible to show that the corresponding sums come within ϵ of a_n and c_n for any ϵ.

We know that $a_n = 1$, since the corresponding sum can be evaluated explicitly. The integral which appears in the expression for c_n equals nI_1, where

$$I_1 = \int_{\substack{x_1+\ldots+x_n=0\\ x_1\geq x_2,\ldots,x_n}} x_1\exp\left(-\frac{n}{2}(x_1^2+\cdots+x_n^2)\right)dx_2\ldots dx_n.$$

We may make the substitution

$$x_1 = \frac{1}{n}(y_2+\cdots+y_n),\quad x_2 = x_1 - y_2,\quad x_3 = x_1 - y_3,\quad \ldots,\quad x_n = x_1 - y_n;$$

then we find $I_1 = I_2/n$, where

$$I_2 = \int_{y_2,\ldots,y_n\geq 0}(y_2+\cdots+y_n)\exp\left(-\frac{Q}{2}\right)dy_2\ldots dy_n,$$

$Q = n(y_2^2+\cdots+y_n^2) - (y_2+\cdots+y_n)^2$. Now by symmetry, I_2 is $(n-1)$ times

the same integral with $(y_2 + \cdots + y_n)$ replaced by y_2; hence $I_2 = (n-1)I_3$, where

$$I_3 = \int_{y_2,\ldots,y_n \geq 0} (ny_2 - (y_2 + \cdots + y_n)) \exp\left(-\frac{Q}{2}\right) dy_2 \ldots dy_n$$
$$= \int_{y_3,\ldots,y_n \geq 0} \exp\left(-\frac{Q_0}{2}\right) dy_3 \ldots dy_n;$$

here Q_0 is Q with y_2 replaced by zero. [When $n = 2$, let $I_3 = 1$.] Now let $z_j = \sqrt{n}\, y_j - (y_3 + \cdots + y_n)/(\sqrt{2} + \sqrt{n})$, $3 \leq j \leq m$. Then $Q_0 = z_3^2 + \cdots + z_n^2$, and we deduce that $I_3 = I_4/n^{(n-3)/2}\sqrt{2}$, where

$$I_4 = \int_{y_3,\ldots,y_n \geq 0} \exp\left(-\frac{z_3^2 + \cdots + z_n^2}{2}\right) dz_3 \ldots dz_n$$
$$= \alpha_n \int \exp\left(-\frac{z_3^2 + \cdots + z_n^2}{2}\right) dz_3 \ldots dz_n$$
$$= \alpha_n (\sqrt{2\pi})^{n-2},$$

where α_n is the "solid angle" in $(n-2)$-dimensional space which is spanned by the vectors $(n + \sqrt{2n}, 0, \ldots, 0) - (1, 1, \ldots, 1), \ldots, (0, 0, \ldots, n + \sqrt{2n}) - (1, 1, \ldots, 1)$, divided by the total solid angle of the whole space. Hence

$$c_n = \frac{(n-1)\sqrt{n}}{2\sqrt{\pi}}\, \alpha_n.$$

We have

$$\alpha_2 = 1, \quad \alpha_3 = \tfrac{1}{2}, \quad \alpha_4 = \frac{1}{\pi} \arctan \sqrt{2} \approx .304,$$

and

$$\alpha_5 = \frac{1}{8} + \frac{3}{4\pi} \arctan \frac{1}{\sqrt{8}} \approx .206.$$

[The value of c_3 was found by Robert M. Kozelka, *Annals of Math. Stat.* **27** (1956), 507–512, but the solution to this problem for higher values of n apparently has never appeared in the literature.]

17. Not unless the queues meet the restrictions which apply to the primitive method (4), (5).

SECTION 2.2.3

1. OVERFLOW is implicit in the operation P ⇐ AVAIL.

2.
```
INSERT  STJ   1F          Store location of "NOP T".
        STJ   9F          Store exit location.
        LD1   AVAIL       rI1 ⇐ AVAIL.
        J1Z   OVERFLOW
        LD3   0,1(LINK)
```

```
                ST3   AVAIL
                STA   0,1(INFO)     INFO(rI1) ← Y.
          1H    LD3   *(0:2)        rI3 ← LOC(T).
                LD2   0,3           rI2 ← T.
                ST2   0,1(LINK)     LINK(rI1) ← T.
                ST1   0,3           T ← rI1.
          9H    JMP   *                    ∎
```

3.
```
          DELETE  STJ   1F          Store location of "NOP T".
                  STJ   9F          Store exit location.
          1H      LD2   *(0:2)      rI2 ← LOC(T).
                  LD3   0,2         rI3 ← T.
                  J3Z   9F          Is T = Λ?
                  LD1   0,3(LINK)   rI1 ← LINK(T).
                  ST1   0,2         T ← rI1.
                  LDA   0,3(INFO)   rA ← INFO(rI1).
                  LD2   AVAIL       AVAIL ⇐ rI3.
                  ST2   0,3(LINK)
                  ST3   AVAIL
                  ENT3  2           Prepare for second exit.
          9H      JMP   *,3                  ∎
```

4.
```
     OVERFLOW  STJ    9F          Store setting of rJ.
               ST1    8F(0:2)     Save rI1 setting.
               LD1    POOLMAX
               INC1   C
               ST1    POOLMAX     Increment POOLMAX.
               CMP1   SEQMIN
               JGE    TOOBAD      Has storage been exceeded?
               ST1    AVAIL       Set AVAIL to new location.
               STZ    0,1(LINK)   Set LINK(AVAIL).
     9H        ENT1   *           Take rJ setting.
               DEC1   2           Subtract 2.
               ST1    *+2(0:2)    Store exit location.
     8H        ENT1   *           Restore rI1.
               JMP    *           Return. ∎
```

5. Inserting at the front is essentially like the basic insertion operation (8), with an additional test for empty queue: P ⇐ AVAIL, INFO(P) ← Y, LINK(P) ← F, if F = Λ then R ← P, F ← P.

To delete from the rear, we would have to find which node links to NODE(R), and that is necessarily inefficient since we have to search all the way from F. This could be done, for example, as follows:

a) If F = Λ then UNDERFLOW, otherwise set P ← LOC(F).
b) if LINK(P) ≠ R then set P ← LINK(P) and repeat this step until LINK(P) = R.
c) Set Y ← INFO(R), AVAIL ⇐ R, R ← P, LINK(P) ← Λ.

6. We could remove the operation LINK(P) ← Λ from (14), if we delete the commands "F ← LINK(P)" and "if F = Λ then set R ← LOC(F)" from (17); the latter are to be replaced by "if F = R then F ← Λ and R ← LOC(F), otherwise set F ← LINK(P)".

The effect of these changes is that the LINK field of the rear node in the queue will contain spurious information which is never interrogated by the program. A trick like this saves execution time and it is quite useful in practice, although it violates one of the basic assumptions of garbage collection (see Section 2.3.5) so it cannot be used in conjunction with such algorithms.

7. (Make sure your solution works for empty lists.)

I1. Set P ← FIRST, Q ← Λ.
I2. If P ≠ Λ, set R ← Q, Q ← P, P ← LINK(Q), LINK(Q) ← R, and repeat this step.
I3. Set FIRST ← Q. ∎

8.

```
        LD1   FIRST        1   P ≡ rI1 ← FIRST.
        ENT2  0            1   Q ≡ rI2 ← Λ.
        J1Z   2F           1   If list is empty, jump.
1H      ENTA  0,2          n   R ≡ rA ← Q.
        ENT2  0,1          n   Q ← P.
        LD1   0,2(LINK)    n   P ← LINK(Q).
        STA   0,2(LINK)    n   LINK(Q) ← R.
        J1NZ  1B           n   Is P ≠ Λ?
2H      ST2   FIRST        1   FIRST ← Q. ∎
```

The time is $(7n + 6)u$.

9. (a) Yes. (b) Yes if true parenthood is considered; no if legal parenthood is considered (a man's daughter might marry his father, as in the song "I'm My Own Grampa"). (c) No ($-1 \prec 1$ and $1 \prec -1$). (d) Let us hope so, or else there is a circular argument. (e) $1 \prec 3$ and $3 \prec 1$. (f) The statement is ambiguous. If we take the position that the subroutines called by y are dependent upon which subroutine calls y, we would have to conclude that the transitive law does not hold. (For example, a general input/output subroutine might call on different processing routines for each I/O device present, but usually not all these processing subroutines are needed in a single program. This is a problem that plagues many automatic programming systems.)

10. For (i) there are three cases: $x = y$; $x \subset y$ and $y = z$; $x \subset y$ and $y \subset z$. For (ii) there are two cases: $x = y$; $x \neq y$. Each of these subcases is handled trivially, as is (iii).

11. "Multiply out" the following to get all 52 solutions: $13749(25 + 52)86 + (1379 + 1397 + 1937 + 9137)(4258 + 4528 + 2458 + 5428 + 2548 + 5248 + 2584 + 5284)6 + (1392 + 1932 + 1923 + 9123 + 9132 + 9213)7(458 + 548 + 584)6$.

12. For example, (a) list all sets with k elements (in any order) before all sets with $k + 1$ elements, $0 \leq k < n$. (b) Represent a subset by a sequence of 0's and 1's showing which elements are in the set. This gives a correspondence between all subsets and (via the binary number system) the integers 0 through $2^n - 1$. The order of correspondence is a topological sequence.

14. If $a_1\, a_2 \ldots a_n$ and $b_1\, b_2 \ldots b_n$ are two possible topological sorts, let j be minimal such that $a_j \neq b_j$; then $a_k = b_j$ and $a_j = b_m$ for some $k, m > j$. Now $b_j \not\preceq a_j$ since $k > j$, and $a_j \not\preceq b_j$ since $m > j$, hence (iv) fails. Conversely if there is only one topological sort $a_1\, a_2 \ldots a_n$, we must have $a_j \preceq a_{j+1}$ for $1 \leq j < n$, since otherwise a_j and a_{j+1} could be interchanged. This and transitivity imply (iv).

Note: The following alternative proofs work also for infinite sets. (a) Every partial ordering can be embedded in a linear ordering. For if we have two elements

with $x_0 \not\preceq y_0$ and $y_0 \not\preceq x_0$ we can generate another partial ordering by the rule "$x \preceq y$ or $x \preceq x_0$ and $y_0 \preceq y$". The latter ordering "includes" the former and has $x_0 \preceq y_0$. Now apply Zorn's lemma or transfinite induction in the usual way to complete the proof. (b) Obviously a linear ordering cannot be embedded in any different linear ordering. (c) A partial ordering which has incomparable elements x_0 and y_0 as in (a) can be extended to two linear orderings in which $x_0 \preceq y_0$ and $y_0 \preceq x_0$, so at least two linear orderings exist.

15. If S is finite, we can list all relations $a \prec b$ that are true in the given partial ordering. By successively removing, one at a time, any relations that are implied by others, we arrive at an irredundant set. The problem is to show there is just one such set, no matter in what order we go about removing redundant relations. If there were two irredundant sets α and β, in which "$a \prec b$" appears in α but not in β, there are $k+1$ relations $a \prec c_1 \prec \cdots \prec c_k \prec b$ in β for some $k \geq 1$. But it is possible to deduce $a \prec c_1$ and $c_1 \prec b$ from α, *without* using the relation $a \prec b$ (since $b \not\preceq c_1$ and $c_1 \not\preceq a$), hence the relation $a \prec b$ is redundant in α.

The result is false for infinite sets S, when there is *at most* one irredundant set of relations. For example if S denotes the integers plus the element ∞ and we define $n \prec n+1$ and $n \prec \infty$ for all n, there is no irredundant set of relations which characterizes this partial ordering.

16. Let S be topologically sorted $x_{p_1}\, x_{p_2} \ldots x_{p_n}$ and apply this permutation to both rows and columns.

17. If k increases from 1 to n in step T4, the output is 1932745860. If k decreases from n to 1 in step T4, as it does in Program T, the output is 9123745860.

18. They link together the items in sorted order: `QLINK[0]` is the first, `QLINK[QLINK[0]]` is the second, and so on; `QLINK[last] = 0`.

19. This would fail in certain cases; when the queue contains only one element in step T5, this would set `F = 0` (thereby emptying the queue), but other entries could be placed in the queue in step T6. This modification would therefore require an additional test of `F = 0` in step T6.

20. At present, items are output only when they come to the front of the queue. A stack *could* be used, in the following way:

Step T4. Set `T` $\leftarrow 0$. For $1 \leq k \leq n$ if `COUNT[`k`]` is zero do the following: Output k, set `SLINK[`k`]` $\leftarrow$ `T`, `T` $\leftarrow k$. (`SLINK[`k`]` $\equiv$ `QLINK[`k`]`.)

Step T5. If `T = 0`, go to T8; otherwise set `P` $\leftarrow$ `TOP[T]`, `T` $\leftarrow$ `SLINK[T]`.

Step T6. Same as before, except when `COUNT[SUC(P)]` goes down to zero we output `SUC(P)`, set `SLINK[SUC(P)]` $\leftarrow$ `T` and `T` $\leftarrow$ `SUC(P)`.

Step T7. Go to T5.

Step T8. Output "0" and proceed as before. ∎

To compare these two algorithms, they are essentially the same except the output now occurs in three places, not in just one. This is less convenient when the machine implementation is considered, since the output is being packed into a buffer.

21. Repeated relations only make the algorithm a little slower and take up more space in the storage pool. A relation "$j < j$" would be treated like a loop (e.g. an arrow from a box to itself in the corresponding diagram).

22. To make the program "fail-safe" we should (a) check that $0 < n <$ (some appropriate maximum); (b) check each relation $j \prec k$ for the conditions $0 < j, k \leq n$; (c) make sure the number of relations doesn't overflow the storage pool area.

23. At the end of step T5, add "TOP[F] $\leftarrow \Lambda$". (Then at all times TOP[1], . . . , TOP[n] point to all the relations not yet cancelled.) In step T8, if N > 0, print "LOOP DETECTED IN INPUT:", and set QLINK[k] $\leftarrow 0$ for $1 \leq k \leq n$. Now add the following steps:

T9. For $1 \leq k \leq n$ set P $\leftarrow$ TOP[k], TOP[k] $\leftarrow 0$, and perform step T10. (This will set QLINK[j] to one of the predecessors of object j, for each j not yet output.) Then go to T11.

T10. If P $\neq \Lambda$, and QLINK[SUC(P)] $= 0$, set QLINK[SUC(P)] $\leftarrow k$. If P $\neq \Lambda$ set P $\leftarrow$ NEXT(P) and repeat this step.

T11. Find a k with QLINK[k] $\neq 0$.

T12. Set TOP[k] $\leftarrow 1$ and $k \leftarrow$ QLINK[k]. Now if TOP[k] $= 0$, repeat this step.

T13. (We have found the start of a loop.) Print the value of k, set TOP[k] $\leftarrow 0$, $k \leftarrow$ QLINK[k], and if TOP[k] $= 1$ repeat this step.

T14. Print the value of k (the beginning and end of the loop) and stop. (*Note:* The loop has been printed backwards; if it is desired to print the loop in forward order, an algorithm like that in Ex. 7 should be used between steps T12 and T13.) ∎

24. Insert three lines in the program of the text:

```
08a   PRINTER  EQU   18
14a            ST6   N
60a            STZ   X,1(TOP)        TOP[F] ← Λ.
```

Replace lines 75–77 by the following:

```
75             J6Z   DONE
76             OUT   LINE1(PRINTER)  Print indication of loop.
77             LD6   N
78             STZ   X,6(QLINK)      QLINK[k] ← 0.
79             DEC6  1
80             J6P   *-2             n ≥ k ≥ 1.
81             LD6   N
82    T9       LD2   X,6(TOP)        P ← TOP[k].
83             STZ   X,6(TOP)        TOP[k] ← 0.
84             J2Z   T9A             Is P = Λ?
85    T10      LD1   0,2(SUC)        rI1 ← SUC(P).
86             LDA   X,1(QLINK)
87             JANZ  *+2             If QLINK[rI1] = 0,
88             ST6   X,1(QLINK)        set it to k.
89             LD2   0,2(NEXT)       P ← NEXT(P).
90             J2P   T10             Is P ≠ Λ?
91    T9A      DEC6  1
92             J6P   T9              n ≥ k ≥ 1.
93    T11      INC6  1
94             LDA   X,6(QLINK)
95             JAZ   *-2             Find k with QLINK[k] ≠ 0.
```

```
96                  ENTX  1
97      T12         STX   X,6(TOP)          TOP[k] ← 1.
98                  LD6   X,6(QLINK)        k ← QLINK[k].
99                  LD1   X,6(TOP)
100                 J1Z   T12               Is TOP[k] = 0?
101     T13         ENTA  0,6
102                 CHAR                    Convert k to alpha.
103                 JBUS  *(PRINTER)
104                 STX   VALUE             Print.
105                 OUT   LINE2(PRINTER)
106                 J1Z   DONE              Stop when TOP[k] = 0.
107                 STZ   X,6(TOP)          TOP[k] ← 0.
108                 LD6   X,6(QLINK)        k ← QLINK[k].
109                 LD1   X,6(TOP)
110                 JMP   T13
111     LINE1       ALF   LOOP              Title line
112                 ALF   DETEC
113                 ALF   TED I
114                 ALF   N INP
115                 ALF   UT:
116     LINE2       ALF                     Succeeding lines
117     VALUE       EQU   LINE2+3
118                 ORIG  LINE2+24
119     DONE        HLT                     End of computation.
120     N           CON   0
121     X           EQU   *
122                 END   TOPSORT           ▮
```

Note: When the relations $10 \prec 1$, $6 \prec 10$, $1 \prec 9$ were added before the data (18), this program printed out "1,10,6,8,5,9,1" as the loop.

26. One solution is to proceed in two phases as follows:

Phase 1. (We use the X table as a (sequential) stack as we mark B = 1 or 2 for each subroutine that needs to be used.)

A0. For $1 \le J \le N$ set $B(X[J]) \leftarrow B(X[J]) + 2$, if $B(X[J]) \le 0$.
A1. If $N = 0$, go to phase 2; otherwise $P \leftarrow X[N]$ and decrease N by 1.
A2. If $|B(P)| = 1$, go to A1, otherwise set $P \leftarrow P + 1$.
A3. If $B(SUB1(P)) \le 0$, set $N \leftarrow N + 1$, $B(SUB1(P)) \leftarrow B(SUB1(P)) + 2$, $X[N] \leftarrow SUB1(P)$. If $SUB2(P) \neq 0$ and $B(SUB2(P)) \le 0$, do a similar set of actions with SUB2(P). Go to A2. ▮

Phase 2. (We go through the table and allocate memory.)

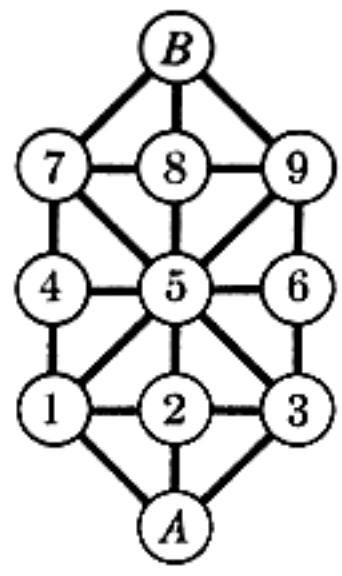
B1. Set $P \leftarrow FIRST$.
B2. If $P = \Lambda$, set $N \leftarrow N + 1$, $BASE(LOC(X[N])) \leftarrow -MLOC$, $SUB(LOC(X[N])) \leftarrow 0$, and terminate the algorithm.
B3. If $B(P) > 0$, set $N \leftarrow N + 1$, $BASE(LOC(X[N])) \leftarrow MLOC$, $SUB(LOC(X[N])) \leftarrow P$, $MLOC \leftarrow MLOC + SPACE(P)$.
B4. Set $P \leftarrow LINK(P)$ and return to B2. ▮

27. Comments on the following code are left to the reader.

```
B       EQU   0:1
SPACE   EQU   2:3
LINK    EQU   4:5
SUB1    EQU   2:3
SUB2    EQU   4:5
BASE    EQU   0:3
SUB     EQU   4:5
A0      LD2   N
        J2Z   B1
1H      LD3   X,2
        LDA   0,3(B)
        JAP   *+3
        INCA  2
        STA   0,3(B)
        DEC2  1
        J2P   1B
        LD1   N
A1      J1Z   B1
        LD2   X,1
        DEC1  1
A2      LDA   0,2(1:1)
        DECA  1
        JAZ   A1
        INC2  1
A3      LD3   0,2(SUB1)
        LDA   0,3(B)
        JAP   9F
        INC1  1
        INCA  2
        STA   0,3(B)
        ST3   X,1
9H      LD3   0,2(SUB2)
        J3Z   A2
        LDA   0,3(B)
        JAP   A2
        INC1  1
        INCA  2
        STA   0,3(B)
        ST3   X,1
        JMP   A2
B1      ENT2  FIRST
        LDA   MLOC
        JMP   1F
B3      LDX   0,2(B)
        JXNP  B4
        INC1  1
        ST2   X,1(SUB)
```

```
         ADD    0,2(SPACE)
1H       STA    X+1,1(BASE)
B4       LD2    0,2(LINK)
B2       J2NZ   B3
         ENNX   1
         STX    X+1,1(0:0)
         STZ    X+1,1(SUB)
```

28. We give here only a few comments related to the military game. Let A be the player with three men whose pieces start on nodes A13; let B be the other player. In this game, A must "trap" B, and if B can cause a position to be repeated for a second time we can consider him the winner. To avoid keeping the entire past history of the game as an integral part of the positions, however, we should modify the algorithm in the following way: Start by marking the positions 157–4, 789–B, 359–6 with B to move as "lost" and apply the suggested algorithm. Now the idea is for player A to move only to B's "lost" positions. But he must also take additional precautions against repeating prior moves. A "good" computer game-playing program will use a random number generator to select between several winning moves when more than one is present, so an obvious technique would be to have the computer playing A just choose randomly among those moves which take him to a "lost" position for B.



Board for "The Military Game."

But there are interesting situations which make this plausible procedure fail! For example, consider position 258–7 with A to move; this is a "won" position. From this position player A might try moving to 158–7 (which is a "lost" position for B, according to the algorithm). But then B plays to 158-B, and this forces A to play to 258–B, after which B plays back to 258–7; he has won, since the former position has been repeated! This example shows that the algorithm must be re-invoked after every move has been made, starting with each position that has previously occurred marked "lost" (if A is to move) or "won" (if B is to move).

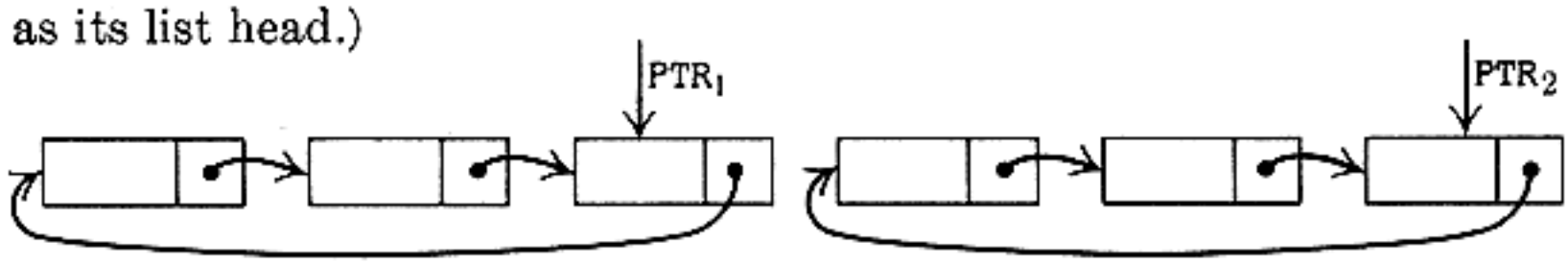
The author has found that this game makes a very satisfactory computer demonstration program.

29. (a) If **FIRST** = Λ, do nothing; otherwise set **P** $\leftarrow$ **FIRST**, and then repeatedly set **P** $\leftarrow$ **LINK(P)** zero or more times until **LINK(P)** = Λ. Finally set **LINK(P)** $\leftarrow$ **AVAIL** and **AVAIL** $\leftarrow$ **FIRST** (and probably also **FIRST** $\leftarrow \Lambda$). (b) If **F** = Λ, do nothing; otherwise set **LINK(R)** $\leftarrow$ **AVAIL** and **AVAIL** $\leftarrow$ **F** (and probably also **F** $\leftarrow \Lambda$, **R** $\leftarrow$ **LOC(F)**).

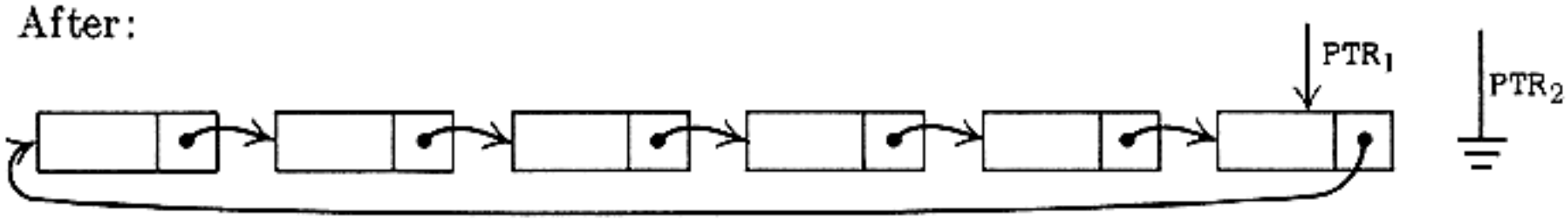
SECTION 2.2.4

1. No it does not help, it seems to hinder (if anything). (The stated convention is *not* especially consistent with the circular list philosophy, unless we put NODE(LOC(PTR)) into the list as its list head.)

2. Before:



After:



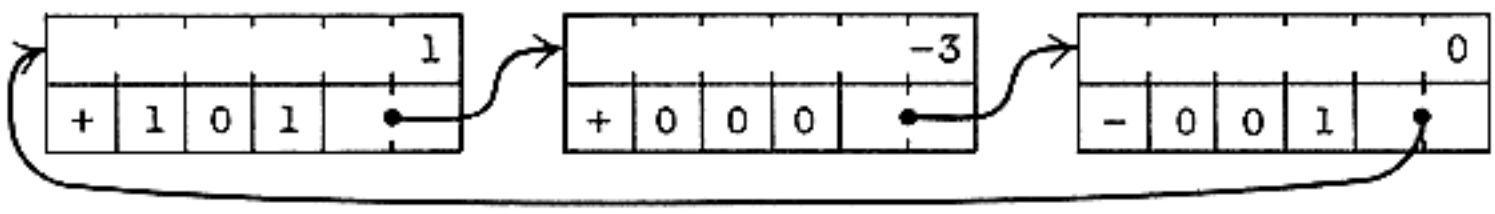
3. If $PTR_1 = PTR_2$, the only effect is $PTR_2 \leftarrow \Lambda$. If $PTR_1 \neq PTR_2$, the exchange of links breaks the list into two parts, as if a circle had been broken in two by cutting at two points; the second part of the operation then makes PTR_1 point to a circular list that consists of the nodes that would have been traversed if, in the original list, we followed the links from PTR_1 to PTR_2.

4. Let HEAD be the address of the list head. To push down Y onto the stack: set P ⇐ AVAIL, INFO(P) ← Y, LINK(P) ← LINK(HEAD), LINK(HEAD) ← P. To pop up the stack onto Y: if LINK(HEAD) = HEAD then UNDERFLOW, otherwise set P ← LINK(HEAD), LINK(HEAD) ← LINK(P), Y ← INFO(P), AVAIL ⇐ P.

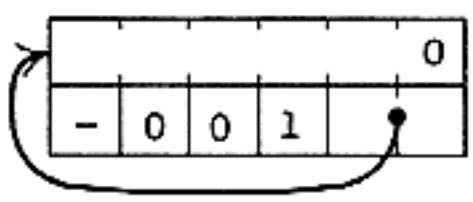
5. If PTR ≠ Λ, start with P ← PTR, Q ← LINK(PTR); then set R ← LINK(Q), LINK(Q) ← P, P ← Q, Q ← R, and repeat this operation until P = PTR.

6.

a)



b)



7. Matching terms in the polynomial are located in one pass over the list, instead of requiring repeated random searches. Also, *increasing* order would be incompatible with the "−1" sentinel.

8. We must know what node points to the current node of interest, if we are going to delete that node or to insert another one ahead of it. There are alternatives, however: we could delete NODE(Q) by setting Q2 ← LINK(Q) and then setting NODE(Q) ← NODE(Q2), AVAIL ⇐ Q2; we could insert a NODE(Q2) in front of NODE(Q) by first interchanging NODE(Q2) ↔ NODE(Q), then setting LINK(Q) ← Q2, Q ← Q2. These clever tricks allow the deletion and insertion *without* knowing which node links to NODE(Q); they were used in early versions of IPL. But they have the disadvantage that the sentinel node at the end of a polynomial will occasionally move, and other link variables may be pointing to this node.

9. Algorithm A with P = Q simply doubles polynomial(Q), as it should. Algorithm M with P = M also gives the expected result. Algorithm M with P = Q sets polynomial(P) ← polynomial(P) times $(1 + t_1)(1 + t_2) \cdots (1 + t_k)$ if M = $t_1 + t_2 + \cdots + t_k$ (although this is not immediately obvious). When M = Q, Algorithm M surprisingly gives the expected result, polynomial(Q) ← polynomial(Q) + polynomial(Q) × polynomial (P), except that the computation blows up when the constant term of polynomial (P) is −1.

10. No changes at all. The only possible difference would be in step M2, removing error checks that A, B, or C might individually overflow (these error checks have not been specified because we have assumed they are not necessary). In other words, the algorithms in this section may be regarded as operations on $f(x^{b^2}, x^b, x)$ instead of on $f(x, y, z)$.

11.

```
COPY  STJ   9F          (comments are left
      ENT3  9F            to the reader)
      LDA   1,1
1H    LD6   AVAIL
      J6Z   OVERFLOW
      LDX   1,6(LINK)
      STX   AVAIL
      STA   1,6
      LDA   0,1
      STA   0,6
      ST6   1,3(LINK)
      ENT3  0,6
      LD1   1,1(LINK)
      LDA   1,1
      JANN  1B
      LD2   8F
      ST2   1,3(LINK)
9H    JMP   *
8H    CON   0
```

12. Let the polynomial copied have p terms. Program A takes $(29p + 13)u$, and to make it a fair comparison we should add the time to create a zero polynomial, e.g. $18u$ with Ex. 14. The program of Ex. 11 takes $(21p + 31)u$, about 75% as much.

13.

```
ERASE  STJ   9F
       LDX   AVAIL
       LDA   1,1(LINK)
       STA   AVAIL
       STX   1,1(LINK)
9H     JMP   *
```

14.

```
ZERO  STJ   9F
      LD1   AVAIL
      J1Z   OVERFLOW
      LDX   1,1(LINK)
      STX   AVAIL
      ENT2  0,1
      MOVE  1F(2)
```

```
                    ST2   1,2(LINK)
          9H        JMP   *
          1H        CON   0
                    CON   -1(ABC)
```

15.

```
          MULT  STJ   9F          Entrance to subroutine
                LDA   5F          Change settings of switches.
                STA   SW1
                LDA   6F
                STA   SW2
                STA   SW3
                JMP   *+2
          2H    JMP   ADD         M2. Multiply cycle.
          1H    LD4   1,4(LINK)   M1. Next multiplier. M ← LINK(M).
                LDA   1,4
                JANN  2B          To M2 if ABC(M) ≥ 0.
          8H    LDA   7F          Restore settings of switches.
                STA   SW1
                LDA   8F
                STA   SW2
                STA   SW3
          9H    JMP   *           Return.
          5H    JMP   *+1         New setting of SW1
                LDA   0,1         COEF(P)
                MUL   0,4           × COEF(M) → rX.
                LDA   1,1(ABC)    ABC(P)
                JAN   *+2
                ADD   1,4(ABC)      + ABC(M), if ABC(P) ≥ 0
                SLA   2           Move into 0:3 field of rA.
                STX   0F          Save rX for use in SW2 and SW3.
                JMP   SW1+1
          6H    LDA   0F          New setting of SW2, SW3
          7H    LDA   1,1         Usual setting of SW1
          8H    LDA   0,1         Usual setting of SW2, SW3
          0H    CON   0           Temp storage
```

16. Let r be the number of terms in polynomial(M). The subroutine requires $21pr + 38r + 29 + 29\sum m' + 19\sum m'' + 29\sum p' + 8\sum q'$, where the latter summations refer to the corresponding quantities during the r activations of program A. The number of terms in polynomial(Q) goes up by $p' - m'$ each activation of program A. If we make the not unreasonable assumption that $m' = 0$ and $p' = \alpha p$ where $0 < \alpha < 1$, we get the respective sums equal to 0, $(1 - \alpha)pr$, αpr, and $rq'_0 + \alpha p\big(r(r-1)/2\big)$, where q'_0 is the value of q' in the first iteration. The grand total is $4\alpha pr^2 + 40pr + 6\alpha pr + 8q'_0 r + 38r + 29$. This analysis indicates that the multiplier ought to have fewer terms than the multiplicand, since we have to skip over unmatching terms in polynomial(Q) more often.

17. There actually is very little advantage; addition and multiplication routines with either type of list would be virtually the same. The efficiency of the ERASE subroutine (see Ex. 13) is apparently the only important difference.

18. Let the link field of node x_i contain $\texttt{LOC}(x_{i+1}) \oplus \texttt{LOC}(x_{i-1})$, where "$\oplus$" denotes either subtraction or "exclusive or." Two adjacent list heads are included in the circular list, to help get things started properly. (The origin of this ingenious technique is unknown.)

1. Insert Y at the left: P ⇐ AVAIL; INFO(P) ← Y; LLINK(P) ← Λ; RLINK(P) ← LEFT; if LEFT ≠ Λ then LLINK(LEFT) ← P else RIGHT ← P; LEFT ← P. Set Y to left and delete: if LEFT = Λ then UNDERFLOW; P ← LEFT; LEFT ← RLINK(P); if LEFT = Λ then RIGHT ← Λ else LLINK(LEFT) ← Λ; AVAIL ⇐ P.

2. Consider the case of several deletions (at the same end) in succession. After each deletion we must know what to delete next. This implies the links in the list point away from that end of the list. So deletion at both ends implies the links must go both ways.

3. To show the independence of CALLUP from CALLDOWN, notice for example that in Table 1 the elevator did not stop at floors 2 or 3 at time 0393–0444 although there were people waiting; these people had pushed CALLDOWN, but if they had pushed CALLUP the elevator would have stopped.

To show the independence of CALLCAR from the others, notice in Table 1 when the doors start to open at time 1398 the elevator has already decided to be GOINGUP. Its state would have been NEUTRAL at that point if CALLCAR[1] = CALLCAR[2] = CALLCAR[3] = CALLCAR[4] = 0, according to step E2, but in fact CALLCAR[2] and CALLCAR[3] have been set to 1 by men nos. 7 and 9 in the elevator. (If we envision the same situation with all floor numbers increased by 1, the fact that STATE = NEUTRAL or STATE = GOINGUP when the doors open would affect whether the elevator would perhaps continue to go downward or would unconditionally go upward.)

4. If a dozen or more people were getting out at the same floor, STATE might be NEUTRAL all during this time, and when E9 calls the DECISION subroutine this may set a new state before anyone has gotten in on the current floor. It happens very rarely indeed (and it certainly was the most puzzling phenomenon observed by the author during his elevator experiments).

5. The state from the time the doors start to open at time 1063 until man 7 gets in at time 1183 would have been NEUTRAL, since there would have been no calls to floor 0. Then man 7 would set CALLCAR[2] ← 1 and the state would correspondingly change to GOINGUP.

6. Add the condition "if OUT < IN then STATE ≠ GOINGUP; if OUT > IN then STATE ≠ GOINGDOWN" to the condition "FLOOR = IN" in steps M2 and M4. In step E4, accept men from QUEUE[FLOOR] only if they are headed in the elevator's direction, unless STATE = NEUTRAL (when we accept all comers); men from QUEUE[FLOOR] who have not been accepted should also push CALLUP or CALLDOWN again (since the state can change in step M5).

7. In line 227 this man is assumed to be in the WAIT list. Jumping to M4A makes sure he stays there. It is assumed that GIVEUPTIME is not zero, and indeed that it is probably 100 or more.

8. Comments are left to the reader.

```
277     E8   DEC4   1
278          ENTA   61
279          JMP    HOLDC
280          LDA    CALL,4(3:5)
281          JAP    1F
282          ENT1   -2,4
283          J1Z    2F
284          LDA    CALL,4(1:1)
285          JAZ    E8
286     2H   LDA    CALL-1,4
287          ADD    CALL-2,4
288          ADD    CALL-3,4
289          ADD    CALL-4,4
290          JANZ   E8
291     1H   ENTA   23
292          JMP    E2A
```

9.

```
01   DECISION  STJ    9F                  Store exit location.
02             J5NZ   9F                  D1. Decision necessary?
03             LDX    ELEV1+2(NEXTINST)
04             DECX   E1                  D2. Should door open?
05             JXNZ   1F                  Jump if elevator not at E1.
06             LDA    CALL+2
07             ENT3   E3                  Prepare to schedule E3,
08             JANZ   8F                    if there is a call on floor 2.
09   1H        ENT1   -4                  D3. Any calls?
10             LDA    CALL+4,1            Search for a nonzero call variable.
11             JANZ   2F
12   1H        INC1   1                   rI1 ≡ j − 4
13             J1NP   *-3
14             LDA    9F(0:2)             All CALL[j], j ≠ FLOOR, are zero
15             DECA   E6B                 Is exit location = line 250?
16             JANZ   9F
17             ENT1   -2                  Set j ← 2.
18   2H        ENT5   4,1                 D4. Set STATE.
19             DEC5   0,4                 STATE ← j − FLOOR.
20             J5NZ   *+2
21             JANZ   1B                  j = FLOOR not allowed in general.
22             JXNZ   9F                  D5. Elevator dormant?
23             J5Z    9F                  Jump if not at E1 or if j ≠ 2.
24             ENT3   E6                  Otherwise schedule E6.
25   8H        ENTA   20                  Wait 20 units of time.
26             ST6    TEMPD               Save rI6.
27             ENT6   ELEV1
28             ST3    2,6(NEXTINST)       Set NEXTINST to E3 or E6.
29             JMP    HOLD                Schedule the activity.
30             LD6    TEMPD               Restore rI6.
31   9H        JMP    *                   Exit from subroutine.
```

11. Initially let LINK[k] = 0, $1 \le k \le n$, and HEAD = 0. During a simulation step that changes V[k], give an error indication if LINK[k] $\neq$ 0; otherwise set LINK[k] $\leftarrow$ HEAD, HEAD $\leftarrow k$ and set NEWV[k] to the new value of V[k]. After each simulation step, set $k \leftarrow$ HEAD, HEAD $\leftarrow$ 0, and do the following operation repeatedly zero or more times until $k = 0$: set V[k] $\leftarrow$ NEWV[k], $t \leftarrow$ LINK[k], LINK[k] $\leftarrow$ 0, $k \leftarrow t$.

Clearly this method is readily adapted to the case of scattered variables, if we include a NEWV and LINK field in each node associated with a variable field V.

12. The WAIT list has deletions from the left to the right, but insertions are sorted in from the right to the left (since the search is likely to be shorter from that side). Also we delete nodes from all three lists in several places when we do not know the predecessor or successor of the node being deleted. Only the ELEVATOR list could be converted to a one-way list, without much loss of efficiency.

Note: It may be preferable to use a non-linear list as the WAIT list in a discrete simulator, to reduce the time for "sorting in". See the "heap sort" algorithms of Chapter 5, for a method that maintains a "smallest in first out" table using only $O(\log n)$ operations when n elements are present. (There is no need for such a fancy method when n is known to be small.) This idea is due to R. W. Floyd.

SECTION 2.2.6

1. (Note that the indices run from 1 to n, not from 0 to n as in Eq. (5).) $\texttt{LOC(A[0, 0])} + 2nJ + 2K = \texttt{LOC(A[J, K])}$, where A[0, 0] is an assumed node that is actually non-existent. If we set $\texttt{J} = \texttt{K} = 1$, we get $\texttt{LOC(A[0, 0])} + 2n + 2 = \texttt{LOC(A[1, 1])}$, so the answer can be expressed in several ways. LOC(A[0, 0]) might be negative.

2. $$\texttt{LOC(A[I}_1, \ldots, \texttt{I}_k\texttt{])} = \texttt{LOC(A[0, \ldots, 0])} + \sum_{1\le r\le k} a_r\texttt{I}_r$$
$$= \texttt{LOC(A[}l_1, \ldots, l_k\texttt{])} - \sum_{1\le r\le k} a_r l_r + \sum_{1\le r\le k} a_r\texttt{I}_r,$$

where $a_r = \prod_{r<s\le k} (u_r - l_r + 1)$.

Note: For a generalization to the structures occurring in the COBOL and PL/I languages, and a simple algorithm to compute the relevant constants, see P. Deuel, *CACM* **9** (1966), 344–347.

3. $1 \le k \le j \le n$ if and only if $0 \le k - 1 \le j - 1 \le n - 1$; so replace k, j, n respectively by $k - 1$, $j - 1$, $n - 1$ in all formulas derived for lower bound zero.

4. $\texttt{LOC(A[J, K])} = \texttt{LOC(A[0, 0])} + (n + 1)\texttt{J} - \texttt{J(J} - 1)/2 + \texttt{K}$.

5. Let $\texttt{A0} = \texttt{LOC(A[0, 0])}$. There are at least two solutions, assuming J is in rI1 and K is in rI2: (1) "LDA TA2,1:7", where location TA2+J is "NOP J+1*J/2+A0,2"; (2) "LDA C1,7:2", where location C1 contains "NOP TA,1:7" and location TA+J says "NOP J+1*J/2+A0". The latter takes one more cycle but doesn't tie the table down to index register 2.

6. (a) $$\texttt{LOC(A[I, J, K])} = \texttt{LOC(A[0, 0, 0])} + \binom{\texttt{I}+2}{3} + \binom{\texttt{J}+1}{2} + \binom{\texttt{K}}{1}.$$

(b) $$\texttt{LOC(B[I, J, K])} = \texttt{LOC(B[0, 0, 0])}$$
$$+ \binom{n+3}{3} - \binom{n+3-\texttt{I}}{3} + \binom{n+2-\texttt{I}}{2} - \binom{n+2-\texttt{I}-\texttt{J}}{2} + \texttt{K};$$

the expansion of this polynomial contains a term in $\texttt{I} \cdot \texttt{J}$; hence it cannot be expressed in the stated form. A tetrahedral array like B is *not* best stored in lexicographic order.

7. $\texttt{LOC(A[I}_1,\ldots,\texttt{I}_k\texttt{])} = \texttt{LOC(A[0},\ldots,\texttt{0])} + \sum_{1\le r\le k}\binom{\texttt{I}_r+k-r}{1+k-r}$. See Ex. 1.2.6–56.

8. Let X[I, J, K] be defined for $0 \le \texttt{I} \le n+1$, $0 \le \texttt{J} \le n+1$, $0 \le \texttt{K} \le n+2$. We can let A[I, J, K] = X[I, J, K], B[I, J, K] = X[J, I + 1, K], C[I, J, K] = X[I + 1, K, J + 1], D[I, J, K] = X[K, I + 1, J + 1], E[I, J, K] = X[J, K, I + 1], F[I, J, K] = X[K, J + 1, I + 2]. Clearly each of these definitions fits within the confines of the X array; we must prove there is no overlap, and that is easily done if we rename the indices of X as i, j, k in each of the six definitions; then we find respectively $i \ge j \ge k$, $j > i \ge k$, $i \ge k > j$, $j \ge k > i$, $k > i \ge j$, $k > j > i$ and these conditions prove there is no overlap. However, the above method does not leave the X array filled: the elements X[$n+1$, J, K] and X[J, K, $n+2$] are unused for $0 \le \texttt{K} \le \texttt{J} \le n+1$. When $n \ge 2$, it appears to be impossible to pack any tighter than this and still retain linear addressing in each index, although the author has not proved this. Some indices might run negatively. The problem is trivial and of no interest when $n = 0$ or 1.

The above method uses locations X[0, 0, 0] through X[$n+1$, n, $n+1$] minus a few holes in between. The arrays, which contain $(n+1)(n+2)(n+3)$ elements in all, have been packed into $(n+2)(n+2)(n+3) - (n+4)$ consecutive cells. Is there a better solution?

9. G1. Set pointer variables P1, P2, P3, P4, P5, P6 to the first locations of the lists FEMALE, A21, A22, A23, BLOND, BLUE, respectively. Assume in what follows that the end of each list is given by link Λ, and Λ is smaller than any other link. If P6 = Λ, stop (the list, unfortunately, is empty).

G2. (Many possible orderings of the following actions could be done; we have chosen to examine EYES first, then HAIR, then AGE, then SEX.) Set P5 ← HAIR(P5) zero or more times until P5 ≤ P6. If now P5 < P6, go to step G5.

G3. Set P4 ← AGE(P4) repeatedly if necessary until P4 ≤ P6. Similarly do the same to P3 and P2 until P3 ≤ P6 and P2 ≤ P6. If now P4, P3, P2 are all smaller than P6, go to G5.

G4. Set P1 ← SEX(P1) until P1 ≤ P6. If P1 = P6, we have found one of the desired girls, so output her address, P6. (Her age can be determined from the settings of P2, P3, and P4.)

G5. Set P6 ← EYES(P6). Now stop if P6 = Λ; otherwise return to G2. ▮

This algorithm is interesting but not the best way to organize a list for such a search.

10. After trying out many different seemingly efficient schemes and analyzing their efficiency, the author feels there seems to be no better way than to divide all people into n approximately equal groups, where n is as large as possible based on the amount of space available, in such a way that a person's characteristics determine the group he is in; then search every person in the appropriate group for the desired characteristics. (See also Chapter 6.)

11. At most $200 + 200 + 3 \cdot 4 \cdot 200 = 2800$ words.

12. VAL(Q0) = c, VAL(P0) = b/a, VAL(P1) = d.

13. It is convenient to have at the end of each list a sentinel which "compares low" in some field on which the list is ordered. A straight one-way list *could* have been used, for example by retaining just the LEFT links in BASEROW[i] and the UP links in

BASECOL[j], by modifying Algorithm S thus: S2, test if P0 = Λ before setting J ← COL(P), and if so set P0 ← LOC(BASEROW[I0]) and go to S3. S3, test if Q0 = Λ and if so, terminate. S4, analogous to changes in S2. S5, test if P1 = Λ and if so treat this as if COL(P1) < 0. S6, test if UP(PTR[J]) = Λ and if so treat as if its ROW field were negative.

These modifications make the algorithm more complicated and save no storage space except a ROW or COL field in the list heads (which in the case of MIX is no saving at all).

14. One could first link together those columns which have a nonzero element in the pivot row, so that all other columns could be skipped as we pivot on each row. Rows in which the pivot column is zero are skipped over immediately.

15. Let rI1 ≡ PIVOT, J; rI2 ≡ P0; rI3 ≡ Q0; rI4 ≡ P; rI5 ≡ P1, X; LOC(BASEROW[i]) ≡ BROW + i; LOC(BASECOL[j]) ≡ BCOL + j; PTR[j] ≡ BCOL + j(1:3).

```
01    ROW         EQU   0:3
02    UP          EQU   4:5
03    COL         EQU   0:3
04    LEFT        EQU   4:5
05    PTR         EQU   1:3
06    PIVOTSTEP   STJ   9F          Subroutine entrance, rI1 = PIVOT
07    S1          LD2   0,1(ROW)
08                ST2   I0          I0 ← ROW(PIVOT).
09                LD3   1,1(COL)
10                ST3   J0          J0 ← COL(PIVOT).
11                LDA   =1.0=       Floating point constant 1
12                FDIV  2,1
13                STA   ALPHA       ALPHA ← 1/VAL(PIVOT).
14                LDA   =1.0=
15                STA   2,1         VAL(PIVOT) ← 1.
16                ENT2  BROW,2      P0 ← LOC(BASEROW([I0]).
17                ENT3  BCOL,3      Q0 ← LOC(BASECOL([J0]).
18                JMP   S2
19    2H          ENTA  BCOL,1
20                STA   BCOL,1(PTR) PTR[J] ← LOC(BASECOL([J]).
21                LDA   2,2
22                FMUL  ALPHA
23                STA   2,2         VAL(P0) ← ALPHA × VAL(P0).
24    S2          LD2   1,2(LEFT)   P0 ← LEFT(P0).
25                LD1   1,2(COL)    J ← COL(P0).
26                J1NN  2B          If J < 0, process J.
27    S3          LD3   0,3(UP)     Q0 ← UP(Q0).
28                LD4   0,3(ROW)    rI4 ← ROW(Q0).
29    9H          J4N   *           If rI4 < 0, exit.
30                CMP4  I0
31                JE    S3          If rI4 = I0, repeat.
32                ST4   I(ROW)      I ← rI4.
33                ENT4  BROW,4      P ← LOC(BASEROW[I]).
34    S4A         LD5   1,4(LEFT)   P1 ← LEFT(P).
35    S4          LD2   1,2(LEFT)   P0 ← LEFT(P0).
```

```
36                  LD1   1,2(COL)      J ← COL(P0).
37                  CMP1  J0
38                  JE    S4            Repeat if J = J0.
39                  ENTA  0,1
40                  SLA   2             rA(0:3) ← J.
41                  J1NN  S5
42                  LDAN  2,3           If J < 0,
43                  FMUL  ALPHA         set VAL(Q0) ← —ALPHA × VAL(Q0).
44                  STA   2,3
45                  JMP   S3
46        1H        ENT4  0,5           P ← P1.
47                  LD5   1,4(LEFT)     P1 ← LEFT(P).
48        S5        CMPA  1,5(COL)
49                  JL    1B            Loop until COL(P1) ≤ J.
50                  JE    S7            If =, go right to S7.
51        S6        LD5   BCOL,1(PTR)   rI5 ← PTR[J].
52                  LDA   I             rA(0:3) ← I.
53        2H        ENT6  0,5           rI6 ← rI5.
54                  LD5   0,6(UP)       rI5 ← UP(rI6).
55                  CMPA  1,5(COL)
56                  JL    2B            Jump if COL(rI5) ← I.
57                  LD5   AVAIL         X ⇐ AVAIL.
58                  J5Z   OVERFLOW
59                  LDA   0,5(UP)
60                  STA   AVAIL
61                  LDA   0,6(UP)       UP(PTR[J])
62                  STA   0,5(UP)       → UP(X).
63                  LDA   1,4(LEFT)     LEFT(P)
64                  STA   1,5(LEFT)     → LEFT(X).
65                  ST1   1,5(COL)      COL(X) ← J.
66                  LDA   I(ROW)
67                  STA   0,5(ROW)      ROW(X) ← I.
68                  STZ   2,5           VAL(X) ← 0.
69                  ST5   1,4(LEFT)     LEFT(P) ← X.
70                  ST5   0,6(UP)       UP(PTR[J]) ← X.
71                  ST5   BCOL,1(PTR)   PTR[J] ← X ≡ P1.
72        S7        LDAN  2,3           —VAL(Q0)
73                  FMUL  2,2             × VAL(P0)
74                  FADD  2,5               + VAL(P1)
75                  JAZ   S8            If significance lost, to S8.
76                  STA   2,5           Otherwise store in VAL(P1).
77                  ST5   BCOL,1(PTR)   PTR[J] ← P1.
78                  ENT4  0,5           P ← P1.
79                  JMP   S4A           P1 ← LEFT(P), to S4.
80        S8        LD6   BCOL,1(PTR)   rI6 ← PTR[J].
81                  JMP   *+2
82                  LD6   0,6(UP)       rI6 ← UP(rI6).
83                  LDA   0,6(UP)
84                  DECA  0,5           Is UP(rI6) = P1?
```

```
85    JANZ   *-3          Loop until equal.
86    LDA    0,5(UP)
87    STA    0,6(UP)      UP(rI6) ← UP(P1).
88    LDA    1,5(LEFT)
89    STA    1,4(LEFT)    LEFT(P) ← LEFT(P1).
90    LDA    AVAIL        AVAIL ⇐ P1.
91    STA    0,5(UP)
92    ST5    AVAIL
93    JMP    M4A          P1 ← LEFT(P), to S4. ∎
```

Note: Using the conventions of Chapter 4, lines 72–75 would actually be coded

```
LDA 2,3; FMUL 2,2; FCMP 2,5; JE S8; STA TEMP; LDA 2,5; FSUB TEMP;
```

with a suitable parameter EPSILON in location zero.

18. $k = 1$, pivot column 3, we obtain

$$\begin{pmatrix} \frac{1}{3} & \frac{2}{3} & \frac{1}{3} \\ -\frac{2}{3} & -\frac{1}{3} & -\frac{2}{3} \\ -\frac{1}{3} & -\frac{2}{3} & -\frac{1}{3} \end{pmatrix};$$

$k = 2$, pivot column 1, we obtain

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ -\frac{3}{2} & \frac{1}{2} & 1 \\ -\frac{1}{2} & -\frac{1}{2} & 0 \end{pmatrix};$$

$k = 3$, pivot column 2, we obtain

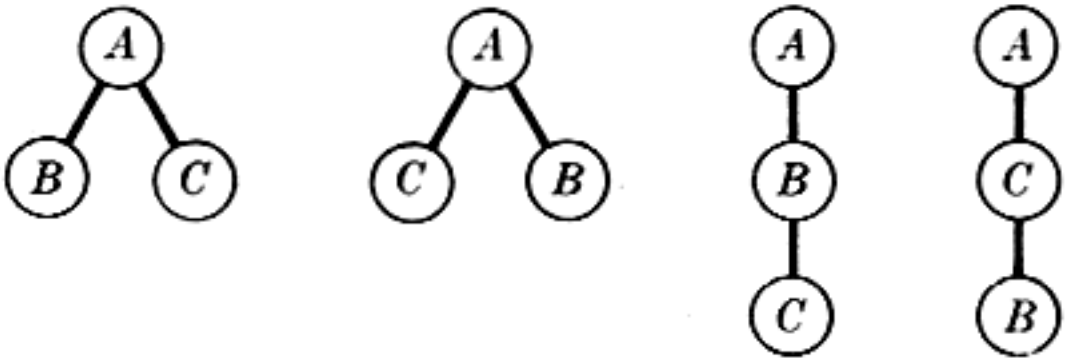
$$\begin{pmatrix} 0 & 1 & 0 \\ -2 & 1 & 1 \\ 1 & -2 & 0 \end{pmatrix};$$

after the final permutations, we have the answer

$$\begin{pmatrix} 1 & -2 & 1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix}.$$

SECTION 2.3

1. There are three ways to choose the root. Once the root has been chosen, say A, there are three ways to partition the remaining nodes into subtrees: $\{B\}, \{C\}$; $\{C\}, \{B\}$; $\{B, C\}$. In the latter case there are two ways to make $\{B, C\}$ into a tree, depending on which is the root. Hence get four trees when A is the root:



and 12 in all. This problem is solved for any number of nodes in Section 2.3.4.4.

2. The first two trees in the answer to Ex. 1 are the same, as oriented trees, so we get only 9 different possibilities in this case. For the general solution, see Section 2.3.4.4.

3. Part 1: To show there is *at least one* such sequence. Let the tree have n nodes. The result is clear when $n = 1$, since X must be the root. If $n > 1$, the definition implies there is a root X_1 and subtrees $T_1, T_2, \ldots, T_m$; either $X = X_1$ or X is a member of a unique T_j. In the latter case, there is by induction a path $X_2, \ldots, X$ where X_2 is the root of T_j, and since X_1 is the father of X_2 we have a path $X_1, X_2, \ldots, X$.

Part 2: To show there is *at most one* such sequence. We will prove by induction that if X is not the root of the tree, X has a unique father (so that X_k determines X_{k-1} determines X_{k-2}, etc.). If the tree has one node, there is nothing to prove; otherwise X is in a unique T_j. Either X is the root of T_j, in which case X has a unique father by definition; or X is not the root of T_j, in which case X has a unique father in T_j by induction, and no node outside of T_j can be X's father.

4. True (unfortunately). A similar situation is the popular expression "She is a star of the nth magnitude." According to the way astronomers give magnitudes to stars, a much better compliment would be "She is a star of the *zero*th magnitude."

5. 4.

6. Let $\text{father}^0(X)$ denote X, $\text{father}^1(X)$ denote X's father, $\text{father}^2(X) = \text{father}(\text{father}(X)) = X$'s grandfather, $\text{father}^k(X) = \text{father}(\text{father}^{k-1}(X)) = X$'s "$(\text{great})^{k-2}$-grandfather." The cousinship condition is that $\text{father}^{m+1}(X) = \text{father}^{m+n+1}(Y)$ but $\text{father}^m(X) \neq \text{father}^{m+n}(Y)$; or, if $n > 0$, possibly the same condition with X, Y interchanged.

7. We go to an unsymmetric relation between X and Y; the condition of Ex. 6 is used, with the convention that $\text{father}^j(X) \neq \text{father}^k(Y)$ if either j or k (or both) is -1. To show that this relation is always valid for some unique m and n, consider the Dewey decimal notation for X and Y, namely $1 . a_1 . \cdots . a_p . b_1 . \cdots . b_q$ and $1 . a_1 . \cdots . a_p . c_1 . \cdots . c_r$, where $p \geq 0$, $q \geq 0$, $r \geq 0$ and (if $qr \neq 0$) $b_1 \neq c_1$. The numbers of any pair of nodes can be written in this form, and clearly we must take $m = q - 1$ and $n + m = r - 1$.

8. The empty binary tree (i.e. the binary tree having no nodes). A "tree" always has at least one node, its root, while a "binary tree" may be vacuous.

9. A is the root, since we conventionally put the root at the top.

10. Any *finite* collection of nested sets corresponds to a forest as defined in the text, as follows: Let $A_1, \ldots, A_n$ be the sets of the collection that are contained in no other. For fixed j, the sub-collection of all sets contained in A_j is nested, and therefore we may assume this sub-collection corresponds to a tree (unordered) with A_j as the root.

11. In a nested collection $\mathcal{C}$ let $X \equiv Y$ if there is some $Z \in \mathcal{C}$ such that $X \cup Y \subseteq Z$. This relation is obviously reflexive and symmetric, and it is in fact an equivalence relation since $W \equiv X$, $X \equiv Y$ implies there are Z_1, Z_2 in $\mathcal{C}$ with $W \subseteq Z_1$, $X \subseteq Z_1 \cap Z_2$, $Y \subseteq Z_2$. Since $Z_1 \cap Z_2 \neq \emptyset$, either $Z_1 \subseteq Z_2$ or $Z_2 \subseteq Z_1$, hence $W \cup Y \subseteq Z_1 \cup Z_2 \in \mathcal{C}$. Now if $\mathcal{C}$ is a nested collection, define an oriented forest corresponding to $\mathcal{C}$ by the rule "X is an ancestor of Y, and Y is a descendant of X, if and only if $X \supset Y$." Each equivalence class of $\mathcal{C}$ corresponds to an oriented tree, which is an oriented forest with

$X \equiv Y$ for all X, Y. (We thereby have generalized the definitions of forest and tree which were given for finite collections.) In these terms, we may define the *level* of X as the cardinal number of $(\{X\} \cup \text{ancestors}(X))$. The *degree* of X is the cardinal number of equivalence classes in the nested collection descendants(X). We say X is the *father* of Y, and Y is a *son* of X, if X is an ancestor of Y but there is no Z such that $X \supset Z \supset Y$. To get *ordered* trees and forests, order the equivalence classes mentioned above in some *ad hoc* manner, for example by embedding the relation $\subseteq$ into linear order.

Example (a): Let $S_{\alpha k} = \{x \mid x = .d_1d_2d_3 \ldots$ in decimal notation, where $\alpha = .e_1e_2e_3 \ldots$ in decimal notation, and $d_j = e_j$ if $j \bmod 2^k \neq 0\}$. The collection $\mathcal{C} = \{S_{\alpha k} \mid k \geq 0, 0 < \alpha < 1\}$ is nested, and gives a tree with infinitely many levels and uncountable degree for each node.

Example (b), (c): It is convenient to define this set in the plane, instead of in terms of real numbers, and this is sufficient since there is a one-to-one correspondence between the plane and the real numbers. Let $S_{\alpha mn} = \{(\alpha, \beta) \mid m/2^n \leq \beta < (m+1)/2^n\}$, and let $T_\alpha = \{(\gamma, \beta) \mid \gamma \leq \alpha\}$. The collection $\mathcal{C} = \{S_{\alpha mn} \mid 0 < \alpha < 1, n \geq 0, 0 \leq m < 2^n\} \cup \{T_\alpha \mid 0 < \alpha < 1\}$ is easily seen to be nested. The sons of $S_{\alpha mn}$ are $S_{\alpha(2m)(n+1)}$ and $S_{\alpha(2m+1)(n+1)}$, and T_α has the son $S_{\alpha 00}$ plus the subtree $\{S_{\gamma mn} \mid \gamma < \alpha\} \cup \{T_\gamma \mid \gamma < \alpha\}$. So each node has degree 2, and each node has uncountably many ancestors of the form T. This construction is due to R. Bigelow.

Note: If we take a suitable well-ordering of the real numbers, and if we define $T_\alpha = \{(\gamma, \beta) \mid \gamma \succ \alpha\}$, we can improve this construction slightly, obtaining a nested collection where each node has degree 2, uncountable level, *and* 2 sons.

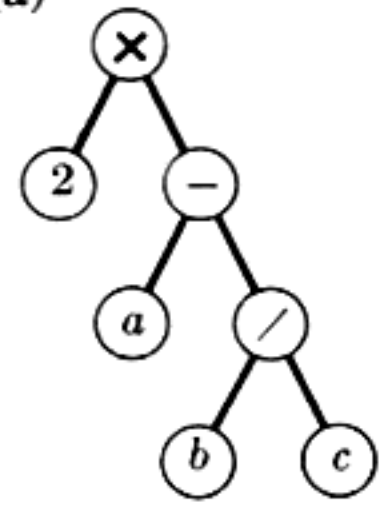
12. We impose an additional condition on the partial ordering (analogous to that of "nested sets") to ensure that it corresponds to a forest: If $x \preceq y$ and $x \preceq z$ then either $y \preceq z$ or $z \preceq y$. To make a tree, also assert the existence of a node r such that $x \succeq r$ for all x. A proof that this gives an unordered tree as defined in the text, when the number of nodes is finite, runs like the proof for nested sets in Ex. 10.

13. a_1, $a_1.a_2$, $\cdots$, $a_1.a_2.\cdots.a_k$.

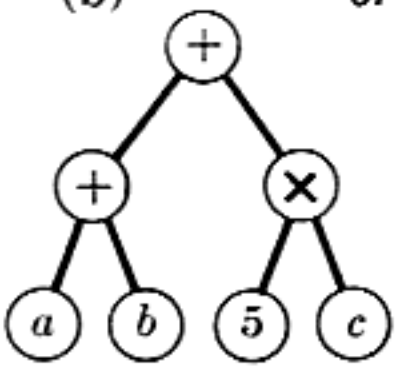
14. Since S is nonempty, it contains an element "$1.a_1.\cdots.a_k$" where k is as small as possible; if $k > 0$ we also take a_k as small as possible in S, and we immediately see that k must be 0, i.e. S contains the element "1". Let this be the root. All other elements have $k > 0$, and so the remaining elements of S can be partitioned into sets $S_j = \{1.j.a_2.\cdots.a_k\}$, $1 \leq j \leq m$, for some $m \geq 0$. If $m \neq 0$, and S_m is non-empty, by reasoning as above we find "$1.j$" is in S_j for each S_j, so that each S_j is non-empty. Then it is easy to see that the sets $S_j' = \{1.a_2.\cdots.a_k \mid 1.j.a_2.\cdots.a_k \text{ is in } S_j\}$ satisfy the same condition as S did, so by induction each of the S_j forms a tree also.

15. Let the root be "1" and let the root of the left subtree of α be $\alpha.0$; the root of the right subtree of α may be named $\alpha.1$. For example in Fig. 19(a), King Christian IX appears in two positions, 1.0.0.0.0 and 1.1.0.0.1.0. For brevity we may drop the decimal points, and write merely 10000 and 110010. *Note:* This notation is due to Francis Galton; see *Natural Inheritance* (Macmillan, 1889), 249. For "pedigrees", it is more mnemonic to use F and M in place of 0 and 1; e.g., Christian IX is Charles's $MFFMF$, i.e. Charles's mother's father's father's mother's father. The 0 and 1 convention is interesting for another reason as it gives us an important correspondence between nodes in a binary tree and positive integers expressed in the binary system (namely, memory addresses in a computer).

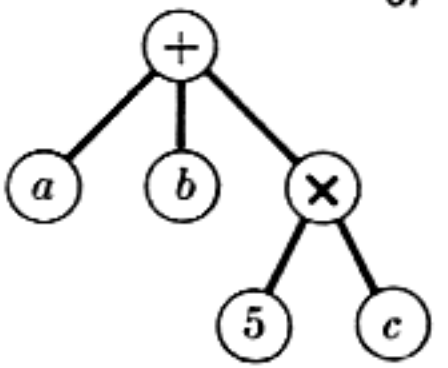
16. (a)



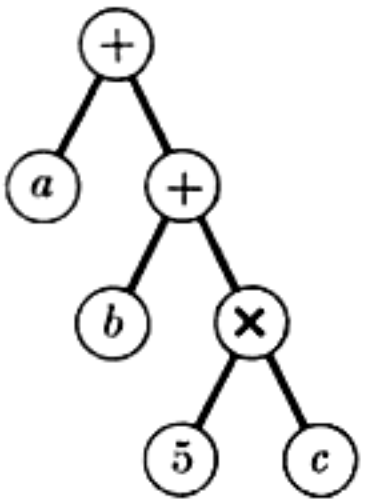
(b)



or



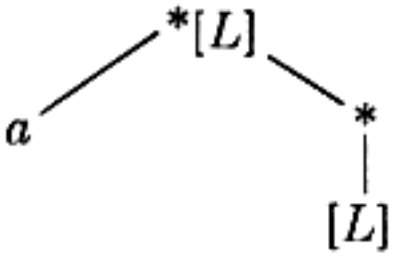
or



17. root(T) = A; root(T[2]) = C; root(T[2, 2]) = E.

18. L[5, 1, 1] = "(2)". L[3, 1] is nonsense, since L[3] is an empty List.

19.



L[2] = "(L)"; L[2, 1, 1] = "a".

20. (Intuitively, the correspondence between b-trees and binary trees is obtained by removing all terminal nodes of the b-tree; see the constructions in Sections 2.3.4.4 and 2.3.4.5.) Let a b-tree with one node correspond to the empty binary tree; and let a b-tree with more than one node, consisting therefore of a root r and b-trees T_1 and T_2, correspond to the binary tree with root r, left subtree T_1', and right subtree T_2', where T_1 and T_2 correspond respectively to T_1' and T_2'.

21. $1 + 0 \cdot n_1 + 1 \cdot n_2 + \cdots + (m - 1) \cdot n_m$. For the number of nodes in the tree is $n_0 + n_1 + n_2 + \cdots + n_m$, and this also equals 1 + (number of sons in the tree) $= 1 + 0 \cdot n_0 + 1 \cdot n_1 + 2 \cdot n_2 + \cdots + m \cdot n_m$.

SECTION 2.3.1

1. INFO(T) = A, INFO(RLINK(T)) = C, etc.; the answer is H.

2. Preorder: 1245367; symmetric order: 4251637; endorder: 4526731.

3. The statement is true (notice for example that nodes 4, 5, 6, 7 always appear in this order in Ex. 2). The result is immediately proved by induction on the size of the binary tree.

4. It is the reverse of endorder. (This is easily proved by induction.)

5. For example in the tree of Ex. 2, preorder is (using binary notation which is in this case equivalent to the Dewey system) 1, 10, 100, 101, 11, 110, 111. This is recognizable as sorting from left to right, as in a dictionary.

In general, the nodes will be listed in preorder if they are sorted lexicographically from left to right, with "blanks" treated as less than 0 or 1. The nodes will be listed in postorder if they are sorted lexicographically with 0 < "blank" < 1. For endorder, use 0 < 1 < "blank".

6. The fact that $p_1\ p_2 \ldots p_n$ is obtainable with a stack is readily proved by induction on n, or in fact we may observe that Algorithm T does precisely what is required in its stack actions. (The corresponding sequence of S's and X's as in Ex. 2.2.1–3 is the same as the sequence of 1's and 2's as subscripts in the dual order, see Ex. 18.)

Conversely, if $p_1\, p_2 \ldots p_n$ is obtainable with a stack and if $p_k = 1$, then $p_1 \ldots p_{k-1}$ is a permutation of $\{2, \ldots, k\}$ and $p_{k+1} \ldots p_n$ is a permutation of $\{k+1, \ldots, n\}$ each of which are obtainable by stack, and which are the permutations corresponding to the left and right subtrees. The proof now proceeds by induction.

7. From the preorder, the root is known; then from the postorder, we know the left subtree and the right subtree; and in fact we know the preorder and postorder of the nodes in the latter subtrees. Hence the tree is readily constructed (and indeed it is quite amusing to construct a simple algorithm which links the tree together in the normal fashion, starting with the nodes linked together in preorder in LLINK and in postorder in RLINK). Similarly, postorder and endorder together characterize the structure. But preorder and endorder do not; there are two binary trees having "AB" as preorder and "BA" as endorder. If all nonterminal nodes of a binary tree have *both* branches nonempty, its structure *is* characterized by preorder and endorder.

8. (a) Trees with all LLINKs null. (b) Trees with zero or one nodes. (c) Trees with all RLINKs null.

9. T1 once, T2 $(2n+1)$ times, T3 n times, T4 $(n+1)$ times, T5 n times. This is derived by induction or by Kirchhoff's law.

10. A binary tree with all RLINKs null will cause all n node addresses to be put in the stack before any are removed.

11. Let a_{nk} be the number of binary trees with n nodes for which the stack in Algorithm T never contains more than k items. If $g_k(z) = \sum_n a_{nk} z^n$, we find $g_1(z) = 1/(1-z)$, $g_2(z) = 1/\big(1 - z/(1-z)\big) = (1-z)/(1-2z)$, . . . , $g_k(z) = 1/\big(1 - zg_{k-1}(z)\big) = q_{k-1}(z)/q_k(z)$ where $q_{-1}(z) = q_0(z) = 1$, $q_{k+1}(z) = q_k(z) - zq_{k-1}(z)$. $g_\infty(z)$ is, of course, the generating function for the number of binary trees, see Section 2.3.4. These formulas make it possible to calculate the value a_{nk} readily, but seem to give no obvious way to find the asymptotic value of $na_{nn} - \sum_{0 \le k < n} a_{nk}$ which is required for this problem. Presumably the mean is asymptotic to $\alpha\sqrt{n}$ for some α. (See the references in the answer to Ex. 2.3.4.5–5, which deal with the case of oriented trees.) Empirical calculations show good agreement with the formula $\sqrt{\pi\, n} - 1.6$.

12. Visit NODE(P) between step T2 and T3, instead of between step T4 and T2. For the proof, show that "Starting at step T2 with . . . original value A[1] . . . A[m]." essentially as in the text.

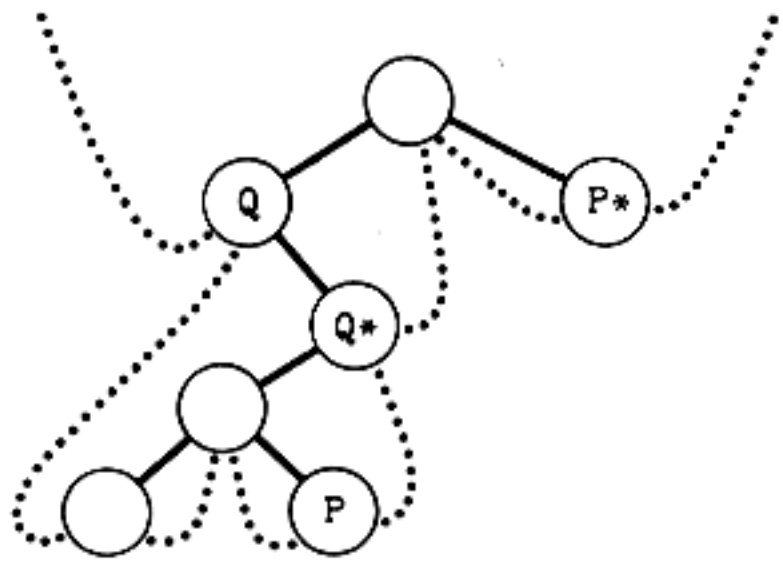
13. Let steps T1, T2 be unchanged. In step T3, put (P, 0) on top of the stack instead of just P. In step T4, when the stack is not empty, set (P, d) $\Leftarrow$ A; and if $d = 0$ set A $\Leftarrow$ (P, 1), P $\leftarrow$ RLINK(P), and return to T2. Finally, step T5 becomes "Visit NODE(P), and go to T4."

14. By induction, there are always exactly $n+1$ A links (counting T when it is null). There are n non-null links, counting T, so the remark in the text about the majority of null links is justified.

15. There is a thread LLINK pointing to a node if and only if it has a nonempty right subtree; there is a thread RLINK pointing to a node if and only if its left subtree is nonempty. (See Fig. 24.)

16. If LTAG(Q) = "+", Q* = LLINK(Q), i.e. one step down to the left. Otherwise Q* is obtained by going upwards in the tree (if necessary) repeatedly until the first time it is possible to go down to the right without retracing steps; typical examples

are the trips from P to P* and from Q to Q* in the following tree:



17. If LTAG(P) = "+", set Q ← LLINK(P) and terminate; otherwise set Q ← P and now set Q ← RLINK(Q) zero or more times until finding RTAG(Q) = "+", and finally set Q ← RLINK(Q) one further time.

18. Modify Algorithm T by inserting a step T2a, "Visit NODE(P) the first time"; and in step T5, we are visiting NODE(P) the second time.

Given a threaded tree the traversal is extremely simple:

$$(P, 1)^{\Delta} = (\texttt{LLINK(P)}, 1) \text{ if } \texttt{LTAG(P)} = \text{"+"}, \text{ otherwise } (P, 2);$$
$$(P, 2)^{\Delta} = (\texttt{RLINK(P)}, 1) \text{ if } \texttt{RTAG(P)} = \text{"+"}, \text{ otherwise } (\texttt{RLINK(P)}, 2).$$

In each case, we move at most one step in the tree; so in practice the dual order and the values of d and e are embedded in a program and not explicitly mentioned.

Suppressing all the "first visits" gives us precisely algorithms T and S; suppressing all the "second visits" gives us the solutions to Exs. 12 and 17.

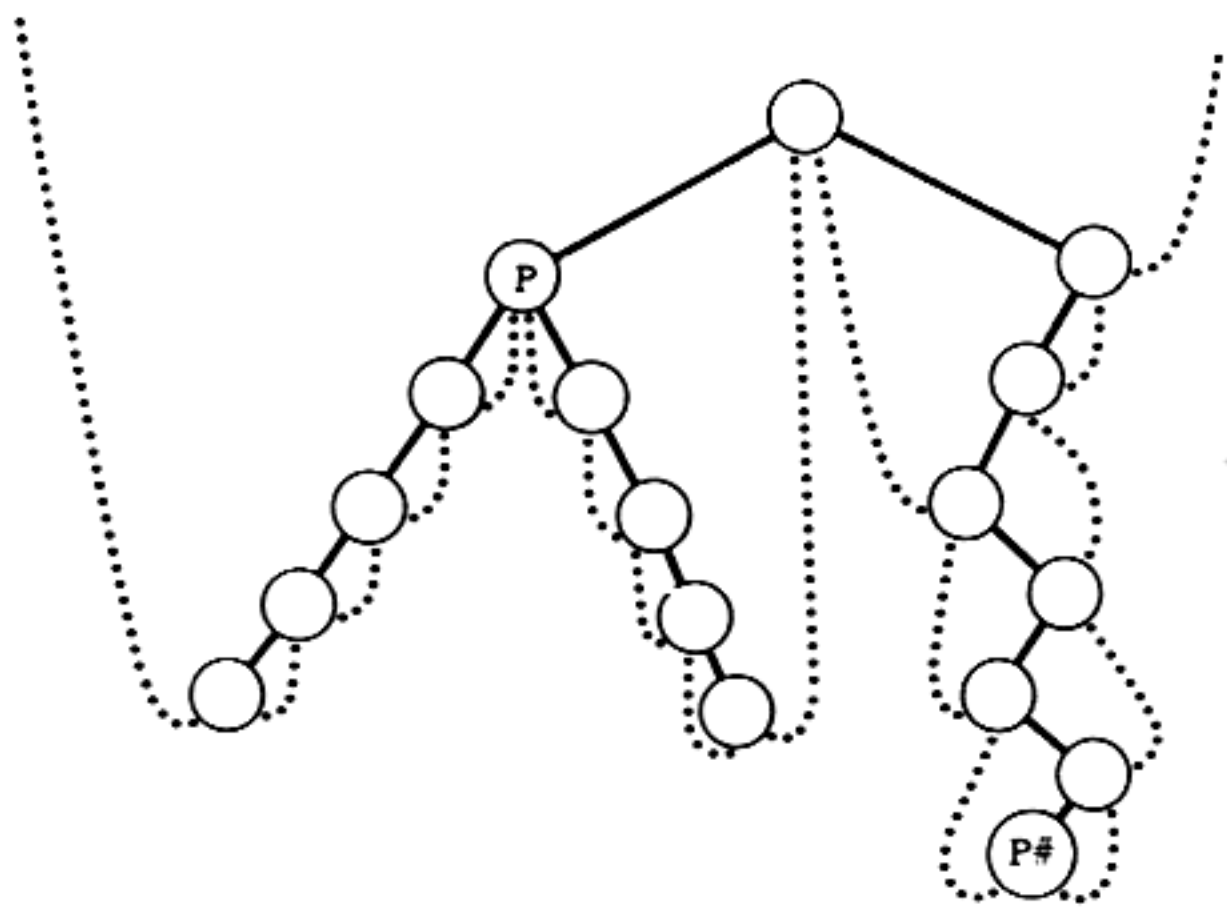
19. E1. Set Q ← P. If Q = HEAD, go to E5.

E2. If RTAG(Q) = "+", set Q ← RLINK(Q) and repeat this step.

E3. Set Q ← RLINK(Q). If now LLINK(Q) = P, go to E5; otherwise set Q ← LLINK(Q).

E4. If RLINK(Q) ≠ P, set Q ← RLINK(Q) and repeat this step; otherwise terminate the algorithm.

E5. If RTAG(Q) = "+", set Q ← Q$ using Algorithm S and repeat this step. ∎



Note: There seems to be no more efficient algorithm than this (consider for example P and P# in the tree shown at the bottom of the previous page), and it is inferior to an algorithm using a stack (like Ex. 13) for traversing an entire tree in endorder.

20. Replace lines 06–09 by:

```
T3  ENT4  0,6
    LD6   AVAIL
    J6Z   OVERFLOW
    LDX   0,6(LINK)
    STX   AVAIL
    ST5   0,6(INFO)
    ST4   0,6(LINK)
```

Replace lines 12–13 by:

```
    LD4   0,6(LINK)
    LD5   0,6(INFO)
    LDX   AVAIL
    STX   0,6(LINK)
    ST6   AVAIL
    ENT6  0,4
```

If two more lines of code are added at line 06

```
T3  LD4   0,5(LLINK)
    J4Z   T5
```

the running time goes down from $30n + a + 3$ to $27a + 7n - 23$ units. (This same device would reduce the running time of program T to $12a + 7n - 7$, which is a slight improvement, if we set $a = (n + 1)/2$.)

21. The following algorithm may in fact be used for traversal in any of the three orders:

V1. [Initialize.] Set P ← LOC(T), Q ← T. If Q = Λ, terminate the algorithm.

V2. [Preorder visit.] If traversing in preorder, visit NODE(Q).

V3. [Go to left.] Set R ← LLINK(Q). If R ≠ Λ, set LLINK(Q) ← P, P ← Q, Q ← R, and go back to V2. (It is assumed that RTAG(P) is initially "+".)

V4. [Postorder visit.] If traversing in postorder, visit NODE(Q).

V5. [Go to right.] Set R ← RLINK(Q). If R ≠ Λ, set RTAG(Q) ← "−", RLINK(Q) ← P, P ← Q, Q ← R, go to V2.

V6. [Endorder visit.] If traversing in endorder, visit NODE(Q).

V7. [Go up.] If P = LOC(T), terminate the algorithm. Otherwise if RTAG(P) = "+", set R ← LLINK(P), LLINK(P) ← Q, Q ← P, P ← R, and go to V4. Otherwise set R ← RLINK(P), RTAG(P) ← "+", RLINK(P) ← Q, Q ← P, P ← R, and go to V6. ∎

Algorithms related to this one are discussed further in Section 2.3.5. It is apparently impossible to solve the stated problem without using the additional one-bit field RTAG in each node, but this has not been proved. The above algorithm works even if the tree involves "shared" subtrees, and it has the possible further advantage that NODE(P) is the father of the node being visited. But the fact that it radically affects the structure while it is doing the processing makes it unsuitable for use except in special situations, such as in connection with the copying routine, Algorithm C.

22. Let rI4 ≡ R, rI5 ≡ Q, rI6 ≡ P, and use other conventions of Programs T and S.

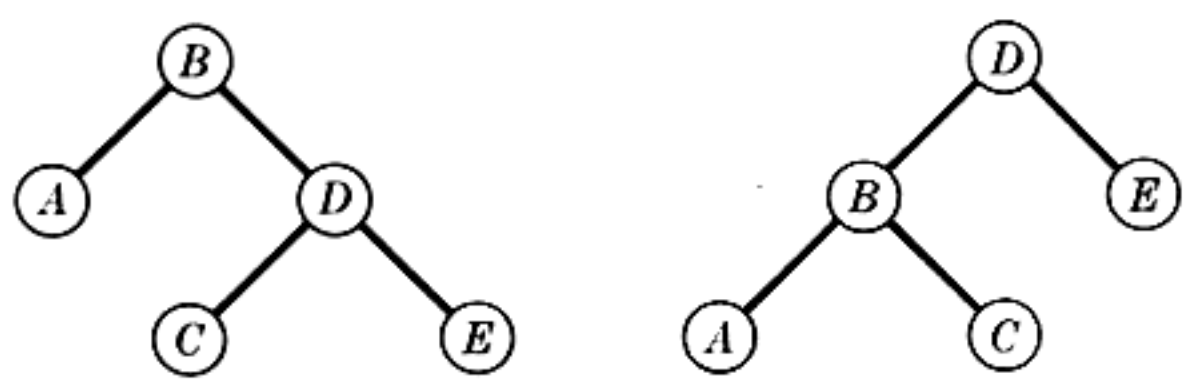
```
V1  ENT6  T              1    VI. Initialize. P ← LOC(T).
    LD5   T(LLINK)       1    Q ← T.
```

```
     J5NZ  V3              1
     JMP   DONE            0      Special exit for empty tree
1H   ST6   0,5(LLINK)      a - 1  LLINK(Q) ← P.
2H   ENT6  0,5             n - 1  P ← Q.
     ENT5  0,4             n - 1  Q ← R.
V3   LD4   0,5(LLINK)        n    V3. Go to left. R ← LLINK(Q).
     J4NZ  1B                n    Repeat if R ≠ Λ.
V4   JMP   VISIT             n    V4. Post-order visit.
V5   LD4   1,5(RLINK)        n    V5. Go to right. R ← RLINK(Q).
     J4Z   V7                n    To V7 if R = Λ.
     ENNA  0,6             n - a
     STA   1,5(RLINKT)     n - a  RLINK(Q) ← P, RTAG(Q) ← "-".
     JMP   2B              n - a  P ← Q, Q ← R, go to V3.
V7   ENT4  -T,6              n    V7. Go up.
     J4Z   DONE              n    Terminate if P = LOC(T).
     LD4   1,6(RLINKT)     n - 1  R ← RLINKT(P).
     J4N   1F              n - 1  Jump if RTAG(P) = "-".
     LD4   0,6(LLINK)      a - 1  R ← LLINK(P).
     ST5   0,6(LLINK)      a - 1  LLINK(P) ← Q.
     ENT5  0,6             a - 1  Q ← P.
     ENT6  0,4             a - 1  P ← R.
     JMP   V4              a - 1
1H   ST5   1,6(RLINKT)     n - a  RLINK(P) ← Q, RTAG(P) ← "+".
     ENT5  0,6             n - a  Q ← P.
     ENN6  0,4             n - a  P ← -R.
     JMP   V7              n - a                                  ∎
```

The running time is $23n - 10$ (curiously independent of a), for $n \neq 0$. So the execution time is competitive with Ex. 20.

23. Insertion to the right: RLINKT(Q) ← RLINKT(P), RLINKT(P) ← +Q, LLINK(Q) ← Λ. Insertion to the left, assuming LLINK(P) = Λ: Set LLINK(P) ← Q, LLINK(Q) ← Λ, RLINKT(Q) ← -P. Insertion to the left, between P and LLINK(P) ≠ Λ: Set R ← LLINK(P), LLINK(Q) ← R, and then if RTAG(R) ≠ "—" set R ← RLINK(R) repeatedly until RTAG(R) = "—"; finally, set RLINK(R) ← Q, LLINK(P) ← Q, RLINKT(Q) ← -P. (A more efficient algorithm for the last case can be used if we know a node F such that P = LLINK(F) or P = RLINK(F); assuming the latter, for example, we could set INFO(P) ↔ INFO(Q), RLINK(F) ← Q, LLINK(Q) ← P, RLINKT(P) ← -Q; this takes a fixed amount of time, but since it switches nodes around in memory it is generally not recommended.)

24. No:



25. We first prove (b), by induction on the number of nodes in T, and similarly (c). Now (a) breaks into special cases; write $T \preceq_1 T'$ if (1) holds, $T \preceq_2 T'$ if (2) holds, etc. Then $T \preceq_1 T'$ and $T' \preceq T''$ implies $T \preceq_1 T''$; $T \preceq_2 T'$ and $T' \preceq T''$ implies $T \preceq_2 T''$; and the remaining two cases are treated by proving (a) by induction on the number of nodes in T.

26. If the dual order of T is $(u_1, d_1), (u_2, d_2), \ldots, (u_{2n}, d_{2n})$ where the u's are nodes and the d's are 1 or 2, form the "trace" of the tree $(v_1, s_1), (v_2, s_2), \ldots, (v_{2n}, s_{2n})$, where $v_j = \text{info}\,(u_j)$, and $s_j = l(u_j)$ or $r(u_j)$ according as $d_j = 1$ or 2. Now $T \preceq T'$ if and only if the trace of T (as defined here) *lexicographically* precedes or equals the trace of T'. Formally, this means either $n \leq n'$ and $(v_j, s_j) = (v'_j, s'_j)$ for $1 \leq j \leq n$, or else there is a k for which $(v_j, s_j) = (v'_j, s'_j)$ for $1 \leq j < k$ and either $v_k \prec v'_k$ or $v_k = v'_k$ and $s_k < s'_k$.

27. **R1.** [Initialize.] Set P ← HEAD, P′ ← HEAD′, i.e. the respective list heads of the given right-threaded binary trees. Go to R3.

R2. [Check INFO.] If INFO(P) $\prec$ INFO(P′), terminate ($T \prec T'$); if INFO(P) $\succ$ INFO(P′), terminate ($T \succ T'$).

R3. [Go to left.] If LLINK(P) = Λ = LLINK(P′), go to R4; if LLINK(P) = $\Lambda \neq$ LLINK(P′), terminate ($T \prec T'$); if LLINK(P) $\neq \Lambda$ = LLINK(P′), terminate ($T \succ T'$); otherwise set P ← LLINK(P), P′ ← LLINK(P′) and go to R2.

R4. [End of tree?] If P = HEAD (or, equivalently, if P′ = HEAD′), terminate (T equivalent to T').

R5. [Go to right.] If RTAG(P) = "—" = RTAG(P′), set P ← RLINK(P), P′ ← RLINK(P′), and go to R4. If RTAG(P) = "—" $\neq$ RTAG(P′), terminate ($T \prec T'$). If RTAG(P) $\neq$ "—" = RTAG(P′), terminate ($T \succ T'$). Otherwise, set P ← RLINK(P), P′ ← RLINK(P′), and go to R2. ∎

To prove the validity of this algorithm (and therefore to understand how it works), one may show by induction on the size of the tree T_0 that the following statement is valid: "Starting at step R2 with P and P′ pointing to the roots of two nonempty right-threaded binary trees T_0 and T'_0, the algorithm will terminate if T_0 and T'_0 are not equivalent, indicating whether $T_0 \prec T'_0$ or $T_0 \succ T'_0$; the algorithm will reach step R4 if T_0 and T'_0 *are* equivalent, with P and P′ then pointing respectively to the successor nodes of T_0 and T'_0 in postorder."

28. Equivalent *and* similar.

29. Prove by induction on the size of T that the following statement is valid: "Starting at step C2 with P pointing to the root of a nonempty binary tree T and with Q pointing to a node that has empty left and right subtrees, the procedure will ultimately arrive at step C6 after attaching copies of the left and right subtrees of NODE(P) to NODE(Q), and with P and Q pointing respectively to the successor nodes of T and NODE(Q) in preorder."

30. Assume the pointer T in (2) is LLINK(HEAD) in (9).

L1. [Initialize.] Set Q ← HEAD, RLINK(Q) ← Q.

L2. [Advance.] Set P ← Q$. (See below.)

L3. [Thread.] If RLINK(Q) = Λ, set RLINK(Q) ← P, RTAG(Q) ← "−", else set RTAG(Q) ← "+". If LLINK(P) = Λ, set LLINK(P) ← Q, LTAG(P) ← "−", else set LTAG(P) ← "+".

L4. [Done?] If P ≠ HEAD, set Q ← P and return to L2. ▮

Step L2 of this algorithm implies the activation of a postorder traversal coroutine like Algorithm T, with the additional proviso that Algorithm T "visits" HEAD after it has fully traversed the tree. This notation is a convenient simplification in the description of tree algorithms, since we need not repeat the stack mechanisms of Algorithm T over and over again. Of course Algorithm S cannot be used during step L2, since the tree hasn't been threaded yet. But the algorithm of Ex. 21 *can* be used in step L2, and this provides us with a very pretty method that threads a tree *without using any auxiliary stack!*

31. (a) Set P ← HEAD; (b) set Q ← P$ (e.g. using Algorithm S, modified for a right-threaded tree); (c) if P ≠ HEAD, AVAIL ⇐ P; (d) if Q ≠ HEAD, set P ← Q and return to (b). [Other solutions which decrease the length of the inner loop are clearly possible, although the order of the basic steps is somewhat critical. The above procedure works since we never return a node to available storage until after Algorithm S has looked at both its LLINK and its RLINK; as observed in the text, each of these links is used precisely once during a complete tree traversal.]

32. RLINK(Q) ← RLINK(P), SUC(Q) ← SUC(P), SUC(P) ← RLINK(P) ← Q, PRED(Q) ← P, PRED(SUC(Q)) ← Q.

33. Inserting NODE(Q) just to the left and below NODE(P) is quite simple: Set LLINKT(Q) ← LLINKT(P), LLINKT(P) ← +Q, RLINK(Q) ← Λ. Insertion to the right is considerably harder, since it essentially requires finding *Q, which is of comparable difficulty to finding Q# (see exercise 19); the node-moving technique discussed in exercise 23 could perhaps be used. So general insertions are more difficult with this type of threading. But the insertions required by Algorithm C are not as difficult as insertions are in general, and in fact the copying process is slightly faster for this kind of threading:

C1. Set P ← HEAD, Q ← U, go to C4. (The assumptions and philosophy of Algorithm C in the text are being used throughout.)

C2. If RLINK(P) ≠ Λ, set R ⇐ AVAIL, LLINK(R) ← LLINK(Q), LTAG(R) ← "−", RLINK(R) ← Λ, RLINK(Q) ← LLINK(Q) ← R.

C3. Set INFO(Q) ← INFO(P).

C4. If LTAG(P) = "+", set R ⇐ AVAIL, LLINK(R) ← LLINK(Q), LTAG(R) ← "−", RLINK(R) ← Λ, LLINK(Q) ← R, LTAG(Q) ← "+".

C5. Set P ← LLINK(P), Q ← LLINK(Q).

C6. If P ≠ HEAD, go to C2. ▮

The algorithm now seems almost too simple to be correct!

Algorithm C for threaded or right-threaded binary trees is slightly longer due to the extra time to calculate P*, Q* in step C5.

It would be possible to thread RLINKs in the usual way or to put #P in RLINK(P), in conjunction with the above copying method, by appropriately setting RLINK(R) and RLINKT(Q) in steps C2 and C4.

34. **A1.** Set Q ← P, and then repeatedly set Q ← RLINK(Q) zero or more times until RTAG(Q) = "—".

A2. Set R ← RLINK(Q). If LLINK(R) = P, set LLINK(R) ← Λ; otherwise set R ← LLINK(R), and repeatedly set R ← RLINK(R) zero or more times until RLINK(R) = P, then finally set RLINKT(R) ← RLINKT(Q). (This step has removed NODE(P) and its subtrees from the original tree.)

A3. Set RLINK(Q) ← HEAD, LLINK(HEAD) ← P. ▮

(The key to inventing and/or understanding this algorithm is the construction of good "before and after" diagrams.)

36. No; cf. the answer to Ex. 1.2.1–15(e).

37. If

$$\texttt{LLINK(P)} = \texttt{RLINK(P)} = \Lambda$$

in the representation (2), let

$$\texttt{LINK(P)} = \Lambda;$$

otherwise let LINK(P) = Q where NODE(Q) corresponds to NODE(LLINK(P)) and NODE(Q+1) to NODE(RLINK(P)). The condition LLINK(P) or RLINK(P) = Λ is represented by a sentinel in NODE(Q) or NODE(Q+1) respectively. This representation uses between n and $2n - 1$ memory positions; under the stated assumptions, (2) would require 18 words of memory, compared to 11 in the present scheme. Insertion and deletion operations are approximately of equal efficiency in either representation.

SECTION 2.3.2

1. If B is empty, $F(B)$ is an empty forest. Otherwise, $F(B)$ consist of a tree T plus the forest $F(\text{rightsubtree}(B))$, where $\text{root}(T) = \text{root}(B)$ and $\text{subtrees}(T) = F(\text{leftsubtree}(B))$.

2. The number of zeroes in the binary notation is the number of decimal points in the decimal notation, and the exact formula for the correspondence is

$$a_1 \,.\, a_2 \,.\, \cdots \,.\, a_k \leftrightarrow 1^{a_1}01^{a_2-1}0 \cdots 01^{a_k-1},$$

where 1^a denotes a ones in a row.

3. Sort the Dewey decimal notations for the nodes lexicographically (from left to right, as in a dictionary), placing a shorter sequence $a_1 \,.\, \cdots \,.\, a_k$ in front of its extensions $a_1 \,.\, \cdots \,.\, a_k \,.\, \cdots \,.\, a_r$ for preorder, and behind its extensions for postorder. (Thus, if we were sorting words instead of sequences of numbers, we would place the words *cat*, *cataract* in usual dictionary order, for preorder, but with the order of initial subwords reversed "*cataract*, *cat*", for postorder.) These rules are readily proved by induction on the size of the tree.

4. It is false. The terminal nodes do occur in the same relative position in preorder and postorder, as induction readily shows; but for example in a tree with exactly two levels we find that endorder is precisely the opposite. The corresponding statement *is* true for binary trees (Ex. 2.3.1–3).

5. First list the roots of all the trees; then list the second-level nodes of the rightmost tree; then the third-level nodes of the rightmost second-level node listed; and so on, always giving the nodes of a complete family together, using the farthest right, highest level family that has not already been listed, whose parent has been listed.

6. We have preorder(T) = preorder(T'), and endorder(T) = postorder(T'), even if T has nodes with only one son; the remaining two orders are not in any simple relation (for example, the root of T comes at the end in one case and about in the middle in the other).

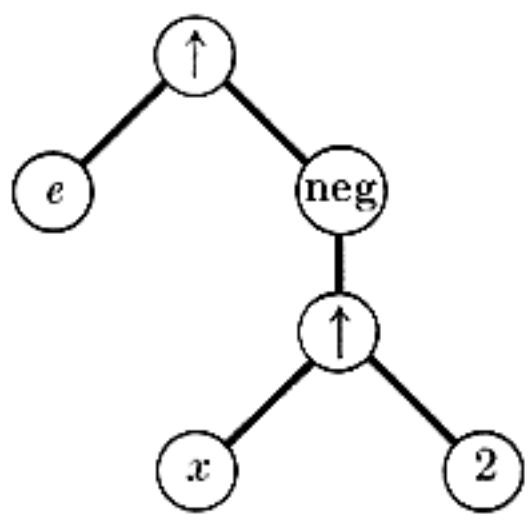
7. (a) yes; (b) no; (c) no; (d) no; (e) yes; (f) yes.

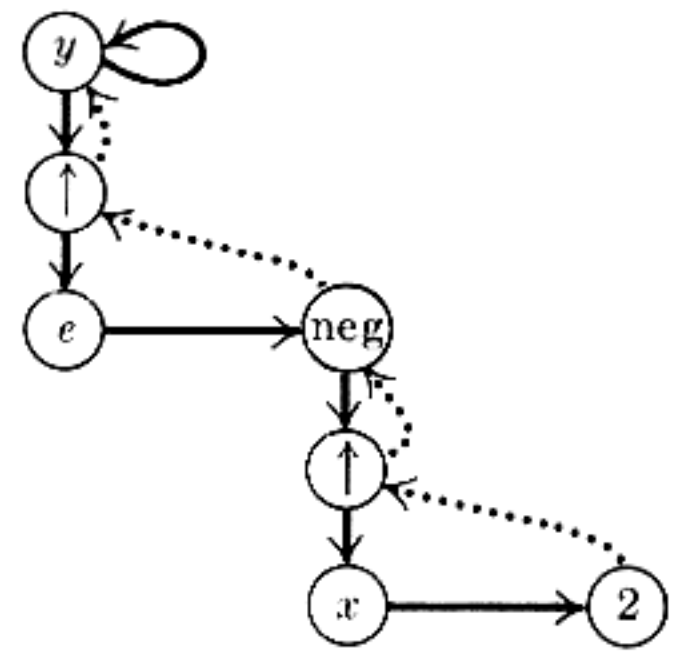
8. $T \preceq T'$ means $\text{info}(\text{root}(T)) \prec \text{info}(\text{root}(T'))$, or these info's are equal and either (a): the subtrees of root(T) are $T_1, \ldots, T_n$ and the subtrees of root(T') are $T'_1, \ldots, T'_{n'}$, where there is a k such that T_j is equivalent to T'_j for $1 \le j < k$ but T_k is $\preceq$ and not equivalent to T'_k; or (b): with the notation of (a), T_j is equivalent to T'_j for $1 \le j \le n$, and $n \le n'$.

9. The number of nonterminal nodes is one less than the number of right links that are Λ, in a nonempty forest, because the null right links correspond to the rightmost son of each nonterminal node, and also to the root of the rightmost tree in the forest. (This fact gives another proof of Ex. 2.3.1–14, since obviously the number of null left links is equal to the number of *terminal* nodes.)

10. The forests are similar if and only if $n = n'$, and $s(u_j) = s(u'_j)$, $1 \le j \le n$; they are equivalent if and only if in addition $\text{info}(u_j) = \text{info}(u'_j)$, $1 \le j \le n$. The proof is similar to the previous proof, by generalizing Lemma 2.3.1P (take $f(u) = s(u) - 1$).

11.





12. If INFO(Q1) $\neq 0$, do the following operations: set R $\leftarrow$ COPY(P1); then if TYPE(P2) = 0 and INFO(P2) $\neq 2$, set R $\leftarrow$ TREE("↑",R,INFO(P0) − 1); if TYPE(P2) $\neq 0$, set R $\leftarrow$ TREE("↑",R,TREE("−",COPY(P2),"1")); then set Q1 $\leftarrow$ MULT(Q1,MULT(COPY(P2),R)).

If INFO(Q) $\neq 0$, set Q $\leftarrow$ TREE("×",MULT(TREE("ln",COPY(P1)),Q),TREE("↑", COPY(P1),COPY(P2)). Finally go to DIFF[4].

13. The following program implements Algorithm C with rI1 $\equiv$ P, rI2 $\equiv$ Q, rI3 $\equiv$ R, and with "list head" changed to "root of tree":

```
62    ST3   6F(0:2)    Save contents of rI3, rI2.
63    ST2   7F(0:2)    C1. Initialize.
64    ENT2  8F         Start by creating NODE(U) with
```

```
65          JMP   1F          RLINK(U) = Λ.
66    2H    ENN2  0,2         Q ← −Q.
67    C2    LDA   0,1         C2. Anything to right?
68          JAN   C3          Jump if RTAG(P) = "−".
69    1H    LD3   AVAIL       R ⇐ AVAIL.
70          J3Z   OVERFLOW
71          LDA   0,3(LLINK)
72          STA   AVAIL
73          LDA   0,2
74          STA   0,3         Set LLINKRLINKT(R) ← LLINKRLINKT(Q).
75          ST3   0,2(RLINKT) RLINKT(Q) ← +R.
76    C3    LDA   1,1         C3. Copy INFO.
77          STA   1,2         INFO field copied.
78          LDA   0,1(TYPE)
79          STA   0,2(TYPE)   TYPE field copied.
80    C4    STZ   0,2(LLINK)  C4. Anything to left?
81          LDA   0,1(LLINK)
82          JAZ   C5          Jump if LLINK(P) = Λ.
83          LD3   AVAIL       R ⇐ AVAIL.
84          J3Z   OVERFLOW
85          ST3   0,2(LLINK)  LLINK(Q) ← R.
86          LDA   0,3(LLINK)
87          STA   AVAIL
88          ENNA  0,2
89          STA   0,3(RLINKT) RLINKT(R) ← −Q.
90          ENT2  0,3         C5. Advance. Set Q ← R.
91          LD1   0,1(LLINK)  Set P ← LLINK(P) = P*.
92          JMP   C2
93    8H    CON   0           Zero constant for initialization
94    C5    LD2N  0,2(RLINKT) Q ← −RLINKT(Q).
95          LD1   0,1(RLINK)  P ← RLINK(P).
96          J2N   2B          Jump if RTAG(Q) = "+".
97    C6    J2NZ  C5          C6. Test if complete.
98          LD1   8B(RLINK)   rI1 ← location of first node created.
99          STZ   8B          Reset constant to zero for next COPY.
100   6H    ENT3  *           Restore index registers.
101   7H    ENT2  *                                    ∎
```

14. Let a be the number of nonterminal (operator) nodes copied. The number of executions of the various lines in the previous program is as follows: 62–65, 1; 66, $n - 1 - a$; 67–68, $n - 1$; 69–75, $n - a$; 76–83, n; 84–92, a; 94–96, n; 97, $a + 1$; 98–101, 1. The total time comes to $(35n + a + 9)u$; about $\frac{1}{4}$ of this is to get available nodes, $\frac{1}{4}$ to traverse, and $\frac{1}{2}$ to copy the INFO and LINK information.

15. Comments are left to the reader.

```
216   DIV   LDA   1,6
217         JAZ   1F
218         JMP   COPYP2
219         ENTA  SLASH
220         ENTX  0,6
```

```
221             JMP   TREE2
222             ENT6  0,1
223     1H      LDA   1,5
224             JAZ   SUB
225             JMP   COPYP2
226             ST1   1F(0:2)
227             ENTA  CON2
228             JMP   TREE0
229             ENTA  UPARROW
230     1H      ENTX  *
231             JMP   TREE2
232             ST1   1F(0:2)
233             JMP   COPYP1
234             ENTA  0,1
235             ENT1  0,5
236             JMP   MULT
237             ENTX  0,1
238     1H      ENT1  *
239             ENTA  SLASH
240             JMP   TREE2
241             ENT5  0,1
242             JMP   SUB
```

16. Comments are left to the reader.

```
243     PWR     LDA   1,6
244             JAZ   4F
245             JMP   COPYP1
246             ST1   R(0:2)
247             LDA   0,3(TYPE)
248             JANZ  2F
249             LDA   1,3
250             DECA  2
251             JAZ   3F
252             INCA  1
253             STA   CON0+1
254             JMP   TREE0
255             STZ   CON0+1
256             JMP   5F
257     2H      JMP   COPYP2
258             ST1   1F(0:2)
259             ENTA  CON1
260             JMP   TREE0
261     1H      ENTX  *
262             ENTA  MINUS
263             JMP   TREE2
264     5H      LDX   R(0:2)
265             ENTA  UPARROW
266             JMP   TREE2
267             ST1   R(0:2)
```

```
268     3H    JMP    COPYP2
269           ENTA   0,1
270     R     ENT1   *
271           JMP    MULT
272           ENTA   0,6
273           JMP    MULT
274           ENT6   0,1
275     4H    LDA    1,5
276           JAZ    ADD
277           JMP    COPYP1
278           ENTA   LOG
279           JMP    TREE1
280           ENTA   0,1
281           ENT1   0,5
282           JMP    MULT
283           ST1    1F(0:2)
284           JMP    COPYP1
285           ST1    2F(0:2)
286           JMP    COPYP2
287     2H    ENTX   *
288           ENTA   UPARROW
289           JMP    TREE2
290     1H    ENTX   *
291           ENTA   TIMES
292           JMP    TREE2
293           ENT5   0,1
294           JMP    ADD
```

20. More generally, let u and v be any nodes of a forest. If u is an ancestor of v, it is immediate by induction that u precedes v in preorder and follows v in postorder. Conversely, suppose u precedes v in preorder and follows v in postorder; we must show that u is an ancestor of v. This is clear if u is the root of the first tree. If u is another node of the first tree, v must be also, since u follows v in postorder; so induction applies. If u is not in the first tree, v must not be either, since u precedes v in preorder; so again induction applies.

21. If `NODE(P)` is a binary operator, pointers to its two operands are `P1 = LLINK(P)` and `P2 = RLINK(P1) = $P`. Algorithm D makes use of the fact that `P2$ = P`, so that `RLINK(P1)` may be changed to `Q1`, a pointer to the derivative of `NODE(P1)`, then later `RLINK(P1)` is reset in step D3. For ternary operations, it is difficult to generalize this trick; we would have, say, `P1 = LLINK(P)`, `P2 = RLINK(P1)`, `P3 = RLINK(P2) = $P`. Now after computing the derivative `Q1`, we could set `RLINK(P1)` ← `Q1` temporarily, and then after computing the next derivative `Q2` we could set `RLINK(Q2)` ← `Q1` and `RLINK(P2)` ← `Q2` and reset `RLINK(P1)` ← `P2`. But this is certainly inelegant, and it becomes progressively more so as the degree of the operator becomes higher. Therefore the device of temporarily changing `RLINK(P1)` in Algorithm D is definitely a *trick*, not a *technique;* a more aesthetic way to control a differentiation process, because it generalizes to operators of higher degree and does not rely on isolated tricks, may be based on Algorithm 2.3.3F, and this is discussed in detail in Ex. 2.3.3–3.

22. From the definition it follows immediately that the relation is transitive, i.e. if $T \subseteq T'$ and $T' \subseteq T''$ then $T \subseteq T''$. (In fact the relation is easily seen to be a partial ordering.) Clearly if we let f be the function taking nodes into themselves, $l(T) \subseteq T$ and $r(T) \subseteq T$. Therefore if $T \subseteq l(T')$ or $T \subseteq r(T')$ we must have $T \subseteq T'$.

Suppose f_l and f_r are functions that respectively show $l(T) \subseteq l(T')$ and $r(T) \subseteq r(T')$. Let $f(u) = f_l(u)$ if u is in $l(T)$, $f(u) = \text{root}(T')$ if u is $\text{root}(T)$, otherwise $f(u) = f_r(u)$. Now it follows easily that f shows that $T \subseteq T'$; for example, if we let $r'(T)$ denote $r(T) - \text{root}(T)$ we have $\text{preorder}(T) = \text{root}(T)\ \text{preorder}\big(l(T)\big)\ \text{preorder}\big(r'(T)\big)$; $\text{preorder}(T') = f\big(\text{root}(T)\big)\ \text{preorder}\big(l(T')\big)\ \text{preorder}\big(r'(T')\big)$.

The converse does not hold, e.g. consider the subtrees with roots b and b' in Fig. 25.

SECTION 2.3.3

1. The family-order representation becomes the preorder representation if LLINK and RLINK are consistently interchanged in the binary tree. Therefore anything that is true for preorder representation remains true for family-order under this correspondence.

2. Make the following changes in Algorithm F: Step F1, change to "last node of the forest in preorder." Step F2, change "$f(x_d), \ldots, f(x_1)$" to "$f(x_1), \ldots, f(x_d)$" in two places. Step F4, "If P is the first node in preorder, terminate the algorithm. (Then the stack contains $f(\text{root}(T_1)), \ldots, f(\text{root}(T_m))$, from top to bottom, where $T_1, \ldots, T_m$ are the trees of the given forest, from left to right.) Otherwise set P ← *P (P ← P − 1 in the given representation), and return to F2."

3. In step D1, also set S ← Λ. (S is a link variable that links to the top of the stack.) Step D2 becomes, for example, "If NODE(P) denotes a unary operator, set Q ← S, S ← RLINK(Q), P1 ← LLINK(P); if it denotes a binary operator, set Q ← S, Q1 ← RLINK(Q), S ← RLINK(Q1), P1 ← LLINK(P), P2 ← RLINK(P1). Then perform DIFF[TYPE(P)]." Step D3 becomes "Set RLINK(Q) ← S, S ← Q." Step D4 becomes "Set P ← PS." The operation LLINK(DY) ← Q may be avoided in step D5 if we assume S ≡ LLINK(DY). This technique clearly generalizes to ternary and higher-order operators.

4. A representation like (10) takes $n - m$ LLINKs and $n + (n - m)$ RLINKs. The difference in total number of links is $n - 2m$ between the two forms of representation. Arrangement (10) is superior when the LLINK and INFO fields require about the same amount of space in a node and when m is rather large, i.e. when the degree of nonterminal nodes is high.

5. It would certainly be silly to include threaded RLINKs, since an RLINK thread just points to FATHER anyway. Threaded LLINKs as in 2.3.2–(4) would be useful if it is necessary to move leftward in the tree, for example if we wanted to traverse a tree in reverse postorder or in family-order; but these operations are not significantly harder without threaded LLINKs unless the nodes tend to have very high degrees.

6. L1. Set P ← FIRST, FIRST ← Λ.

L2. If P = Λ, terminate. Otherwise set Q ← RLINK(P).

L3. If FATHER(P) = Λ, set RLINK(P) ← FIRST, FIRST ← P; otherwise set RLINK(P) ← LSON(FATHER(P)), LSON(FATHER(P)) ← P.

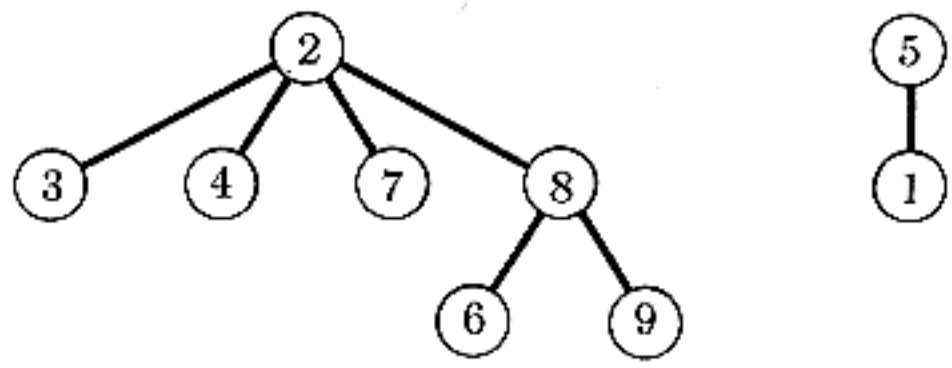
L4. Set P ← Q and return to L2. ∎

7. $\{1, 5\}\ \{2, 3, 4, 7\}\ \{6, 8, 9\}$.

8. Perform step E3 of Algorithm E, then test if $j = k$.

9.

FATHER[k]:	5	0	2	2	0	8	2	2	8
k:	1	2	3	4	5	6	7	8	9



10. (It would be of interest to solve the similar problem for the following "speeded up" version of the equivalence algorithm when n is large, as suggested by Alan Tritter: In step E4, set FATHER[x] $\leftarrow k$ for all values $x \neq k$ which were encountered in step E3. This means an extra pass is made up the trees, but it collapses the trees further so that future searches are faster.)

11. It suffices to define the transformation which is done for each input (P, j, Q, k):

T1. If FATHER(P) $\neq \Lambda$, set $j \leftarrow j +$ DELTA(P), P $\leftarrow$ FATHER(P), and repeat this step.

T2. If FATHER(Q) $\neq \Lambda$, set $k \leftarrow k +$ DELTA(Q), Q $\leftarrow$ FATHER(Q), and repeat this step.

T3. If P = Q, check that $j = k$ (otherwise an error has been made in the input, the equivalences are contradictory). If P $\neq$ Q, set DELTA(Q) $\leftarrow j - k$, FATHER(Q) $\leftarrow$ P, LBD(P) $\leftarrow \min$ (LBD(P), LBD(Q) + DELTA(Q)), UBD(P) $\leftarrow$ max (UBD(P), UBD(Q) + DELTA(Q)).

Note: It is possible to allow the "ARRAY X[l:u]" declarations to occur intermixed with equivalences, or to allow assignment of certain addresses of variables before others are equivalenced to them, etc., under suitable conditions which are not difficult to understand. For further development of this algorithm, see *CACM* **7** (1964), 301–303. Similar algorithms apply to other programming languages, as in COBOL's SAME AREA clauses.

12. (a) Yes. (If this condition is not required, it would be possible to avoid the loops on S which appear in steps A2 and A9.) (b) P returns, but Q does not. To make Q return to the root, one could set Q $\leftarrow$ UP(Q) zero or more times until UP(Q) = Λ.

13. The crucial fact is that the UP chain leading upward from P always mentions the same variables and the same exponents for these variables as the UP chain leading upward from Q, except that the latter chain may include additional steps for variables with exponent zero. (This condition holds throughout most of the algorithm, except during the execution of steps A9 and A10.) Now we get to step A8 either from A3 or from A10, and in each case it was verified that EXP(Q) $\neq 0$. Therefore EXP(P) $\neq 0$ and in particular it follows that P $\neq \Lambda$, Q $\neq \Lambda$, UP(P) $\neq \Lambda$, UP(Q) $\neq \Lambda$, and the result stated in the exercise follows. So the proof depends on showing that the UP chain condition stated above is preserved by the actions of the algorithm.

16. The INFO1, RLINK tables together with the suggestion for computing LTAG in the text gives us the equivalent of a binary tree represented in the usual manner. The

idea is to traverse this tree in postorder, counting degrees as we go:

P1. Let R, D, and I be stacks which are initially empty; then set R $\Leftarrow n+1$, D $\Leftarrow 0$, $j \leftarrow 0$, $k \leftarrow 0$.

P2. If top(R) $> j+1$, (i.e. if LTAG[j] = "+", if that field were present), go to P5.

P3. If I is empty, terminate the algorithm; otherwise set $i \Leftarrow$ I, $k \leftarrow k+1$, INFO2[k] $\leftarrow$ INFO1[i], DEGREE[k] $\Leftarrow$ D.

P4. If RLINK[i] = 0, go to P3; otherwise delete the top of R (which will equal RLINK[i]).

P5. Set top(D) $\leftarrow$ top(D) $+ 1$, $j \leftarrow j+1$, I $\Leftarrow j$, D $\Leftarrow 0$, and if RLINK[j] $\neq 0$ set R $\Leftarrow$ RLINK[j]. Go to P2. ▮

17. We prove (by induction on the number of nodes in a *single tree* T) that if P is a pointer to T, and if the stack is initially empty, steps F2 through F4 will end with the single value $f(\text{root}(T))$ on the stack. This is true for $n = 1$. If $n > 1$, there are $0 < d =$ DEGREE(root(T)) subtrees $T_1, \ldots, T_d$; by induction and the nature of a stack, and since postorder consists of $T_1, \ldots, T_d$ followed by root(T), the algorithm computes $f(T_1), \ldots, f(T_d)$, and then $f(\text{root}(T))$, as desired. The validity of Algorithm F for forests follows.

18. G1. Set the stack empty, and let P point to the root of the tree (the last node in postorder). Evaluate f(NODE(P)).

G2. Push DEGREE(P) copies of f(NODE(P)) down onto the stack.

G3. If P is the first node in postorder, terminate the algorithm. Otherwise set P $\leftarrow$ $P (this would be simply P $\leftarrow$ P $-$ 1 in (9)).

G4. Evaluate f(NODE(P)) using the value at the top of the stack (which is f(NODE(FATHER(P)))). Pop this value off the stack, and return to G2. ▮

Note: An algorithm analogous to this one can be based on preorder instead of postorder as in exercise 2.

SECTION 2.3.4.1

1. (B, A, C, D, B), (B, A, C, D, E, B), (B, D, C, A, B), (B, D, E, B), (B, E, D, B), (B, E, D, C, A, B).

2. Let $(V_0, V_1, \ldots, V_n)$ be a path of smallest possible length from V to V'. If now $V_j = V_k$ for some $j < k$, $(V_0, \ldots, V_j, V_{k+1}, \ldots, V_n)$ is a shorter path.

3. (The fundamental path traverses e_3 and e_4 once, but cycle C_2 traverses them "-1" times, giving a net total of zero.) Traverse the following edges: e_1, e_2, e_6, e_7, e_9, e_{10}, e_{11}, e_{12}, e_{14}.

4. If not, let G'' be the subgraph of G' obtained by deleting each edge e_j for which $E_j = 0$. Then G'' is a finite graph which has no cycles and at least one edge, so by the proof of Theorem A there is at least one vertex, V, which is adjacent to exactly one other vertex, V'. Let e_j be the edge joining V to V'; then Kirchhoff's equation (1) at vertex V is $E_j = 0$, contradicting the definition of G''.

5. $A = 1 + E_8$, $B = 1 + E_8 - E_2$, $C = 1 + E_8$, $D = 1 + E_8 - E_5$, $E = 1 + E_{17} - E_{21}$, $F = 1 + E_{13} + E_{17} - E_{21}$, $G = 1 + E_{13}$, $H = E_{17} - E_{21}$,

$J = E_{17}$, $K = E_{19} + E_{20}$, $L = E_{17} + E_{19} + E_{20} - E_{21}$, $P = E_{17} + E_{20} - E_{21}$, $Q = E_{20}$, $R = E_{17} - E_{21}$, $S = E_{25}$. *Note:* In this case it is also possible to solve for $E_2, E_5, \ldots, E_{25}$ in terms of $A, B, \ldots, S$; hence there are nine independent solutions, explaining why we eliminated six variables in Eq. 1.3.3–(8).

6. Fundamental cycles: $C_0 = e_0 + e_1 + e_4 + e_9$ (fundamental path is $e_1 + e_4 + e_9$); $C_5 = e_5 + e_3 + e_2$; $C_6 = e_6 - e_2 + e_4$; $C_7 = e_7 - e_4 - e_3$; $C_8 = e_8 - e_9 - e_4 - e_3$. Therefore we find $E_1 = 1$, $E_2 = E_5 - E_6$, $E_3 = E_5 - E_7 - E_8$, $E_4 = 1 + E_6 - E_7 - E_8$, $E_9 = 1 - E_8$.

7. In the case considered in Ex. 5, the number of independent variables among $A, B, \ldots, S$ is the same as the number of independent variables among $E_1, \ldots, E_{27}$, namely the number of fundamental cycles. However in Ex. 6 there are four fundamental cycles and four variables A, B, C, D; in this case we have $A = 1 + E_5 - E_7 - E_8$, $B = E_5$, $C = E_5$, $D = 1 + E_6 - E_8$ so there are only *three* quantities which are actually independent in spite of the four independent variables E_5, E_6, E_7, E_8. The reason is that an increase in E_6 and E_8 together with an equivalent decrease in E_7 makes no change in A, B, C, D; stated another way, there are two ways to go from "start" to "stop" in which the *vertices* are touched the same number of times but the *edges* are not (e.g. Start–A–D–B–C–Stop, Start–A–B–C–D–Stop). Such pairs of paths are impossible in Fig. 3.

If we consider the case where the flowchart has many, many edges (e.g. nearly $\binom{n}{2}$ edges when there are n vertices), it is clear that there are many more fundamental cycles than vertices, and in fact Kirchhoff's law may be of *no help whatever* in detecting redundancies between the number of times each *vertex* is encountered. For example, if we add an edge from B to "stop" in the graph of Ex. 6, the quantities A, B, C, D are essentially independent of each other.

Unfortunately it appears to be difficult to tell in advance just how much help Kirchhoff's law will be if we are concerned about vertices instead of edges. Our beautiful theory works only for the edges! Indeed, it should be stressed that Kirchhoff's law is essentially directed only to edges; there is no easy analog to Eq. (1) in which only vertex variables, not edge variables, appear. In general when we set up equations like the "deductions" made between (7) and (8) in Section 1.3.3, these equations are being set up from the understood edge variables, not the vertex variables.

The moral of this story is that *Kirchhoff's law should not be regarded as a set of equations relating the vertex variables* (although the rather simple examples in Section 1.3.3 and elsewhere in this book may have given this impression); it is properly regarded as a set of relations between the edge variables. When the number of edges does not greatly exceed the number of vertices, Kirchhoff's law will yield important redundancies between the vertex variables, but the precise number is unclear.

8. Several approaches may be used; perhaps the simplest is the following: Each edge from a vertex to itself becomes a "fundamental cycle" all by itself; if there are k edges $e, e', \ldots, e^{(k)}$ between vertices V and V', make $k - 1$ fundamental cycles $e' \pm e, \ldots, e^{(k)} \pm e$ (choosing $+$ or $-$ according as the edges go in the opposite or the same direction), and then proceed as if only edge e were present.

Actually this situation would be much simpler conceptually if we had defined a graph in such a way that multiple edges are allowed between vertices, and edges are allowed from a vertex to itself; paths and cycles would be defined in terms of edges instead of vertices. This type of definition is, in fact, made in the following section for directed graphs.

9. (The following solution is based on the idea that we may print out each edge that does not make a cycle with the preceding edges.) Use Algorithm 2.3.3E, with each pair (a_i, b_i) representing $a_i \equiv b_i$ in the notation of that algorithm. The only change is to print (a_i, b_i) if $j \neq k$ in step E4.

To show this algorithm is valid, we must prove that (a) the algorithm prints out no edges that form a cycle, and (b) if G contains at least one free subtree, the algorithm prints out $n - 1$ edges. Define $j \equiv k$ if there exists a path from V_j to V_k or if $j = k$. This is clearly an equivalence relation, and moreover $j \equiv k$ if and only if this relation can be deduced from the equivalences $a_1 \equiv b_1, \ldots, a_m \equiv b_m$. Now (a) is valid since the algorithm prints out no edges that form a cycle with previously printed edges; (b) is true because if all vertices of G are equivalent, FATHER$[k] = 0$ for precisely one value of k.

10. If the terminals have all been connected together, the corresponding graph must be connected in the technical sense. A minimum number of wires clearly will involve no cycles, so we must have a free tree. By Theorem A, a free tree contains $n - 1$ wires, and a graph with n vertices and $n - 1$ edges is a free tree if and only if it is connected.

11. It is sufficient to prove that when $n > 1$ and $c(n - 1, n)$ is the minimum of the $c(i, n)$, there exists at least one minimum cost tree in which T_{n-1} is wired to T_n. (For, any minimum cost tree with $n > 1$ terminals and with T_{n-1} wired to T_n must also be a minimum cost tree with $n - 1$ terminals if we regard T_{n-1} and T_n as "common", using the stated convention in the algorithm.)

To prove the above statement, suppose we have a minimum cost tree in which T_{n-1} is not wired to T_n. If we add the wire $T_{n-1}T_n$ we obtain a cycle, and any of the other wires in that cycle may be removed; removing the other wire touching T_n gives us another tree, whose total cost is not greater than the original, in which $T_{n-1}T_n$ appears.

12. Keep two auxiliary tables, $a(i)$ and $b(i)$, for $1 \leq i < n$, representing the fact that the cheapest connection from T_i to a chosen terminal is to $T_{b(i)}$, and its cost is $a(i)$; initially $a(i) = c(i, n)$ and $b(i) = n$. Then do the following operation $n - 1$ times: Find i such that $a(i) = \min_{1 \leq j < n} a(j)$; connect T_i to $T_{b(i)}$; for $1 \leq j < n$ if $c(i, j) < a(i)$ set $a(i) \leftarrow c(i, j)$ and $b(i) \leftarrow j$; and set $a(i) \leftarrow \infty$.

(It is somewhat more efficient to avoid the use of ∞, keeping instead a one-way linked list of those j which have not yet been chosen. This straightforward improvement is left to the reader. With or without the improvement, the algorithm takes $O(n^2)$ operations.)

13. We must prove G is connected. If $V \neq V'$ and VV' is not an edge of G, add the edge VV' to G; this introduces a cycle which must involve the new edge, so it may be written $(V, V', V_2, \ldots, V)$; hence there is a path in G from V' to V.

14. If there is no path from V_i to V_j, for some $i \neq j$, then no product of the transpositions will move i to j. So if all permutations are generated, the graph must be connected. Conversely if it is connected, remove edges if necessary until we have a tree. Then renumber the vertices so that V_n is adjacent to only one other vertex, namely V_{n-1}. (See the proof of Theorem A.) Now the transpositions other than $(n - 1, n)$ form a tree with $n - 1$ vertices; so by induction if π is any permutation on $\{1, 2, \ldots, n\}$ which leaves n fixed, π can be written as a product of the transpositions other than $(n - 1, n)$. If π moves n to j then $\pi(j, n - 1)(n - 1, n) = \rho$ fixes n; hence $\pi = \rho(n - 1, n)(j, n - 1)$ can be written as a product of the given transpositions.

1. Let $(e_1, \ldots, e_n)$ be an oriented path of smallest possible length from V to V'. If now $\text{init}(e_j) = \text{init}(e_k)$ for $j < k$, $(e_1, \ldots, e_{j-1}, e_k, \ldots, e_n)$ would be a shorter path; a similar argument applies if $\text{fin}(e_j) = \text{fin}(e_k)$ for $j < k$. Hence $(e_1, \ldots, e_n)$ is simple.

2. Those cycles in which all signs are the same: C_0, C_8, C''_{13}, C_{17}, C''_{19}, C_{20}.

3. For example, use three vertices A, B, C, with arcs from B to A and to C.

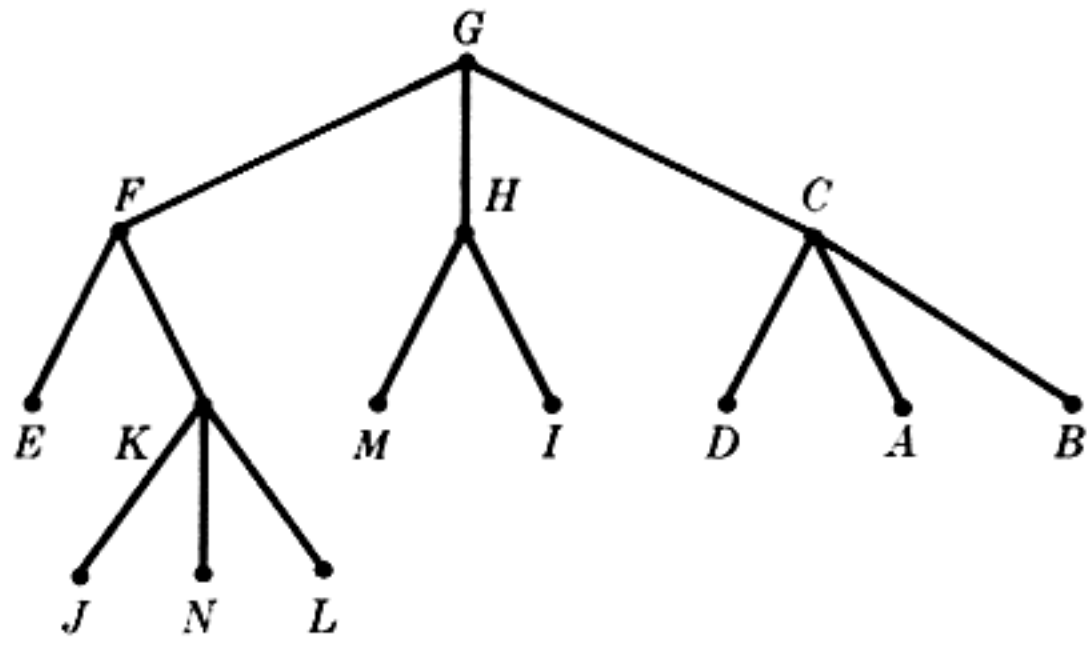
4. If there are no oriented cycles, Algorithm 2.2.3T topologically sorts G. If there is an oriented cycle, topological sorting is clearly impossible. (Depending on how this exercise is interpreted, oriented cycles of length 1 could be excluded from the above discussion.)

5. Let k be the smallest integer such that $\text{fin}(e_k) = \text{init}(e_j)$ for some $j \leq k$. Then $(e_j, \ldots, e_k)$ is an oriented cycle.

6. False (on a technicality), just because there may be several different arcs from one vertex to another.

7. True for finite directed graphs: For if we disregard the direction of the arcs we have a graph with n vertices, $n - 1$ edges, and no cycles. (Any cycle that is not an oriented cycle includes a vertex that is the initial vertex of two arcs, and this would contradict (a).) For infinite graphs the result is obviously false since we might have vertices R, V_1, V_2, V_3, ... and arcs from V_j to V_{j+1} for $j \geq 1$.

9. All arcs point upward.



10. **G1.** Set $k \leftarrow F[j]$, $F[j] \leftarrow 0$.

G2. If $k = 0$, stop; otherwise set $m \leftarrow F[k]$, $F[k] \leftarrow j$, $j \leftarrow k$, $k \leftarrow m$, and repeat step G2. ▮

11. This algorithm combines Algorithm 2.3.3E with the method of the preceding exercise, so that all oriented trees have arcs which correspond to actual arcs in the directed graph; $S[j]$ is an auxiliary table which tells whether an arc goes from j to $F[j]$ ($S[j] = +1$) or from $F[j]$ to j ($S[j] = -1$). Initially $F[1] = \cdots = F[n] = 0$. The following steps may be used to process each arc (a, b):

C1. Set $j \leftarrow a$, $k \leftarrow F[j]$, $F[j] \leftarrow 0$, $s \leftarrow S[j]$.

C2. If $k = 0$, go to C3; otherwise set $m \leftarrow F[k]$, $t \leftarrow S[k]$, $F[k] \leftarrow j$, $S[k] \leftarrow -s$, $s \leftarrow t$, $j \leftarrow k$, $k \leftarrow m$, and repeat step C2.

C3. (Now a appears as the root of its tree.) Set $j \leftarrow b$, and then if $F[j] \neq 0$ repeatedly set $j \leftarrow F[j]$ until $F[j] = 0$.

C4. If $j = a$, go to C5; otherwise set $F[a] \leftarrow b$, $S[a] \leftarrow +1$, print (a, b) as an arc belonging to the free subtree, and terminate.

C5. Print "CYCLE" followed by "(a, b)".

C6. If $F[b] = 0$, terminate. Otherwise if $S[b] = +1$, print "$+(b, F[b])$", else print "$-(F[b], b)$"; set $b \leftarrow F[b]$ and repeat step C6. ∎

12. It equals the in-degree; the out-degree of each vertex can be only 0 or 1.

13. Define a sequence of oriented subtrees of G as follows: G_0 is the vertex R alone. G_{k+1} is G_k, plus any vertex V of G that is not in G_k but for which there is an arc from V to V' where V' *is* in G_k, plus one such arc $e[V]$ for each such vertex. It is immediate by induction that G_k is an oriented tree for all $k \geq 0$, and that if there is an oriented path of length k from V to R in G then V is in G_k. Therefore G_∞, the set of all V and $e[V]$ in any of the G_k, is the desired oriented subtree of G.

14. $(e_{12}, e_{20}, e_{00}, e'_{01}, e_{10}, e_{01}, e'_{12}, e_{22}, e_{21})$, $(e_{12}, e_{20}, e_{00}, e'_{01}, e'_{12}, e_{22}, e_{21}, e_{10}, e_{01})$,
$(e_{12}, e_{20}, e'_{01}, e_{10}, e_{00}, e_{01}, e'_{12}, e_{22}, e_{21})$, $(e_{12}, e_{20}, e'_{01}, e'_{12}, e_{22}, e_{21}, e_{10}, e_{00}, e_{01})$,
$(e_{12}, e_{22}, e_{20}, e_{00}, e'_{01}, e_{10}, e_{01}, e'_{12}, e_{21})$, $(e_{12}, e_{22}, e_{20}, e_{00}, e'_{01}, e'_{12}, e_{21}, e_{10}, e_{01})$,
$(e_{12}, e_{22}, e_{20}, e'_{01}, e_{10}, e_{00}, e_{01}, e'_{12}, e_{21})$, $(e_{12}, e_{22}, e_{20}, e'_{01}, e'_{12}, e_{21}, e_{10}, e_{00}, e_{01})$,

in "lexicographic order"; the eight possibilities come from the independent choices of which of e_{00} or e'_{01}, e_{10} or e'_{12}, e_{21} or e_{22}, should precede the other.

15. If it is connected and balanced, it either has only one vertex or there is an Eulerian circuit which touches all the vertices; twice around that circuit will touch any given vertex V the first time and any other given vertex V' the second time.

16. Consider the directed graph G with vertices $V_1, \ldots, V_{13}$ and with an arc from V_j to V_k for each k in pile j. Winning the game is equivalent to the existence of an Eulerian circuit in this directed graph (for if the game is won the final card turned up must come from the center; this graph is balanced). Now if the game is won, we have an oriented subsubtree by Lemma E. Conversely if the stated configuration is an oriented subtree, the game is won by Theorem D.

17. $\frac{1}{13}$. This answer can be obtained, as the author first obtained it, by laborious enumeration of oriented trees of special types and the application of generating functions, etc., based on the methods of Section 2.3.4.4; it also follows easily from the following simple, direct proof: Define an order for turning up *all* cards of the deck, as follows: Obey the rules of the game until getting stuck, then "cheat" by turning up the first available card (find the first pile that is not empty, going clockwise from pile 1) and continue as before, until eventually all cards have been turned up. The cards *in the order of turning up* are in completely random order (since the value of a card need not be specified until after it is turned up). So the problem is just to calculate the probability that in a randomly shuffled deck the last card is a king. More generally the probability that k cards are still face down when the game is over is the probability that the last king in a random shuffle is followed by k cards, namely $4!\binom{51-k}{3}\frac{48!}{52!}$. Hence a man playing this game without cheating will turn up an average of exactly 42.4 cards per game. *Note:* Similarly, it can be shown that the probability

that the player will have to "cheat" k times in the process described above is exactly given by the Stirling number $\left[{13 \atop k+1}\right]/13!$. (See Section 1.2.10, Eq. (9) and Ex. 7; the case of a more general deck is considered in Ex. 18.)

18. (a) If there is a cycle $(V_0, V_1, \ldots, V_k)$, where necessarily $3 \le k \le n$, the sum of the k rows of A corresponding to the k edges of this cycle is a row of zeroes; so if G is not a free tree the determinant of A_0 is zero.

Now if G is a free tree we may regard it as an ordered tree with root V_0, and we can rearrange the rows and columns of A_0 so that columns are in preorder and so that the kth row corresponds to the edge from the kth vertex (column) to its father; then the matrix is triangular with ± 1 in each diagonal position, so the determinant is ± 1.

(b) By the Binet-Cauchy formula (Ex. 1.2.3–46) we have

$$\det A_0^T A_0 = \sum_{1 \le i_1 < \cdots < i_n \le m} (\det A_{i_1 \ldots i_n})^2$$

where $A_{i_1 \ldots i_n}$ represents a matrix consisting of rows $i_1, \ldots, i_n$ of A_0 (thus corresponding to a choice of n edges of G). The result now follows from (a).

19. (a) The conditions $a_{00} = 0$ and $a_{jj} = 1$ are just conditions (a), (b) of the definition of oriented tree. If G is not an oriented tree there is an oriented cycle (by Ex. 7), and this means the rows of A_0 corresponding to the vertices in this oriented cycle sum to a row of zeroes; hence $\det A_0 = 0$. If G is an oriented tree, assign an arbitrary order to the sons of each family and regard G as an ordered tree. Now permute rows and columns of A_0 until they correspond to preorder of the vertices. Since the same permutation has been applied to the rows as to the columns, the determinant is unchanged; and the resulting matrix is triangular with $+1$ in every diagonal position.

(b) We may assume $a_{0j} = 0$ for all j, since no arc emanating from V_0 can participate in an oriented subtree. We may also assume $a_{jj} > 0$ for all $j \ge 1$ since otherwise the whole jth row is zero and there obviously are no oriented subtrees. Now use induction on the number of arcs: If $a_{jj} > 1$ let e be some arc leading from V_j; let B_0 be a matrix like A_0 but with arc e deleted, and let C_0 be the matrix like A_0 but with all arcs *except* e that lead from V_j deleted.
Example: $A_0 = \left(\begin{smallmatrix} 3 & -2 \\ -1 & 2 \end{smallmatrix}\right)$, $j = 1$, $e =$ arc from V_1 to V_0; then $B_0 = \left(\begin{smallmatrix} 2 & -2 \\ -1 & 2 \end{smallmatrix}\right)$, $C_0 = \left(\begin{smallmatrix} 1 & 0 \\ -1 & 2 \end{smallmatrix}\right)$. Then $\det A_0 = \det B_0 + \det C_0$, since the matrices agree in all rows except row j and in that row A_0 is the sum of B_0, C_0. Moreover, the number of oriented subtrees of G is the number of subtrees which do *not* use e (namely, $\det B_0$, by induction) plus the number which *do* use e (namely, $\det C_0$).

20. Using Ex. 18 we find $B = A_0^T A_0$. Or, using Ex. 19, B is the matrix A_0 for the directed graph G' with two arcs (one in each direction) in place of each edge of G; and each free subtree of G corresponds uniquely to an oriented subtree of G' with root V_0 (the directions of the arcs are determined by the choice of root).

21. (This result may be derived from interesting but considerably more complicated arguments used in the paper of van Aardenne-Ehrenfest and de Bruijn quoted in the text. The following derivation is not only simpler, it also may be generalized to determine the number of oriented subtrees of G^* when G is an *arbitrary* directed graph; see D. E. Knuth, *Journal of Combinatorial Theory* **3** (1967), 309–314.)

Construct the matrices A and A^* as in Ex. 19. For the example graphs G, G^* in Figs. 3 and 4,

$$A = \begin{pmatrix} 2 & -2 & 0 \\ -1 & 3 & -2 \\ -1 & -1 & 2 \end{pmatrix}$$

$$A^* = \begin{array}{c} \\ [00] \\ [10] \\ [10] \\ [01] \\ [01] \\ [21] \\ [12] \\ [12] \\ [22] \end{array} \begin{array}{c} \begin{array}{ccccccccc} [00] & [10] & [10] & [01] & [01] & [21] & [12] & [12] & [22] \end{array} \\ \left(\begin{array}{ccc|ccc|ccc} 2 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 3 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 3 & -1 & -1 & 0 & 0 & 0 & 0 \\ \hline 0 & -1 & 0 & 3 & 0 & 0 & -1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 3 & 0 & -1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 3 & -1 & -1 & 0 \\ \hline 0 & 0 & -1 & 0 & 0 & -1 & 3 & 0 & -1 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 3 & -1 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 2 \end{array}\right) \end{array}$$

Add the indeterminate λ to the upper left corner element of A and A^* (in the example this gives $2 + \lambda$ in place of 2). If $t(G)$, $t(G^*)$ are the numbers of oriented subtrees of G and G^* we have $t(G) = (1/\lambda)(n+1) \det A$, $t(G^*) = (1/\lambda)m(n+1) \det A^*$. (The number of oriented subtrees of a balanced graph is the same for any given root, e.g. by Ex. 22.)

If we group vertices V_{jk} for equal k the matrix A^* can be partitioned as shown above. Let $B_{kk'}$ be the submatrix of A^* consisting of the rows for V_{jk} and the columns for $V_{j'k'}$, for all j and j' such that V_{jk} and $V_{j'k'}$ are in G^*. By adding the 2nd, . . . , mth columns of each submatrix to the first column and then subtracting the first row of each submatrix from the 2nd, . . . , mth rows, the matrix A^* is transformed so that

$$B_{kk'} = \begin{pmatrix} a_{kk'} & * & \dots & * \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 0 \end{pmatrix} \text{ for } k \neq k', \qquad B_{kk} = \begin{pmatrix} a_{kk} + \lambda\delta_{k0} & * & \dots & * \\ -\lambda\delta_{k0} & m & & 0 \\ \vdots & & \ddots & \vdots \\ -\lambda\delta_{k0} & 0 & \dots & m \end{pmatrix}.$$

Here "*" indicates values which are more or less irrelevant. It follows that $\det A^*$ is $m^{m(n-1)}$ times the determinant of

$$\begin{pmatrix} \lambda + a_{00} & * & * & \dots & * & a_{01} & \dots & a_{0m} \\ -\lambda & m & 0 & \dots & 0 & 0 & \dots & 0 \\ \vdots & & & & & & & \\ -\lambda & 0 & 0 & \dots & m & 0 & \dots & 0 \\ a_{10} & * & * & \dots & * & a_{11} & \dots & a_{1n} \\ \vdots & & & & & & & \\ a_{n0} & * & * & \dots & * & a_{n1} & \dots & a_{nn} \end{pmatrix}$$

The asterisks left are all zero except for precisely one -1 in each column. Add the last n rows to the top row, and expand the determinant by the first row, to get $m^{n(m-1)+m-1} \det A - (m-1)m^{n(m-1)+m-2} \det A$.

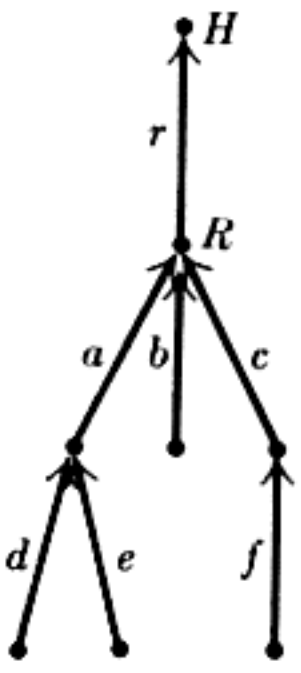
22. The total number is $(\sigma_1 + \cdots + \sigma_n)$ times the number of Eulerian circuits starting with a given edge e_1, where init $(e_1) = V_1$. Each such circuit determines an oriented subtree with root V_1 by Lemma E, and for each of the T oriented subtrees there are $\prod_{1 \le j \le n} (\sigma_j - 1)!$ paths satisfying the three conditions of Theorem D, corresponding to the different order in which the arcs $\{e \mid \text{init}\,(e) = V_j,\ e \neq e[V_j],\ e \neq e_1\}$ are entered into P. (Cf. Ex. 14.)

23. Construct the directed graph G_k with m^{k-1} vertices as in the hint, and let $[x_1, \ldots, x_k]$ denote the arc mentioned there. For each function that has maximum period length, we can define a unique corresponding Eulerian circuit, by letting $f(x_1, \ldots, x_k) = x_{k+1}$ if arc $[x_1, \ldots, x_k]$ is followed by $[x_2, \ldots, x_{k+1}]$. (We regard Eulerian circuits as being the same if one is just a cyclic permutation of the other.) Now $G_k = G_{k-1}^*$ in the sense of Ex. 21, so G_k has $m^{m^{k-1}-m^{k-2}}$ times as many oriented subtrees as G_{k-1}; by induction G_k has $m^{m^{k-1}-1}$ oriented subtrees, and $m^{m^{k-1}-k}$ with a given root. Therefore by Ex. 22 the number of functions with maximum period, i.e. the number of Eulerian circuits of G_k starting with a given arc, is $m^{-k}(m!)^{m^{k-1}}$.

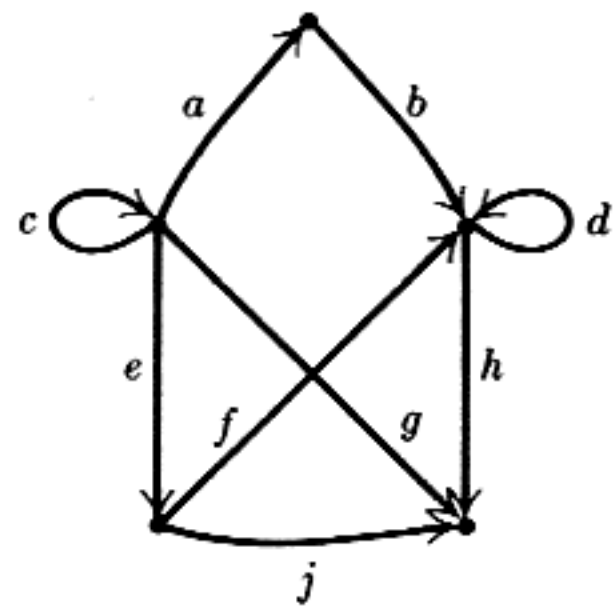
24. Define a new directed graph having E_j copies of e_j, for $0 \le j \le m$. This graph is balanced and so we know there is an Eulerian circuit $(e_0, \ldots)$ by Theorem G. The desired oriented path comes by deleting the edge e_0 from this Eulerian circuit.

25. Assign an order to the sets of arcs having common initial vertices, and assign an order to the sets of arcs having common final vertices. Now for each arc e, let the fields in the node representing e be the following: If e' is the next arc (in the assumed ordering) for which init (e') = init (e), let ALINK point to e' and let ATAG = "+"; if e is the last arc (in the assumed ordering) with this initial vertex, however, let ATAG = "−" and let ALINK be a pointer to the first arc e' for which init (e) = fin (e'); if no such e' exists, let ALINK = Λ. Define BLINK and BTAG by the same rules, reversing the roles of *init* and *fin*.

Examples:



arc	ALINK	ATAG	BLINK	BTAG
a	d	−	b	+
b	Λ	−	c	+
c	f	−	r	−
d	Λ	−	e	+
e	Λ	−	a	−
f	Λ	−	c	−
r	a	−	Λ	−



arc	ALINK	ATAG	BLINK	BTAG
a	c	+	b	−
b	a	−	d	+
c	e	+	a	−
d	h	+	f	+
e	g	+	f	−
f	j	+	h	−
g	c	−	h	+
h	b	−	j	+
j	e	−	Λ	−

Note: If in the oriented tree representation we add another arc from H to itself, we get an interesting situation; either we get the standard conventions 2.3.1–(7) with LLINK, LTAG, RLINK, RTAG *interchanged* in the list head, or (if the new arc is placed last in the ordering) we get the standard conventions except RTAG = "+" in the node associated with the root of the tree!

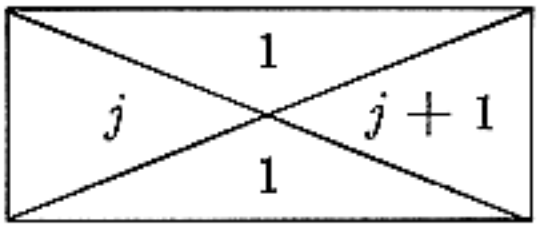
This exercise is based on an idea communicated to the author by W. C. Lynch. It would be interesting to explore further properties of this representation, e.g., to compare tree-traversal algorithms with the Eulerian circuit constructions of this section.

SECTION 2.3.4.3

1. The root is the empty sequence; arcs go from $(x_1, \ldots, x_n)$ to $(x_1, \ldots, x_{n-1})$.

2. Take one domino type and rotate it 180° to get another domino type; these two types give an obvious way to tile the plane (without further rotations) by replication of a 2×2 pattern.

3. Consider the set of domino types



for all positive integers j. Then the upper half plane may be tiled in uncountably many ways; but whatever square is placed in the center of the plane puts a finite limit on the distance it can be continued to the left.

4. Systematically enumerate all possible ways to tile an $n \times n$ block, for $n = 1, 2, \ldots$, looking for toroidal solutions within these blocks. If there is no way to tile the plane, the infinity lemma tells us there is an integer n with no $n \times n$ solutions. If there *is* a way to tile the plane, the assumption tells us there is an n with an $n \times n$ solution containing a rectangle that yields a toroidal solution. Hence in either case the algorithm will terminate. (But the stated assumption is false, as shown in the next exercise, and in fact there is no algorithm which will determine in a finite number of steps whether or not there exists a way to tile the plane with a given set of types.)

5. Start by noticing that we need classes $\begin{smallmatrix}\alpha & \beta \\ \gamma & \delta\end{smallmatrix}$ replicated in 2×2 groups in any solution. Then, step 1: Consider just the α squares; we show that the pattern $\begin{smallmatrix}a & b \\ c & d\end{smallmatrix}$ must be replicated in 2×2 groups of α squares. Step $n > 1$: We determine a pattern that must appear in a cross-shaped region of height and width $2^n - 1$. The middle of the crosses has the pattern $\begin{smallmatrix}Na & Nb \\ Nc & Nd\end{smallmatrix}$ replicated throughout the plane.

For example, after step 3 we will know the contents of 7×7 blocks throughout the plane, separated by unit length strips, every eight units. The 7×7 blocks which are of class Na in the center have the form

αa	βKQ	αb	βQP	αa	βBK	αb
γPJ	δNa	γRB	δQK	γLJ	δNb	γPB
αc	βDS	αd	βQTY	αc	βBS	αd
γPQ	δPJ	γPXB	δNa	γRQ	δRB	γRB
αa	βUK	αb	βDP	αa	βBK	αb
γTJ	δNc	γSB	δDS	γST	δNd	γTB
αc	βQS	αd	βDT	αc	βBS	αd

The middle column and the middle row is the "cross" just filled in during step 3; the other four 3×3 squares were filled in after step 2; the squares just to the right and below this 7×7 square are part of a 15×15 cross to be filled in at step 4.

6. Let k and m be fixed. Consider an oriented tree whose vertices each represent, for some n, one of the partitions of $\{1, \ldots, n\}$ into k parts, containing no arithmetic progression of length m. A node that partitions $\{1, \ldots, n+1\}$ is a son of one for $\{1, \ldots, n\}$ if the two partitions agree on $\{1, \ldots, n\}$. If there were an infinite path from the root we would have a way to divide *all* integers into k sets with no arithmetic progression of length m. Hence, by the infinity lemma and van der Waerden's theorem, this tree is finite. (If $k = 2$, $m = 3$, the tree can be rapidly calculated by hand, and the least value of N is 9.)

7. There exist two sets S_0, S_1 which partition the integers such that neither contains any infinite *computable* sequence (cf. Ex. 3.5–32). So in particular there is no infinite arithmetic progression. Theorem K does not apply because there is no way to put partial solutions into a tree with finite degrees at each vertex.

8. Let a "counterexample sequence" be an infinite sequence of trees that violates Kruskal's theorem, if such sequences exist. Assume the theorem is false; then let T_1 be a tree with the smallest possible number of nodes such that T_1 can be the first tree in a counterexample sequence; if $T_1, \ldots, T_j$ have been chosen, let T_{j+1} be a tree with the smallest possible number of nodes such that $T_1, \ldots, T_j, T_{j+1}$ is the beginning of a counterexample sequence. This process defines a counterexample sequence $T_1, T_2, \ldots$. None of these T's is just a root. Now, we look at this sequence very carefully:

(a) Suppose there is a subsequence $T_{n_1}, T_{n_2}, \ldots$ for which $l(T_{n_1}), l(T_{n_2}), \ldots$ is a counterexample sequence. This is impossible, otherwise $T_1, \ldots, T_{n_1-1}, l(T_{n_1}), l(T_{n_2}), \ldots$ would be a counterexample sequence, contradicting the definition of T_{n_1}.

(b) Because of (a), there are only finitely many j for which $l(T_j)$ cannot be embedded in $l(T_k)$ for any $k > j$. Therefore by taking n_1 larger than any such j we may find a subsequence for which $l(T_{n_1}) \subseteq l(T_{n_2}) \subseteq l(T_{n_3}) \subseteq \cdots$.

(c) Now by the result of Ex. 2.3.2–22, $r(T_{n_j})$ cannot be embedded in $r(T_{n_k})$ for any $k > j$, else $T_{n_j} \subseteq T_{n_k}$. Therefore $T_1, \ldots, T_{n_1-1}, r(T_{n_1}), r(T_{n_2}), \ldots$ is a counterexample sequence. But this contradicts the definition of T_{n_1}. This proves Kruskal's theorem.

Note: Kruskal's theorem does not seem to follow simply from the infinity lemma, although they seem to be related in a vague way; there are in general infinitely many trees T such that $T_1 \not\subseteq T$, $T_2 \not\subseteq T$, $\ldots$, $T_n \not\subseteq T$ when $T_1, T_2, \ldots, T_n$ are given.

SECTION 2.3.4.4

1. $\ln A(z) = \ln z + \sum_{k\geq 1} a_k \ln\left(\frac{1}{1-z^k}\right) = \ln z + \sum_{k,t\geq 1} \frac{a_k z^{kt}}{t} = \ln z + \sum_{t\geq 1} \frac{A(z^t)}{t}.$

2. By differentiation, and equating the coefficients of z^n, we obtain the identity

$$na_{n+1} = \sum_{k\geq 1} \sum_{d\backslash k} d a_d a_{n+1-k}.$$

Now interchange the order of summation.

4. (a) $A(z)$ certainly converges at least for $|z| < \frac{1}{4}$, since a_n is less than the number of *ordered* trees b_{n-1}. Since $A(1)$ is infinite and all coefficients are positive, there is a positive number $\alpha \leq 1$ such that $A(z)$ converges for $|z| < \alpha$, and there is a singularity at $z = \alpha$. Let $\psi(z) = (1/z)A(z)$; since $\psi(z) > e^{z\psi(z)}$, we see $\psi(z) = m$ implies $z < \ln m/m$, so $\psi(z)$ is bounded and $\lim_{z\to\alpha-}\psi(z)$ exists. Thus $\alpha < 1$, and by Abel's limit theorem $a = \alpha \cdot \exp\left(a + \frac{1}{2}A(\alpha^2) + \frac{1}{3}A(\alpha^3) + \cdots\right)$.

(b) $A(z^2), A(z^3), \ldots$ are analytic for $|z| < \sqrt{\alpha}$, and $\frac{1}{2}A(z^2) + \frac{1}{3}A(z^3) + \cdots$ converges uniformly in a slightly smaller disk.

(c) If $\partial F/\partial w = a - 1 \neq 0$, the implicit function implies there is an analytic function $f(z)$ in a neighborhood of $(\alpha, a/\alpha)$ such that $F\big(z, f(z)\big) = 0$. But this implies $f(z) = (1/z)A(z)$, contradicting the fact that $A(z)$ is singular at α.

(d) Obvious.

(e) $\partial F/\partial w = A(z) - 1$ and $|A(z)| < A(\alpha) = 1$ since the coefficients of $A(z)$ are all positive. Hence as in (c), $A(z)$ is regular at all such points.

(f) Near $(\alpha, 1/\alpha)$ we have the identity $0 = \beta(z - \alpha) + (\alpha/2)(w - 1/\alpha)^2 +$ higher order terms, where $w = (1/z)A(z)$; so w is an analytic function of $\sqrt{z - \alpha}$ here by the implicit function theorem. Consequently there is a region $|z| < \alpha_1$ minus a cut $[\alpha, \alpha_1]$ in which $A(z)$ has the stated form. (The minus sign is chosen since a plus sign would make the coefficients ultimately negative.)

(g) Any function of the stated form has coefficients asymptotically

$$\frac{\sqrt{2\beta}}{\alpha^n}\binom{1/2}{n}. \quad \left(\text{Note that } \binom{3/2}{n} = O\left(\frac{1}{n}\binom{1/2}{n}\right).\right)$$

For further details, and asymptotic values of the number of free trees, see the cited article.

5.
$$c_n = \sum_{j_1+2j_2+\cdots=n} \binom{c_1+j_1-1}{j_1} \cdots \binom{c_n+j_n-1}{j_n} - c_n, \qquad n > 1.$$

Therefore

$$\begin{aligned} 2C(z) + 1 - z &= (1-z)^{-c_1}(1-z^2)^{-c_2}(1-z^3)^{-c_3}\cdots \\ &= \exp\left(C(z) + \tfrac{1}{2}C(z^2) + \cdots\right). \end{aligned}$$

We find $C(z) = z + z^2 + 2z^3 + 5z^4 + 12z^5 + 33z^6 + 90z^7 + \cdots$. There is no obvious connection with $A(z)$, although it is plausible some relation does exist.

6. $zG(z)^2 = 2G(z) - 2 - zG(z^2)$; $G(z) = 1 + z + z^2 + 2z^3 + 3z^4 + 6z^5 + 11z^6 + 23z^7 + \cdots$. See *AMM* **56** (1949), 697–699 for references.

7. $g_n \sim ca^n n^{-3/2}$, where $c = .791603$, $a = 2.48325$.

8.

9. If there are two centroids, by considering a path from one to the other we find there can't be intermediate points, so any two centroids are adjacent. It is impossible for a tree to contain three mutually adjacent vertices, so there are at most two.

10. If X, Y are adjacent, let $s(X, Y)$ be the number of vertices in the Y-subtree of X. Then $s(X, Y) + s(Y, X) = n$. The argument in the text shows that if Y is a centroid, $\text{height}(X) = s(X, Y)$. Therefore if both X and Y are centroids, $\text{height}(X) = \text{height}(Y) = n/2$.

In terms of this notation, the argument in the text goes on to show that if $s(X, Y) \geq s(Y, X)$, there is a centroid in the Y subtree of X. So if two free trees with m vertices are joined by an edge between X and Y, we obtain a free tree in which $s(X, Y) = m = s(Y, X)$, and there must be two centroids (namely X and Y).

11. $zT(z)^t = T(z) - 1$; i.e., $z + T(z)^{-t} = T(z)^{1-t}$. By Eq. 1.2.9–21, $T(z) = \sum_n A_n(1, -t)z^n$, so the number of t-ary trees is

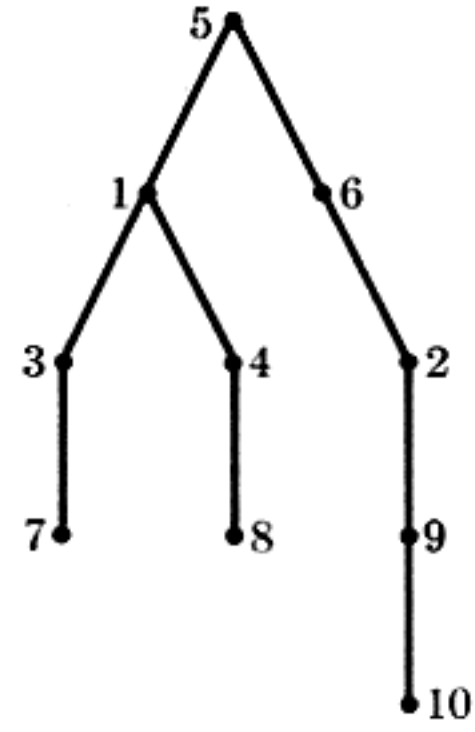
$$\binom{1+tn}{n}\frac{1}{1+tn} = \binom{tn}{n}\frac{1}{(t-1)n+1}.$$

12. Consider the directed graph which has one arc from V_i to V_j for all $i \neq j$. The matrix A_0 of Ex. 2.3.4.2–19 is a combinatorial $(n-1) \times (n-1)$ matrix with $n-1$ on the diagonal and -1 off the diagonal. So its determinant is

$$(n + (n-1)(-1))n^{n-2} = n^{n-2},$$

the number of oriented trees with a given root. (Ex. 2.3.4.2–20 could also be used.)

13.

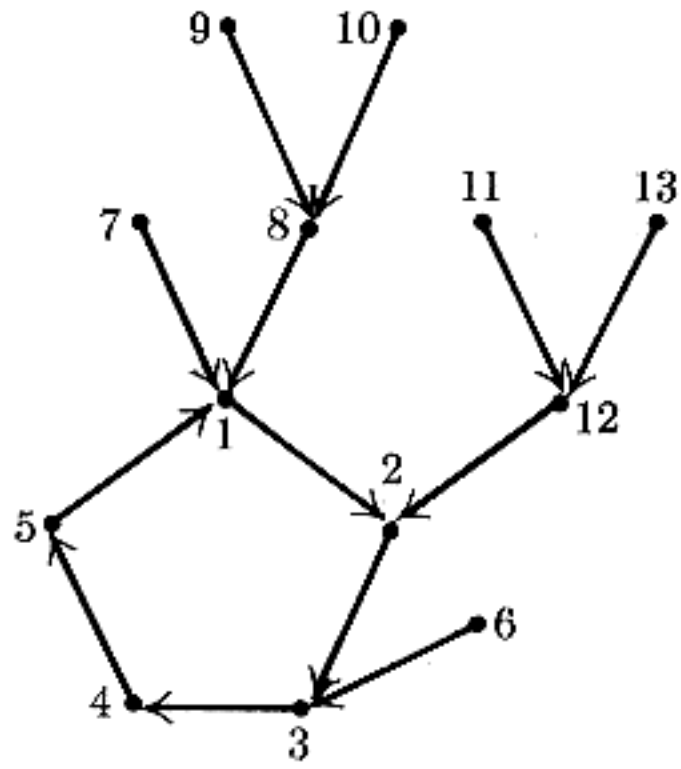


14. True, since the root will not become a leaf until all other branches have been removed.

15. In the canonical representation, $V_1, V_2, \ldots, V_{n-1}, f(V_{n-1})$ is a topological sort of the oriented tree considered as a directed graph, but this order would not in general be output by Algorithm 2.2.3T. Algorithm 2.2.3T can be changed so that it determines the values of $V_1, V_2, \ldots, V_{n-1}$ if the "insert into queue" operation of step T6 is replaced by a procedure which adjusts links so that the entries of the list appear in ascending order from front to rear.

17. There must be exactly one cycle $x_1, x_2, \ldots, x_k$ where $f(x_j) = x_{j+1}$ and $f(x_k) = x_1$. We will enumerate all f having a cycle of length k such that the iterates of each x ultimately come into this cycle. Define the canonical representation $f(V_1), f(V_2), \ldots, f(V_{m-k})$ as in the text; now $f(V_{m-k})$ is in the cycle, so we continue to get a "canonical representation" by writing down the rest of the cycle $f(f(V_{m-k}))$, $f(f(f(V_{m-k})))$, etc.

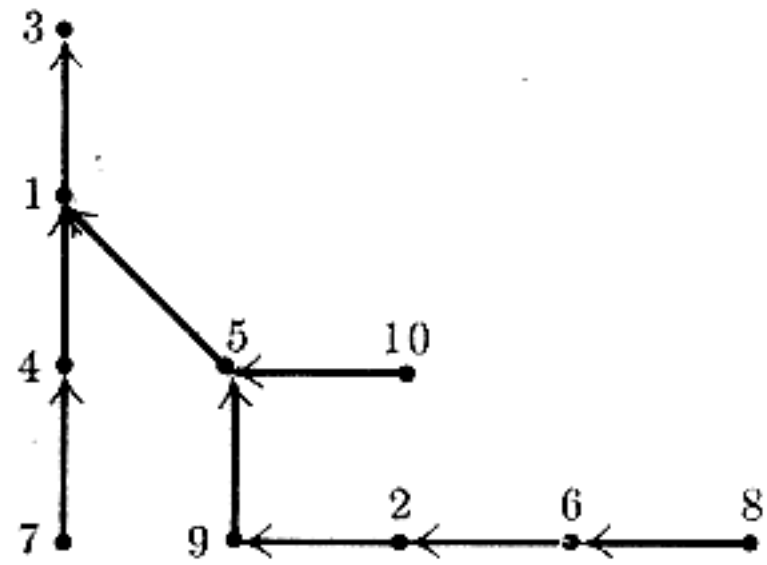
For example, the function with $m = 13$ whose graph is



leads to the representation 3, 1, 8, 8, 1, 12, 12, 2, 3, 4, 5, 1. We obtain a sequence of $m - 1$ numbers in which the last k are distinct. Conversely, from any such sequence we can reverse the construction (assuming k is known), hence there are precisely $m(m-1)\cdots(m-k+1)m^{m-k-1}$ such functions having a k-cycle.

18. To reconstruct the tree from a sequence $s_1, s_2, \ldots, s_{n-1}$, begin with s_1 as the root and successively attach arcs to the tree which point to $s_1, s_2, \ldots$; if vertex s_k has appeared earlier, leave the initial vertex of the arc leading to s_{k-1} nameless, otherwise give this vertex the name s_k. After all $n - 1$ arcs have been placed, give names to all vertices which remain nameless by using the numbers that have not yet appeared, and assigning names in increasing order to nameless vertices in the order of their creation.

For example from 3, 1, 4, 1, 5, 9, 2, 6, 5 we would construct



There is no simple connection between this method and the one in the text. Several more representations are possible; see the article by E. H. Neville, *Proc. Cambridge Phil. Soc.* 49 (1953), 381–385.

19. The canonical representation will have precisely $n - k$ different values, so we enumerate the sequences of $n - 1$ numbers with this property. The answer is $n(n-1)(n-2)\cdots(k+1)\left\{{n-1 \atop n-k}\right\}$. (See Section 3.3.2D.)

20. Consider the canonical representation of such trees. We are asking how many terms of $(x_1 + \cdots + x_n)^{n-1}$ have k_0 exponents zero, k_1 exponents one, etc. This is plainly the coefficient of such a term times the number of such terms, namely $(n-1)!/(0!)^{k_0}(1!)^{k_1}\ldots(n!)^{k_n}$ times $n!/k_0!k_1!\ldots k_n!$.

21. There are none with an even number of vertices; if there are $n = 2m + 1$ vertices, the answer is obtained from Ex. 20 with $k_0 = m + 1$, $k_2 = m$: $\binom{2m+1}{m}(2m)!/2^m$.

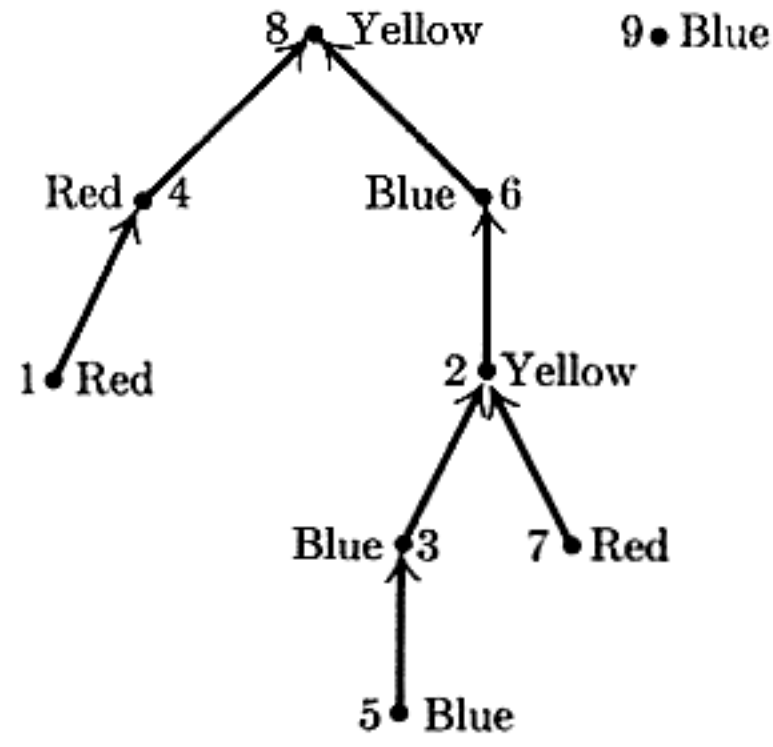
22. Exactly n^{n-2}; for if X is a particular vertex, the free trees are in one-to-one correspondence with oriented trees having root X.

23. It is possible to put the labels on every unlabeled, ordered tree in $n!$ ways, and each of these labeled, ordered trees is distinct. So the total number is $n!b_{n-1} = (2n - 2)!/(n - 1)!$.

24. There are as many with one given root as with another, so the answer in general is $1/n$ times the answer in Ex. 23; and in this particular case the answer is 30.

25. For $0 \leq q < n$, $r(n, q) = (n - q)n^{q-1}$. (Special case $s = 1$ in Eq. (22).)

26.



27. Given a function g from $\{1, 2, \ldots, r\}$ to $\{1, 2, \ldots, q\}$ such that adding arcs from V_k to $U_{g(k)}$ introduces no oriented cycles, construct a sequence $a_1, \ldots, a_r$ as follows: Call vertex V_k "free" if there is no oriented path from V_j to V_k for any $j \neq k$. Since there are no oriented cycles, there must be at least one free vertex. Let b_1 be the smallest integer for which V_{b_1} is free; and assuming $b_1, \ldots, b_t$ have been chosen, let b_{t+1} be the smallest integer different from $b_1, \ldots, b_t$ for which $V_{b_{t+1}}$ is free in the graph obtained by deleting the arcs from V_{b_k} to $U_{g(b_k)}$ for $1 \leq k \leq t$. This rule defines a permutation $b_1, \ldots, b_r$ of the integers $\{1, 2, \ldots, r\}$. Let $a_k = g(b_k)$ for $1 \leq k \leq r$; this defines a sequence such that $1 \leq a_k \leq q$ for $1 \leq k < r$, and $1 \leq a_r \leq p$.

Conversely if such a sequence $a_1, \ldots, a_r$ is given, call a vertex V_k "free" if there is no j for which $a_j > p$ and $f(a_j) = k$. Since $a_r \leq p$ there are at most $r - 1$ non-free vertices. Let b_1 be the smallest integer for which V_{b_1} is free; and assuming $b_1, \ldots, b_t$ have been chosen let b_{t+1} be the smallest integer different from $b_1, \ldots, b_t$ for which $V_{b_{t+1}}$ is free with respect to the sequence $a_{t+1}, \ldots, a_r$. This rule defines a permutation $b_1, \ldots, b_r$ of the integers $\{1, 2, \ldots, r\}$. Let $g(b_k) = a_k$ for $1 \leq k \leq r$; this defines a function such that adding arcs from V_k to $U_{g(k)}$ introduces no oriented cycles.

28. Let f be any of the n^{m-1} functions from $\{2, \ldots, m\}$ to $\{1, 2, \ldots, n\}$, and consider the directed graph with vertices $U_1, \ldots, U_m, V_1, \ldots, V_n$ and arcs from U_k to $V_{f(k)}$ for $1 < k \leq m$. Apply Ex. 27 with $p = 1$, $q = m$, $r = n$, to show there are m^{n-1} ways to add further arcs from the V's to the U's to obtain an oriented tree with root U_1. Since there is a one-to-one correspondence between the desired set of

free trees and the set of oriented trees with root U_1, the answer is $n^{m-1}m^{n-1}$. [*Note:* This construction can be extensively generalized; see D. Knuth, *Canadian J. Math.* **20** (1968), 1077–1086.]

29. If $y = x^t$, then $(tz)y = \ln y$, and we see it is sufficient to prove the identity for $t = 1$. Now if $zx = \ln x$ we know by Ex. 25 that $x^m = \sum_k E_k(m, 1)z^k$ for nonnegative integers m. Now

$$x^r = e^{zxr} = \sum_k \frac{(zxr)^k}{k!} = \sum_{j,k} \frac{r^k z^{k+j} E_j(k, 1)}{k!j!} = \sum_{j,k} \frac{z^k}{k!}\binom{k}{j} E_j(k-j, 1)r^{k-j}$$

$$= \sum_k \frac{z^k}{k!} \sum_j \binom{k-1}{j} k^j r^{k-j} = \sum_k z^k E_k(r, 1).$$

30. Each graph described defines a set $C_x \subseteq \{1, \ldots, n\}$, where j is in C_x if and only if there is a path from t_j to r_i for some $i \leq x$. For a given C_x each graph described is composed of two independent parts: one of the $x(x + \epsilon_1 z_1 + \cdots + \epsilon_n z_n)^{\epsilon_1 + \ldots + \epsilon_n - 1}$ graphs on the vertices r_i, s_{jk}, t_j for $i \leq x$ and $j \in C_x$, where $\epsilon_j = 1$ iff $j \in C_x$, plus one of the $y\big(y + (1 - \epsilon_1)z_1 + \cdots + (1 - \epsilon_n)z_n\big)^{(1-\epsilon_1)+\ldots+(1-\epsilon_n)-1}$ graphs on the remaining vertices.

31. $G(z) = z + G(z)^2 + G(z)^3 + G(z)^4 + \cdots = z + G(z)^2/\big(1 - G(z)\big)$. Hence $G(z) = \frac{1}{4}(1 + z - \sqrt{1 - 6z + z^2})$. [*Note:* It is perhaps remarkable that these numbers for $n > 1$ are just half the values obtained in Ex. 2.2.1–11, i.e. half the number of permutations on n elements obtainable with a restricted deque. Another problem equivalent to this one was posed and solved by Schröder, *Zeitschrift für Mathematik* **15** (1870), 361–376, who determined the number of ways to insert nonoverlapping diagonals in a convex $(n + 1)$-gon. Can the reader find any connection between this problem and Ex. 2.2.1–11, to explain why the answers are essentially the same?]

32. Zero if $n_0 \neq 1 + n_2 + 2n_3 + 3n_4 + \cdots$ (cf. Ex. 2.3–21), otherwise

$$(n_0 + n_1 + \cdots + n_m - 1)!/n_0!n_1! \ldots n_m!\,.$$

To prove this result we recall that an unlabeled tree with $n = n_0 + n_1 + \cdots + n_m$ nodes is characterized by the sequence $d_1\, d_2 \ldots d_n$ of the degrees of the nodes in postorder (Section 2.3.3). Furthermore such a sequence of degrees corresponds to a tree if and only if $\sum_{1 \leq j \leq k} (1 - d_j) > 0$ for $0 < k \leq n$. (This important property of Polish notations is readily proved by induction; cf. Algorithm 2.3.3F with f a function that creates a tree, like the **TREE** function of Section 2.3.2.) In particular, d_1 must be 0. The answer to our problem is therefore the number of sequences $d_2 \ldots d_n$ with n_j occurrences of j for $j > 0$, namely the multinomial coefficient

$$\binom{n-1}{n_0 - 1, n_1, \ldots, n_m},$$

minus the number of such sequences $d_2 \ldots d_n$ for which $\sum_{2 \leq j \leq k} (1 - d_j) < 0$ for some $k \geq 2$.

We may enumerate the latter sequences as follows: Let t be minimal such that $\sum_{2 \leq j \leq t} (1 - d_j) < 0$; then $\sum_{2 \leq j \leq t} (1 - d_j) = -s$ where $1 \leq s < d_t$, and we may form the subsequence $d'_2 \ldots d'_n = d_{t-1} \ldots d_2 0 d_{t+1} \ldots d_n$ which has n_j occurrences

of j for $j \neq d_t$, $n_j - 1$ occurrences of j for $j = d_t$. Now $\sum_{2\le j\le k}(1 - d'_j)$ is equal to d_t when $k = n$, equal to $d_t - s$ when $k = t$, and less than $d_t - s$ when $k < t$. (To prove the latter statement, note that

$$\sum_{2\le j\le k}(1 - d'_j) = \sum_{2\le j<t}(1 - d_j) - \sum_{2\le j\le t-k}(1 - d_j) \le \sum_{2\le j<t}(1 - d_j)$$
$$= d_t - s - 1.)$$

It follows that, given s and any sequence $d'_2 \ldots d'_n$, the construction can be reversed; hence the number of sequences $d_2 \ldots d_n$ which have a given value of d_t and s is the multinomial coefficient

$$\binom{n-1}{n_0, \ldots, n_{d_t} - 1, \ldots, n_m}.$$

The number of sequences $d_2 \ldots d_n$ which correspond to trees is therefore obtained by summing over the possible values of d_t and s:

$$\sum_{0\le j\le m}(1 - j)\binom{n-1}{n_0, \ldots, n_j - 1, \ldots, n_m} = \frac{(n-1)!}{n_0!n_1!\ldots n_m!}\sum_{0\le j\le m}(1 - j)n_j$$

and the latter sum is 1.

An even simpler proof of this result has been given by G. N. Raney (*Transactions of the American Math. Society* **94** (1960), 441–451). If $d_1\, d_2 \ldots d_n$ is any sequence with n_j appearances of j, there is precisely one cyclic rearrangement $d_k \ldots d_n d_1 \ldots d_{k-1}$ that corresponds to a tree, namely the rearrangement where k is maximal such that $\sum_{1\le j\le k}(1 - d_j)$ is minimal.

Either of the methods above can be generalized to show that the number of (ordered, unlabelled) forests having f trees and n_j nodes of degree j is $(n-1)!f/n_0!n_1!\ldots!\,n_m!$, provided the necessary condition $n_0 = f + n_2 + 2n_3 + \cdots$ is satisfied. This result was independently proved by L. Takács, whose paper is cited in the answer to Ex. 2.2.1–4.

33. Consider the number of trees with n_1 nodes labelled 1, n_2 nodes labelled 2, . . . , and such that each node labelled j has degree e_j. Let this number be $c(n_1, n_2, \ldots)$, with the specified degrees $e_1, e_2, \ldots$ regarded as fixed. The generating function $G(z_1, z_2, \ldots) = \sum c(n_1, n_2, \ldots)z_1^{n_1}z_2^{n_2}\ldots$ satisfies the identity $G = z_1G^{e_1} + \cdots + z_rG^{e_r}$, since $z_jG^{e_j}$ enumerates those trees whose root is labelled j. And by the result of the previous exercise,

$$c(n_1, n_2, \ldots) = \begin{cases} \dfrac{(n_1 + n_2 + \cdots - 1)!}{n_1!n_2!\ldots}, & \text{if } (1-e_1)n_1 + (1-e_2)n_2 + \cdots = 1; \\ 0, & \text{otherwise.} \end{cases}$$

More generally, since G^f enumerates the number of ordered forests having such labels, we have for integer $f > 0$

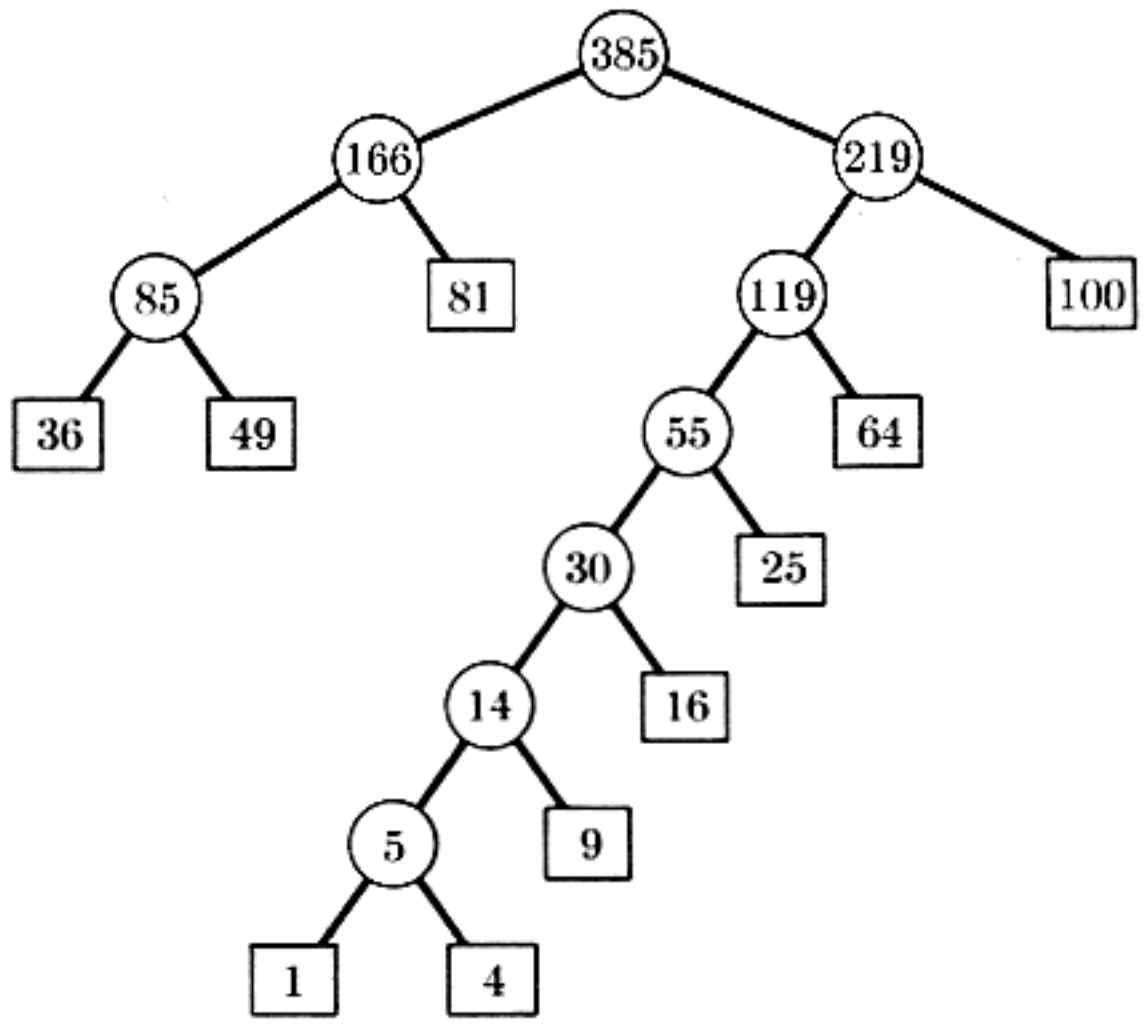
$$w^f = \sum_{f=(1-e_1)n_1+(1-e_2)n_2+\ldots} \frac{(n_1 + n_2 + \cdots - 1)!f}{n_1!n_2!}\, z_1^{n_1}z_2^{n_2}\ldots$$

These formulas are meaningful when $r = \infty$, and they are essentially equivalent to "Lagrange's inversion formula."

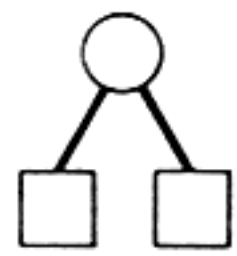
SECTION 2.3.4.5

1. Yes, there are $\binom{8}{5}$ in all, since the nodes numbered 8, 9, 10, 11, 12 may be attached in any of eight positions below 4, 5, 6, and 7.

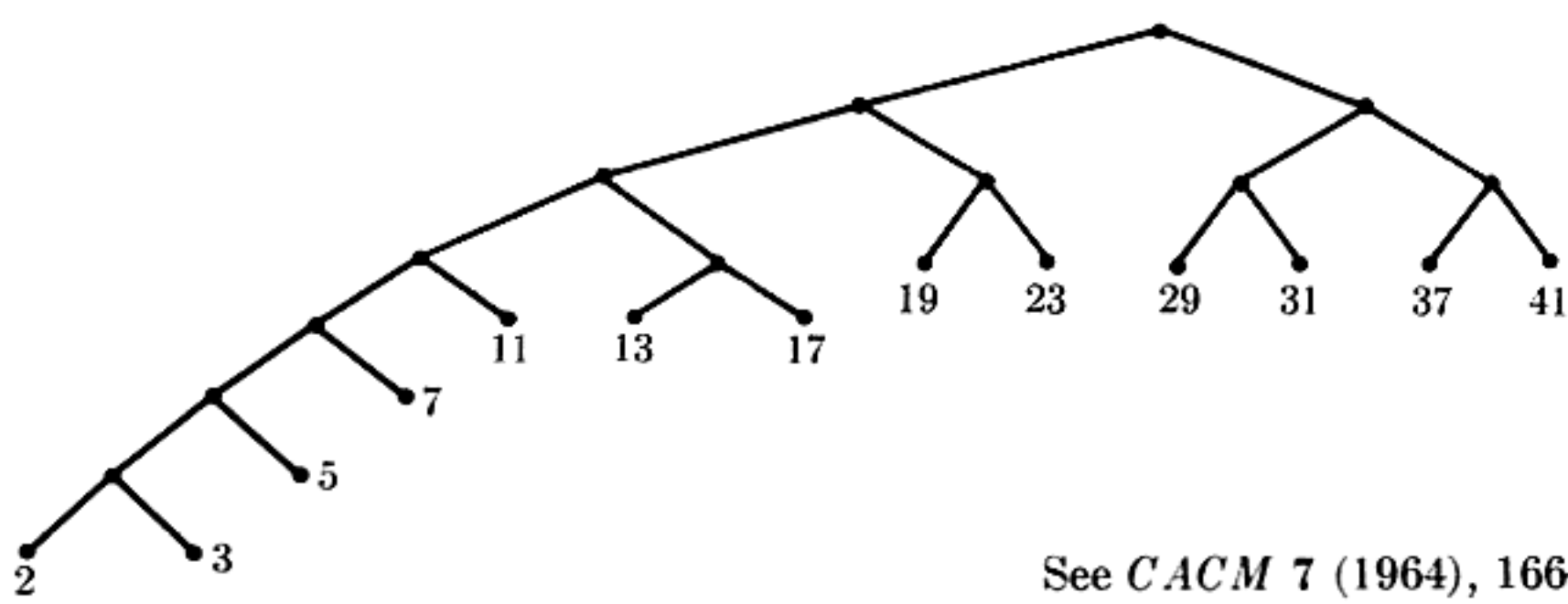
2.



3. By induction on m, the condition is necessary. Conversely if $\sum_{1\le j\le m} 2^{-l_j} = 1$, we want to construct an extended binary tree with these path lengths. If $m = 1$, then $l_1 = 0$ and the construction is trivial. Otherwise we may assume the l's are ordered so that $l_1 = l_2 = \cdots = l_q > l_{q+1} \ge l_{q+2} \ge \cdots \ge l_m > 0$ for some q with $1 \le q \le m$. Now $2^{l_1-1} = \sum_{1\le j\le m} 2^{l_1-l_j-1} = \frac{1}{2}q +$ integer, hence q is even. By induction on m there is a tree with path lengths $l_1 - 1, l_3, l_4, \ldots, l_m$; take such a tree and replace one of the terminals at distance $l_1 - 1$ from the root by



4. First, find a tree by Huffman's method. If $w_j < w_{j+1}$, then $l_j > l_{j+1}$, or else the tree would not be optimal. The construction in the answer to Ex. 3 now gives us another tree with these same path lengths and with the weights in the proper sequence. For example, the tree (11) becomes



See *CACM* **7** (1964), 166–169.

5. (a) $$b_{np} = \sum_{\substack{k+l=n-1 \\ r+s+n-1=p}} b_{kr} b_{ls}.$$

Hence $zB(w, wz)^2 = B(w, z) - 1$. (b) Take the partial derivative with respect to w:

$$2zB(w, wz)\big(B_w(w, wz) + zB_z(w, wz)\big) = B_w(w, z).$$

Therefore if $H(z) = B_w(1, z) = \sum_n h_n z^n$, we find $H(z) = 2zB(z)\big(H(z) + zB'(z)\big)$; and the known formula for $B(z)$ implies

$$H(z) = \frac{1}{1-4z} - \frac{1}{z}\left(\frac{1-z}{\sqrt{1-4z}} - 1\right),$$

so

$$h_n = 4^n - \frac{3n+1}{n+1}\binom{2n}{n}.$$

The average value is h_n/b_n. (c) Asymptotically, this comes to $n\sqrt{\pi n} - 3n + O(\sqrt{n})$.

For the solution to similar problems, see John Riordan, *IBM J. Res. and Devel.* **4** (1960), 473–478; and A. Rényi and G. Szekeres, "On the height of trees," *J. Australian Math. Soc.* **7** (1967), 497–507.

6. $n + s - 1 = tn$.

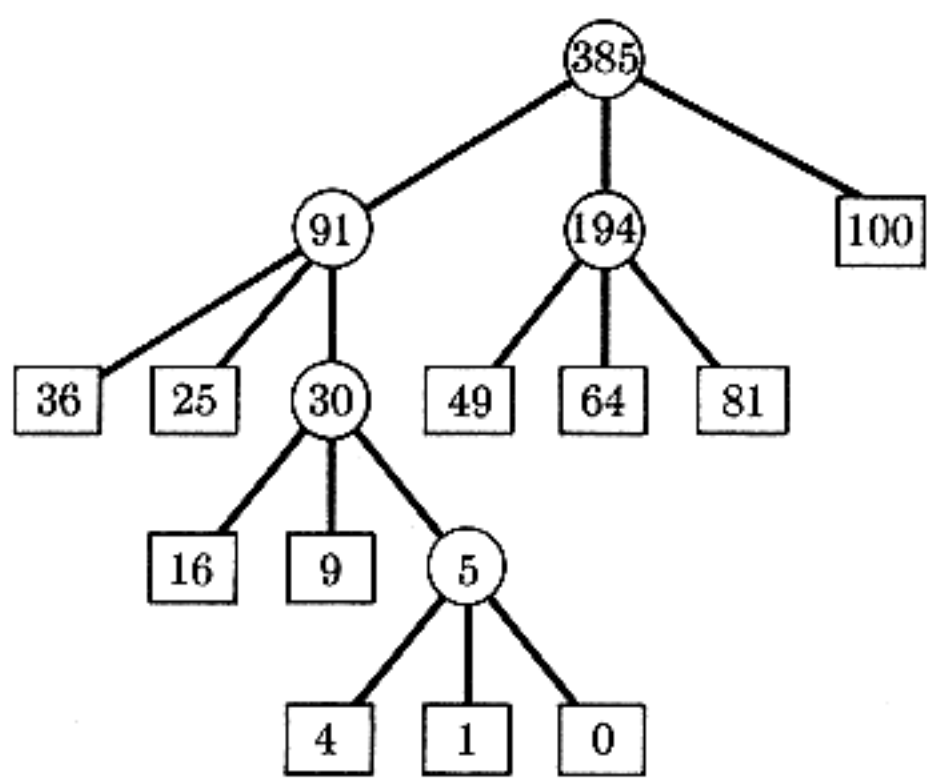
7. $E = (t-1)I + tn$.

8. $$\sum_{1 \le k \le n} \lfloor \log_t ((t-1)k) \rfloor = \text{(summation by parts) } nq - \sum_{\substack{0 \le k \le n \\ \exists j, (t-1)k+1 = t^j}} k.$$

The latter sum may be rewritten $\sum_{1 \le j \le q} (t^j - 1)/(t-1)$.

9. Induction on the size of the tree.

10. By adding extra *zero* weights, if necessary, we may assume that $m \bmod (t-1) = 1$. To obtain a t-ary tree with minimum weighted path length, combine the smallest t values at each step and replace them by their sum. The proof is essentially the same as the binary case. The desired ternary tree is



11. The "Dewey" notation is the binary representation of the node number.

1. A List structure is a directed graph in which the arcs leaving each vertex are ordered, and some of the vertices which have no arcs leaving (out-degree 0) are designated "atoms." Furthermore there is a vertex S such that there is an oriented path from S to V for all vertices $V \neq S$.

2. Not in the same way, since thread links in the usual representation lead back to "FATHER" which is not unique for sub-Lists. However, the representation discussed in Ex. 2.3.4.2–25 can perhaps be used (although this idea has not yet been exploited at the time of writing).

3. If only P0 is to be marked, the algorithm certainly operates correctly. If $n > 1$ nodes are to be marked, then note that ATOM(P0) = 0. Step E4 then sets ALINK(P0) $\leftarrow \Lambda$ and executes the algorithm with P0 replaced by ALINK(P0) and T replaced by P0. By induction (note that since MARK(P0) is now 1, all links to P0 are equivalent to Λ by steps E4 and E5), we see that ultimately we will mark all nodes on paths that start with ALINK(P0) and do not pass through P0; and we will then get to step E6 with T = P0 and P = ALINK(P0). Now since ATOM(T) = 1, step E6 restores ALINK(P0) and ATOM(P0) and we reach step E5. Step E5 sets BLINK(P0) $\leftarrow \Lambda$, etc., and a similar argument shows that we will ultimately mark all nodes on paths that start with BLINK(P0) and do not pass through P0 or nodes reachable from ALINK(P0). Then we will get to E6 with T = P0, P = BLINK(P0), and finally we get to E6 with T = Λ, P = P0.

4. The program which follows incorporates the suggested improvements in the speed of processing atoms which appear in the text after the statement of Algorithm E.

In steps E4 and E5 of the algorithm, we want to test if MARK(Q) = 0. If NODE(Q) = +0, this is an unusual case which can be properly handled by setting it to −0 and treating it as if it were originally −0, since it has ALINK and BLINK both Λ. This simplification is not reflected in the timing calculations below.

rI1 ≡ P, rI2 ≡ T, rI3 ≡ Q, rX ≡ −1 (for setting MARKs)

```
01      MARK    EQU     (0:0)
02      ATOM    EQU     (1:1)
03      ALINK   EQU     (2:3)
04      BLINK   EQU     (4:5)
05      E1      LD1     P0              1       E1. Initialize. P ← P0.
06              ENT2    0               1       T ← Λ.
07              ENTX    -1              1       rX ← −1.
08      E2      STX     0,1(MARK)       1       E2. Mark. MARK(P) ← "−".
09      E3      LDA     0,1(ATOM)       1       E3. Atom?
10              JAZ     E4              1       Jump if ATOM(P) = 0.
11      E6      J2Z     DONE            n       E6. Up.
12              ENT3    0,2           n − 1     Q ← T.
13              LDA     0,3(ATOM)     n − 1
14              JANZ    1F            n − 1     Jump if ATOM(T) = 1.
15              LD2     0,3(BLINK)      t2      T ← BLINK(Q).
16              ST1     0,3(BLINK)      t2      BLINK(Q) ← P.
17              ENT1    0,3             t2      P ← Q.
18              JMP     E6              t2
```

19	1H	STZ	0,2(ATOM)	$t1$	ATOM(T) ← 0.
20		LD2	0,3(ALINK)	$t1$	T ← ALINK(Q).
21		ST1,	0,3(ALINK)	$t1$	ALINK(Q) ← P.
22		ENT1	0,3	$t1$	P ← Q.
23	E5	LD3	0,1(BLINK)	n	*E5. Down* BLINK. Q ← BLINK(P).
24		J3Z	E6	n	Jump if Q = Λ.
25		LDA	0,3	$n - b2$	
26		STX	0,3(MARK)	$n - b2$	MARK(Q) ← "−".
27		JANP	E6	$n - b2$	Jump if NODE(Q) was already marked.
28		LDA	0,3(ATOM)	$t2 + a2$	
29		JANZ	E6	$t2 + a2$	Jump if ATOM(Q) = 1.
30		ST2	0,1(BLINK)	$t2$	BLINK(P) ← T.
31	E4A	ENT2	0,1	$n - 1$	T ← P.
32		ENT1	0,3	$n - 1$	P ← Q.
33	E4	LD3	0,1(ALINK)	n	*E4. Down* ALINK. Q ← ALINK(P).
34		J3Z	E5	n	Jump if Q = Λ.
35		LDA	0,3	$n - b1$	
36		STX	0,3(MARK)	$n - b1$	MARK(Q) ← "−".
37		JANP	E5	$n - b1$	Jump if NODE(Q) was already marked.
38		LDA	0,3(ATOM)	$t1 + a1$	
39		JANZ	E5	$t1 + a1$	Jump if ATOM(Q) = 1.
40		STX	0,1(ATOM)	$t1$	ATOM(P) ← 1.
41		ST2	0,1(ALINK)	$t1$	ALINK(P) ← T.
42		JMP	E4A	$t1$	T ← P, P ← Q, to E4. ▮

By Kirchhoff's law, $t1 + t2 + 1 = n$. The total time is $(34n + 4t1 + 3a - 5b - 8)u$, where n is the number of non-atomic nodes marked, a is the number of atoms marked, b is the number of Λ links encountered in marked non-atomic nodes, and $t1$ is the number of times we went down an ALINK ($0 \le t1 < n$).

5. (The following is the 'fastest known' marking algorithm.)

S1. Set MARK(PO) ← 1. If ATOM(PO) = 1, the algorithm terminates; otherwise set S ← 0, R ← PO, T ← Λ.

S2. Set P ← BLINK(R). If P = Λ or MARK(P) = 1, go to S3. Otherwise set MARK(P) ← 1. Now if ATOM(P) = 1, go to S3; otherwise if S < N set S ← S + 1, STACK[S] ← P, and go to S3; otherwise set F ← 0 and go to S5.

S3. Set P ← ALINK(R). If P = Λ or MARK(P) = 1, go to S4. Otherwise set MARK(P) ← 1. Now if ATOM(P) = 1, go to S4; otherwise if S < N set S ← S + 1, STACK[S] ← P, and go to S4; otherwise set F ← 1 and go to S5.

S4. If S = 0, terminate the algorithm; otherwise set R ← STACK[S], S ← S − 1, and go to S2.

S5. Set Q ← ALINK(P). If Q = Λ or MARK(Q) = 1, go to S6. Otherwise set MARK(Q) ← 1. Now if ATOM(Q) = 1, go to S6; otherwise set ATOM(P) ← 1, ALINK(P) ← T, T ← P, P ← Q, go to S5.

S6. Set Q ← BLINK(P). If Q = Λ or MARK(Q) = 1, go to S7. Otherwise set MARK(Q) ← 1. Now if ATOM(Q) = 1, go to S7; otherwise set BLINK(P) ← T, T ← P, P ← Q, go to S5.

S7. If T = Λ, go to S3 if F = 0, to S4 if F = 1. Otherwise set Q ← T. If ATOM(Q) = 1, set ATOM(Q) ← 0, T ← ALINK(Q), ALINK(Q) ← P, P ← Q, and return to S6. If ATOM(Q) = 0, set T ← BLINK(Q), BLINK(Q) ← P, P ← Q, and return to S7. ▮

Reference: *CACM* **10** (1967), 501–506.

6. From the second phase of garbage collection (or perhaps also the initial phase if all mark bits are set to zero at this time).

7. Delete steps E2 and E3, and delete "ATOM(P) ← 1" in E4. Set MARK(P) ← 1 in step E5 and use "MARK(Q) = 0", "MARK(Q) = 1" in step E6 in place of the present "ATOM(Q) = 1", "ATOM(Q) = 0" respectively. The idea is to set the MARK bit only after the left subtree has been marked. *This algorithm works even if the tree has overlapping (shared) subtrees,* but it does not work for all recursive List structures such as those with NODE(ALINK(Q)) an ancestor of NODE(Q). (Note that ALINK of a marked node is never changed.)

8. Solution 1: Analogous to Algorithm E, but simpler.

F1. Set T ← Λ, P ← PO.

F2. Set MARK(P) ← 1, and set P ← P + SIZE(P).

F3. If MARK(P) = 1, go to F5.

F4. Set Q ← LINK(P). If Q ≠ Λ and MARK(Q) = 0, set LINK(P) ← T, T ← P, P ← Q, and go to F2.

F5. If T = Λ, stop. Otherwise set Q ← T, T ← LINK(Q), LINK(Q) ← P, P ← Q − 1, and return to F3. ▮

Solution 2: Analogous to Algorithm D. For this solution, we assume the SIZE field is large enough to contain a link address. Such an assumption is probably not justified by the statement of the problem, but it lets us use a slightly faster method than the first solution when it is applicable.

G1. Set T ← Λ, MARK(PO) ← 1, P ← PO + SIZE(PO).

G2. If MARK(P) = 1, go to G5.

G3. Set Q ← LINK(P), P ← P − 1.

G4. If Q ≠ Λ and MARK(Q) = 0, set MARK(Q) ← 1, S ← SIZE(Q), SIZE(Q) ← T, T ← Q + S. Go back to G2.

G5. If T = Λ, stop. Otherwise set P ← T and find the first value of Q = P, P − 1, P − 2, . . . for which MARK(Q) = 1; set T ← SIZE(Q) and SIZE(Q) ← P − Q. Go back to G2. ▮

9. H1. Set L ← 0, K ← M + 1, MARK(0) ← 1, MARK(M + 1) ← 0.

H2. Increase L by one, and if MARK(L) = 1 repeat this step.

H3. Decrease K by one, and if MARK(K) = 0 repeat this step.

H4. If L > K, go to step H5; otherwise set NODE(L) ← NODE(K), ALINK(K) ← L, MARK(K) ← 0, and return to H2.

H5. For L = 1, 2, . . . , K do the following: Set MARK(L) ← 0.
If ATOM(L) = 0 and ALINK(L) > K, set ALINK(L) ← ALINK(ALINK(L)).
If ATOM(L) = 0 and BLINK(L) > K, set BLINK(L) ← ALINK(BLINK(L)). ▮

10. **Z1.** [Initialize.] Set F ← PO, R ⇐ AVAIL, NODE(R) ← NODE(F), REF(F) ← R. (Here F and R are pointers for a queue set up in the REF fields of all header nodes encountered.)

Z2. [Begin new List.] Set P ← F, Q ← REF(P).

Z3. [Advance to right.] Set P ← RLINK(P). If P = Λ, go to Z6.

Z4. [Copy one node.] Set Q1 ⇐ AVAIL, RLINK(Q) ← Q1, Q ← Q1, NODE(Q) ← NODE(P).

Z5. [Translate sub-List link.] If T(P) = 1, set P1 ← REF(P), and if REF(P1) = Λ set REF(R) ← P1, R ⇐ AVAIL, REF(P1) ← R, NODE(R) ← NODE(P1), REF(Q) ← R. If T(P) = 1 and REF(P1) ≠ Λ, set REF(Q) ← REF(P1). Go to Z3.

Z6. [Move to next List.] Set RLINK(Q) ← Λ. If REF(F) ≠ R, set F ← REF(REF(F)) and return to Z2. Otherwise set REF(R) ← Λ, P ← PO.

Z7. [Final cleanup.] Set Q ← REF(P). If Q ≠ Λ, set REF(P) ← Λ and P ← Q and repeat step Z7. ∎

11. Here is a pencil-and-paper method which can be written out more formally to answer the problem: First attach a unique name (e.g. a capital letter) to each List in the given set; in the example we would have for example $A = (a{:}C, b, a{:}F)$, $F = (b{:}D)$, $B = (a{:}F, b, a{:}E)$, $C = (b{:}G)$, $G = (a{:}C)$, $D = (a{:}F)$, $E = (b{:}G)$. Now make a list of pairs of List names that must be proved equal. Successively add pairs to this list until either a contradiction is found because we have a pair which disagree on the first level (then the originally given Lists are unequal), or until the list of pairs does not imply any further pairs (then the originally given Lists are equal). In the example, this list of pairs would originally contain only the given pair, AB; then it gets the further pairs CF, EF (by matching A and B), DG (from CF) and then we have a self-consistent set.

To prove the validity of this method, observe that (a) if it returns the answer "unequal", the given Lists are unequal; (b) if the given Lists are unequal, it returns the answer "unequal"; (c) it always terminates.

12. When the AVAIL list contains N nodes, where N is a specified constant to be chosen as discussed below, initiate another coroutine which shares computer time with the main routine and does the following: (a) Marks all N nodes on the AVAIL list; (b) Marks all other nodes which are accessible to the program; (c) links all unmarked nodes together to prepare a new AVAIL list for use when the current AVAIL list is empty, and (d) resets the mark bits in all nodes. One must choose N and the ratio of time sharing so there is a positive guarantee that operations (a), (b), (c), and (d) are complete before N nodes are taken from the AVAIL list, yet the main routine is running sufficiently fast. It is necessary to use some care in step (b) to make sure all nodes "accessible to the program" are included, as the program continues to run; details are omitted here. If the list formed in (c) has less than N nodes, it may be necessary to stop eventually because memory space might become exhausted.

SECTION 2.4

1. Preorder.
2. Essentially proportional to the number of Data Table entries created.

3. Change step A5 to: "A5′. [Remove top level.] Remove the top stack entry; and if the new level number at the top of the stack is $\geq L$, let (L1, P1) be the new entry at the top of the stack and repeat this step. Otherwise set BROTHER(P1) $\leftarrow$ Q and then let (L1, P1) be the new entry at the top of the stack."

4. Rule (c) is violated if and only if there is a data item whose *complete qualification* A_0 OF ... OF A_n is also a COBOL reference to some other data item. Therefore Algorithm A would be extended to check, as each new data item is added to the Data Table, whether its complete qualification is contained in some previous reference of the same name, or whether the complete qualification of some previous item of the same name is also a reference to this new data item. This means essentially that Algorithm B is appended to Algorithm A in the appropriate ways.

5. Make the following changes:

Step	*replace*	*by*
B1.	P $\leftarrow$ LINK(P_0)	P $\leftarrow$ LINK(INFO(T))
B2.	$k \leftarrow 0$	K $\leftarrow$ T
B3.	$k < n$	RLINK(K) $\neq \Lambda$
B4.	$k \leftarrow k + 1$	K $\leftarrow$ RLINK(K)
B6.	NAME(S) = P_k	NAME(S) = INFO(K)

6. A simple modification of Algorithm B makes it search only for complete references (if $k = n$ and FATHER(S) $\neq \Lambda$ in step B3, or if NAME(S) $\neq P_k$ in step B6, set P $\leftarrow$ PREV(P) and go to B2). The idea is to run through this modified Algorithm B first; then, if Q is *still* Λ, perform the unmodified algorithm.

7. MOVE MONTH OF DATE OF SALES TO MONTH OF DATE OF PURCHASES. MOVE DAY OF DATE OF SALES TO DAY OF DATE OF PURCHASES. MOVE YEAR OF DATE OF SALES TO YEAR OF DATE OF PURCHASES. MOVE ITEM OF TRANSACTION OF SALES TO ITEM OF TRANSACTION OF PURCHASES. MOVE QUANTITY OF TRANSACTION OF SALES TO QUANTITY OF TRANSACTION OF PURCHASES. MOVE PRICE OF TRANSACTION OF SALES TO PRICE OF TRANSACTION OF PURCHASES. MOVE TAX OF TRANSACTION OF SALES TO TAX OF TRANSACTION OF PURCHASES.

8. If and only if α or β is an elementary item. (It may be of interest to note that the author failed to handle this case properly in his first draft of Algorithm C, and it actually made the algorithm more complicated.)

9. "MOVE CORRESPONDING α TO β", if neither α nor β is elementary, is equivalent to the set of statements "MOVE CORRESPONDING A OF α TO A OF β" taken over all names A common to groups α and β. (This is a more elegant way to state the definition than the more traditional and more cumbersome definition of "MOVE CORRESPONDING" given in the text.) We may verify that Algorithm C satisfies this definition, using an inductive proof that steps C2 through C5 will ultimately terminate with P = P0 and Q = Q0. Further details of the proof are filled in as we have done many times before in a "tree induction" (cf. the proof of Algorithm 2.3.1T).

10. (a) Set S1 $\leftarrow$ LINK(P_k). Then repeatedly set S1 $\leftarrow$ PREV(S1) zero or more times until either S1 = Λ (NAME(S) $\neq P_k$) or S1 = S (NAME(S) = P_k). (b) Set P1 $\leftarrow$ P and then set P1 $\leftarrow$ PREV(P1) zero or more times until PREV(P1) = Λ; do a similar operation with variables Q1, Q; and then test if P1 = Q1. Alternatively if the Data

Table entries are ordered so that PREV(P) < P for all P, a faster test can be made in an obvious way depending on whether P > Q or not and following the PREV links of the larger to see if the smaller is encountered.

11. A miniscule improvement in the speed of step C4 would be achieved by adding a new link field BROTHER1(P) ≡ SON(FATHER(P)). More significantly, we could modify the SON and BROTHER links so that NAME(BROTHER(P)) > NAME(P); this would speed up the search in step C3 considerably because it would require only one pass over each family to find the matching members. This would therefore remove the only "search" present in Algorithms B or C. Algorithms A and C are readily modified for this interpretation, and the reader may find this an interesting exercise. (However, if we consider the relative frequency of MOVE CORRESPONDING statements and the usual size of family groups, the resulting speedup will not be terribly significant in the translation of actual COBOL programs.)

12. Leave steps B1, B2, B3 unchanged; change the other steps thus:

B4. Set $k \leftarrow k + 1$, R ← LINK(P_k).

B5. If R = Λ, there is no match; set P ← PREV(P) and go to B2. If R ≤ S ≤ SCOPE(R), go to B3. Otherwise set R ← PREV(R) and repeat step B5. ▮

This algorithm does *not* adapt to the PL/I convention of Ex. 6.

13. Use the same algorithm, minus the operations that set NAME, FATHER, SON, and BROTHER. Whenever removing the top stack entry in step A5, set SCOPE(L1) ← Q − 1. When the input is exhausted in step A2, simply set L ← 0 and continue, then terminate the algorithm if L = 0 in step A7.

14. The following algorithm, using an auxiliary stack (cf. Chapter 8), has steps numbered to show a direct correspondence with the text's algorithm.

C1. Set P ← P0, Q ← Q0, and set the stack contents empty.

C2. If SCOPE(P) = P or SCOPE(Q) = Q, output (P, Q) as one of the desired pairs and go to C5. Otherwise put (P, Q) on the stack and set P ← P + 1, Q ← Q + 1.

C3. Determine if P and Q point to entries with the same name (cf. Ex. 10(b)). If so, go to C2. If not, let (P1, Q1) be the entry at the top of the stack; if SCOPE(Q) < SCOPE(Q1), set Q ← SCOPE(Q) + 1 and repeat step C3.

C4. Let (P1, Q1) be the entry at the top of the stack. If SCOPE(P) < SCOPE(P1), set P ← SCOPE(P) + 1, Q ← Q1 + 1, and go back to C3. If SCOPE(P) = SCOPE(P1), set P ← P1, Q ← Q1 and remove the top entry of the stack.

C5. If the stack is empty, the algorithm terminates. Otherwise go to C4. ▮

SECTION 2.5

1. In such fortuitous circumstances, a stack-like operation may be used as follows: Let the memory pool area be locations 0 through M — 1, and let AVAIL point to the lowest free location. To reserve N words, report failure if AVAIL + N $\geq$ M, otherwise set AVAIL $\leftarrow$ AVAIL + N. To free these N words, just set AVAIL $\leftarrow$ AVAIL — N.

2. The amount of storage space for an item of length l is $k\lceil l/(k - b)\rceil$, which has the average value $kL/(k - b) + (1 - \alpha)k$, where α is assumed to be $\frac{1}{2}$, independent

of k. This expression is a minimum (for real values of k) when $k = b + \sqrt{2bL}$. So choose k to be the integer just above or just below this value, whichever gives the lowest value of $kL/(k - b) + \frac{1}{2}k$. For example if $b = 1$ and $L = 10$, $k \approx 1 + \sqrt{20} = 5$ or 6; both are equally good. For much greater detail about this problem, see *JACM* **12** (1965), 53–70.

4. rI1 ≡ Q, rI2 ≡ P.

```
A1    LDA    N              rA ← N.
      ENT2   AVAIL          P ← LOC(AVAIL).
A2A   ENT1   0,2            Q ← P.
A2    LD2    0,1(LINK)      P ← LINK(Q).
      J2N    OVERFLOW       If P = Λ, no room.
A3    CMPA   0,2(SIZE)
      JG     A2A            Jump if N > SIZE(P).
A4    SUB    0,2(SIZE)      rA ← N − SIZE(P) ≡ K.
      JANZ   *+3            Jump if K ≠ 0.
      LDX    0,2(LINK)      LINK(P)
      STX    0,1(LINK)      → LINK(Q).
      STA    0,2(SIZE)      SIZE(P) ← K.
      LD1    0,2(SIZE)      Optional ending,
      INC1   0,2            sets rI1 ← P + K. ▮
```

5. Probably not. The unavailable storage area just before location P will subsequently become available, and its length will be increased by the amount K; an increase of 99 would not be negligible.

6. The idea is to try to search in different parts of the AVAIL list each time. We can use a "roving pointer," called ROVER for example, which is treated as follows: In step A1, set Q ← ROVER. After step A4, set ROVER ← LINK(Q). In step A2, when P = Λ the first time during a particular execution of Algorithm A, set Q ← LOC(AVAIL) and repeat step A2. When P = Λ the *second* time, the algorithm terminates unsuccessfully. In this way ROVER will tend to point to a random spot in the AVAIL list, and the sizes will be more balanced. At the beginning of the program, set ROVER ← Λ; it is *also* necessary to set ROVER to Λ everywhere else in the program where the block whose address equals the current setting of ROVER is taken out of the AVAIL list.

7. 2000, 1000 with requests of sizes 800, 1200.

8. In step A1, also set R ← Λ. In step A2, if P = Λ go to A6. In step A3, go to A5 not A4. Add new steps as follows:

A5. [Better fit?] If R = Λ or M > SIZE(P), set R ← Q and M ← SIZE(P). Then set Q ← P and return to A2.

A6. [Any found?] If R = Λ, the algorithm terminates unsuccessfully. Otherwise set Q ← R, P ← LINK(Q), and go to A4. ▮

9. Obviously if we are so lucky as to find SIZE(P) = N, we have a "best fit" and it is not necessary to search farther. (When there are only very few different block sizes, this occurs rather often.) If a "boundary tag" method like in Algorithm C is being used, it is possible to maintain the AVAIL list in sorted order, so the length of search could be cut down to $\frac{1}{2}$ the length of the list or less, on the average; more generally, we could in this case break the AVAIL list into several separate lists for

different size ranges, or even make it into a pseudo-balanced tree structure as described in Chapter 6.

10. Make the following changes:

Step B2, for "P > PO" read "P ≥ PO".

Step B3, insert "If PO + N > P (and P ≠ Λ), set P ← LINK(P) and repeat step B3."

Step B4, for "Q + SIZE(Q) = PO", read "Q + SIZE(Q) ≥ PO"; and for "SIZE (Q) ← SIZE(Q) + N" read "SIZE(Q) ← PO + N − Q".

11. If ROVER ≠ Λ and PO > ROVER, we can set Q ← ROVER instead of Q ← LOC(AVAIL) in step B1. If there are n entries in the AVAIL list, the average number of iterations of step B2 is $(2n+3)(n+2)/6(n+1) = \frac{1}{3}n + \frac{5}{6} + O(\frac{1}{n})$. For example if $n = 2$ we get 9 equally probable situations:

	PO < AVAIL	AVAIL < PO < LINK(AVAIL)	LINK(AVAIL) < PO
ROVER=AVAIL	1 //	1	2
ROVER=LINK(AVAIL)	1	2 //	1
ROVER=Λ	1	2	3

The average is $\frac{1}{9}(\binom{2}{2} + \binom{3}{2} + \binom{4}{2} + \binom{3}{2} + \binom{2}{2}) = \frac{1}{9}(\binom{5}{3} + \binom{4}{3}) = \frac{14}{9}$.

12. A1. Set P ← ROVER, F ← 0.

A2. If P = LOC(AVAIL) and F = 0, set P ← AVAIL, F ← 1, and repeat step A2.
If P = LOC(AVAIL) and F ≠ 0, the algorithm terminates unsuccessfully.

A3. If SIZE(P) ≥ N, go to A4; otherwise set P ← LINK(P) and return to A2.

A4. Set ROVER ← LINK(P), K ← SIZE(P) − N. If K < c (where c is a constant which must equal 2 or more), set LINK(LINK(P + 1)) ← ROVER, LINK(ROVER + 1) ← LINK(P + 1), L ← P; otherwise set L ← P + K, SIZE(P) ← SIZE(L − 1) ← K, TAG(L − 1) ← "−", SIZE(L) ← N. Set TAG(L) ← TAG(L + SIZE(L) − 1) ← "+". ∎

13. rI1 ≡ P, rX ≡ F, rI2 ≡ L.

```
LINK    EQU   4:5
SIZE    EQU   1:2
TSIZE   EQU   0:2
TAG     EQU   0:0
A1      LDA   N           rA ← N.
        SLA   3           Shift into SIZE field.
        ENTX  0           F ← 0.
        LD1   ROVER       P ← ROVER.
        JMP   A2
A3      CMPA  0,1(SIZE)
        JLE   A4          Jump if N ≤ SIZE(P).
        LD1   0,1(LINK)   P ← LINK(P).
A2      ENT2  -AVAIL,1    rI2 ← P − LOC(AVAIL).
        J2NZ  A3
        JXNZ  OVERFLOW    Is F ≠ 0?
        ENTX  1           Set F ← 1.
```

```
         LD1    AVAIL(LINK)   P ← AVAIL.
         JMP    A2
A4       LD2    0,1(LINK)
         ST2    ROVER         ROVER ← LINK(P).
         LDA    0,1(SIZE      rA ≡ K ← SIZE(P) — N.
         SUB    N
         CMPA   =C=
         JGE    1F            Jump if K ≥ c.
         LD3    1,1(LINK)     rI3 ← LINK(P + 1).
         ST2    0,3(LINK)     LINK(rI3) ← ROVER.
         ST3    1,2(LINK)     LINK(ROVER + 1) ← rI3.
         ENT2   0,1           L ← P.
         LD3    0,1(SIZE)     rI3 ← SIZE(P).
         JMP    2F
1H       STA    0,1(SIZE)     SIZE(P) ← K.
         LD2    0,1(SIZE)
         INC2   0,1           L ← P + K.
         LDAN   0,1(SIZE)     rA ← −K.
         STA    -1,2(TSIZE)   SIZE(L − 1) ← K, TAG(L − 1) ← "−".
         LD3    N             rI3 ← N.
2H       ST3    0,2(TSIZE)    TAG(L) ← "+", also set SIZE(L) ← rI3.
         INC3   0,2
         STZ    -1,3(TAG)     TAG(L + SIZE(L) − 1) ← "+". ∎
```

14. (a) This field is needed to locate the beginning of the block, in step C2. It could be replaced (perhaps to advantage) by a link to the first word of the block. (b) This field is needed because it is necessary to reserve more than N words at times (for example if $K = 1$), and the amount reserved must be present when the block is subsequently freed.

15, 16. rI1 ≡ P0, rI2 ≡ P1, rI3 ≡ F, rI4 ≡ B, rI6 ≡ −N.

```
D1   LD1    P0            D1.
     LD2    0,1(SIZE)
     ENN6   0,2           N ← SIZE(P0).
     INC2   0,1           P1 ← P0 + N.
     LD5    0,2(TSIZE)
     J5N    D4            To D4 if TAG(P1) = "−".
D2   LD5    -1,1(TSIZE)   D2.
     J5N    D7            To D7 if TAG(P0 − 1) = "−".
D3   LD3    AVAIL(LINK)   D3. Set F ← AVAIL.
     ENT4   AVAIL         B ← LOC(AVAIL).
     JMP    D5            To D5.
D4   INC6   0,5           D4. N ← N + SIZE(P1).
     LD3    0,2(LINK)     F ← LINK(P1).
     LD4    1,2(LINK)     B ← LINK(P1 + 1).
     CMP2   ROVER         (This part because of the ROVER
     JNE    *+3           feature of Ex. 12:
     ENTX   AVAIL         If P1 = ROVER,
     STX    ROVER               set ROVER ← LOC(AVAIL).)
```

```
        DEC2   0,5              P1 ← P1 + SIZE(P1).
        LD5    -1,1(TSIZE)
        J5N    D6               To D6 if TAG(P0 − 1) = "−".
D5      ST3    0,1(LINK)        D5. LINK(P0) ← F.
        ST4    1,1(LINK)        LINK (P0 + 1) ← B.
        ST1    1,3(LINK)        LINK (F + 1) ← P0.
        ST1    0,4(LINK)        LINK(B) ← P0.
        JMP    D8               To D8.
D6      ST3    0,4(LINK)        D6. LINK(B) ← F.
        ST4    1,3(LINK)        LINK(F + 1) ← B.
D7      INC6   0,5              D7. N ← N + SIZE(P0 − 1).
        INC1   0,5              P0 ← P0 − SIZE(P0 − 1).
D8      ST6    0,1(TSIZE)       D8. SIZE(P0) ← N, TAG(P0) ← "−".
        ST6    -1,2(TSIZE)      SIZE(P1 − 1) ← N, TAG (P1− 1) ← "−". ∎
```

17. Both LINK fields equal to LOC(AVAIL).

18. Algorithm A reserves the upper end of a large block. When storage is completely available, the first fit method actually begins by reserving the high-order locations, but once these become available again they are not re-reserved since a "fit" is usually found already in the lower locations; thus the initial large block at the lower end of memory quickly disappears with "first fit." A large block rarely is the "best fit," however, so the best fit method leaves a large block at the beginning of memory.

19. Use the algorithm of Ex. 12, except delete the references to SIZE (L− 1), TAG(L − 1), and TAG(L + SIZE(L) − 1) from step A4; also insert the following actions at the beginning of step A3: "Set P1 ← P + SIZE(P). If TAG(P1) = "−", set LINK(LINK(P1) + 1) ← LINK(P1 + 1), LINK(LINK(P1 + 1)) ← LINK(P1), SIZE(P) ← SIZE(P) + SIZE(P1), and repeat step A3. Otherwise:"

Clearly the situation of (2), (3), (4) can't occur here; the only real effect on the storage allocation is that the search here will tend to be longer than in Ex. 12, and sometimes K will be less than c although there is really another available block preceding this one that we do not know about.

(An alternative is to take the collapsing out of the inner loop A3, and to do the collapsing only in step A4 before the final allocation or in the inner loop when the algorithm would otherwise have terminated unsuccessfully. This alternative requires a simulation study to see if it is an improvement or not.)

20. When a buddy is found to be available, during the collapsing loop, we want to remove that block from its AVAIL[k] list, but we do not know which links to update unless (a) we do a possibly long search, or (b) the list is doubly-linked.

21. If $n = 2^k\alpha$, where $1 \le \alpha \le 2$, a_n is $2^{2k+1}(\alpha - \frac{2}{3}) + \frac{1}{3}$, and b_n is $2^{2k-1}\alpha^2 + 2^{k-1}\alpha$. The ratio a_n/b_n for large n is essentially $4(\alpha - \frac{2}{3})/\alpha^2$, which takes its minimum value $\frac{4}{3}$ when $\alpha = 1$ and 2, and its maximum value $\frac{3}{2}$ when $\alpha = 1\frac{1}{3}$. So a_n/b_n approaches no limit, it oscillates between these two extremes.

22. This idea requires a TAG field in several words of the 11-word block, not only in the first word. It is a workable idea, provided these extra TAG bits can be spared, and it would appear to be especially suitable for use in computer hardware.

23. 011011110100; 011011100000.

24. This introduces a bug in the program; we may get to step S1 when TAG(0) = 1, since S2 may return to S1. To make it work, add "TAG(L) ← 0" after "L ← P" in step S2. (It is easier to assume instead that TAG(2^m) = 0.)

25. The idea is quite correct. (Note that criticism need not be negative.) The list heads AVAIL[k] may be eliminated for $n < k \leq m$; the algorithms of the text may be used if "m" is changed to "n" in steps R1, S1. The initial conditions (13), (14) should be changed to indicate 2^{m-n} blocks of size 2^n instead of one block of size 2^m.

26. Using the binary representation of M, we can easily modify the initial conditions (13), (14) so that all memory locations are divided into blocks whose size is a power of two. In Algorithm S, TAG(P) should be regarded as 0 whenever P ≥ M.

27. rI1 ≡ k, rI2 ≡ j, rI3 ≡ $j - k$, rI4 ≡ L, LOC(AVAIL[j]) = AVAIL + j; assume that there is an auxiliary table TWO[j] = 2^j, stored in location TWO + j, for $0 \leq j \leq m$. Assume further that TAG = +, − represents TAG = 0, 1; TAG(LOC(AVAIL[j])) = "−", except, as a sentinel, TAG(LOC(AVAIL[$m+1$])) = "+".

```
00    KVAL    EQU   5:5
01    TAG     EQU   0:0
02    LINKF   EQU   1:2
03    LINKB   EQU   3:4
04    TLNKF   EQU   0:2
05    R1      LD1   K                  1     R1. Find block.
06            ENT2  0,1                1     j ← k.
07            ENT3  0                  1
08            LD4   AVAIL,2(LINKF)     1
09    1H      ENT5  AVAIL,2            1+R
10            DEC5  0,4                1+R
11            J5NZ  R2                 1+R   Jump if AVAILF[j] ≠ LOC(AVAIL[j]).
12            INC2  1                  R     Increase j
13            INC3  1                  R
14            LD4N  AVAIL,2(TLNKF)     R
15            J4NN  1B                 R     Is j ≤ m?
16            JMP   OVERFLOW
17    R2      LD5   0,4(LINKF)         1     R2. Remove from list.
18            ST5   AVAIL,2(LINKF)     1     AVAILF[j] ← LINKF(L).
19            ENTA  AVAIL,2            1
20            STA   0,5(LINKB)         1     LINKB(L) ← LOC(AVAIL[j]).
21            STZ   0,4(TAG)           1     TAG(L) ← 0.
22    R3      J3Z   DONE               1     R3. Split required?
23    R4      DEC3  1                  R     R4. Split.
24            DEC2  1                  R     Decrease j.
25            LD5   TWO,2              R     rI5 ≡ P
26            INC5  0,4                R     P ← L + 2^j.
27            ENNA  AVAIL,2            R
28            STA   0,5(TLNKF)         R     TAG(P) ← 1, LINKF(P) ← LOC(AVAIL[j]).
29            STA   0,5(LINKB)         R     LINKB(P) ← LOC(AVAIL[j]).
30            ST5   AVAIL,2(LINKF)     R     AVAILF[j] ← P.
31            ST5   AVAIL,2(LINKB)     R     AVAILB[j] ← P.
32            ST2   0,5(KVAL)          R     KVAL(P) ← j.
33            J3P   R4                 R     Go to R3.
34    DONE    ...                            ∎
```

28. $r\mathrm{I}1 \equiv k$, $r\mathrm{I}5 \equiv \mathrm{P}$, $r\mathrm{I}4 \equiv \mathrm{L}$; assume $\mathtt{TAG}\ (2^m) =$ "+".

```
01  S1  LD4   L                1      S1. Is buddy available?
02      LD1   K                1
03  1H  ENTA  0,4              1+S
04      XOR   TWO,1            1+S    rA ← buddy_k(L).
05      STA   TEMP             1+S
06      LD5   TEMP             1+S    P ← rA.
07      LDA   0,5              1+S
08      JANN  S3               1+S    Jump if TAG(P) = 0.
09      CMP1  0,5(KVAL)        B+S
10      JNE   S3               B+S    Jump if KVAL(P) ≠ k.
11  S2  LD2   0,5(LINKF)         S    S2. Combine with buddy.
12      LD3   0,5(LINKB)         S
13      ST3   0,2(LINKF)         S    LINKF(LINKB(P)) ← LINKF(P).
14      ST2   0,3(LINKB)         S    LINKB(LINKF(P)) ← LINKB(P).
15      INC1  1                  S    Increase k.
16      CMP4  TEMP               S
17      JL    1B                 S
18      ENT4  0,5                A    If L > P, set L ← P.
19      JMP   1B                 A
20  S3  LD2   AVAIL,1(LINKF)     1    S3. Put on list.
21      ENNA  AVAIL,1            1
22      STA   0,4(0:4)           1    TAG(L) ← 1,LINKB(L) ← LOC(AVAIL[k]).
23      ST2   0,4(LINKF)         1    LINKF(L) ← AVAILF[k].
24      ST1   0,4(KVAL)          1    KVAL(L) ← k.
25      ST4   0,2(LINKB)         1    LINKB(AVAILF[k]) ← L.
26      ST4   AVAIL,1(LINKF)     1    AVAILF[k] ← L.   ∎
```

29. Yes, but only at the expense of some searching, or (better) an additional table of TAG bits packed somehow. (It is tempting to suggest that buddies not be joined together during Algorithm S, but only in Algorithm R if there is no block large enough to meet the request; but this probably leads to a badly fragmented memory.)

33. G1. [Clear LINKs.] Set $\mathtt{P} \leftarrow 1$, and repeat the operation $\mathtt{LINK(P)} \leftarrow \Lambda$, $\mathtt{P} \leftarrow \mathtt{P} + \mathtt{SIZE(P)}$ until $\mathtt{P} = \mathtt{AVAIL}$. (This merely sets the LINK field in the first word of each node to Λ; we may assume in most cases that this step is unnecessary, since LINK(P) is set to Λ in step G9 below and it can be set to Λ by the storage allocator.)

G2. [Initialize marking phase.] Set $\mathtt{TOP} \leftarrow \mathtt{USE}$, $\mathtt{LINK(TOP)} \leftarrow \mathtt{AVAIL}$, $\mathtt{LINK(AVAIL)} \leftarrow \Lambda$. (TOP points to the top of a stack as in Algorithm 2.3.5D.)

G3. [Pop up stack.] Set $\mathtt{P} \leftarrow \mathtt{TOP}$, $\mathtt{TOP} \leftarrow \mathtt{LINK(TOP)}$. If $\mathtt{TOP} = \Lambda$, go to G5.

G4. [Put new links on stack.] For $1 \le k \le \mathtt{T(P)}$, do the following operations: Set $\mathtt{Q} \leftarrow \mathtt{LINK(P} + k\mathtt{)}$, and if $\mathtt{Q} \neq \Lambda$, $\mathtt{LINK(Q)} = \Lambda$ set $\mathtt{LINK(Q)} \leftarrow \mathtt{TOP}$, $\mathtt{TOP} \leftarrow \mathtt{Q}$.

G5. [Initialize next phase.] (Now $\mathtt{P} = \mathtt{AVAIL}$, and the marking phase has been completed so that the first word of each accessible node has a nonnull LINK. Now we wish to combine adjacent inaccessible nodes, for speed in later

steps, and to assign new addresses to the accessible ones.) Set Q ← 1, LINK(AVAIL) ← Q, SIZE(AVAIL) ← 0, P ← 1. (Location AVAIL is being used as a sentinel to signify the end of a loop in subsequent phases.)

G6. [Assign new addresses.] If LINK(P) = Λ, go to G7. Otherwise if SIZE(P) = 0, go to G8. Otherwise set LINK(P) ← Q, Q ← Q + SIZE(P), P ← P + SIZE(P), and repeat this step.

G7. [Collapse available areas.] If LINK(P + SIZE(P)) = Λ, increase SIZE(P) by SIZE(P + SIZE(P)) and repeat this step. Otherwise set P ← P + SIZE(P) and return to G6.

G8. [Translate all links.] (Now the LINK field in the first word of each accessible node contains the address to which the node will be moved.) Set USE ← LINK(USE), and AVAIL ← Q. Then set P ← 1, and repeat the following operation until SIZE(P) = 0: If LINK(P) ≠ Λ, set LINK(Q) ← LINK(LINK(Q)) for P < Q ≤ P + T(P); then regardless of the value of LINK(P), set P ← P + SIZE(P).

G9. [Move.] Set P ← 1, and repeat the following operation until SIZE(P) = 0: Set Q ← LINK(P), and if Q ≠ Λ set LINK(P) ← Λ and NODE(Q) ← NODE(P); then whether Q = Λ or not, set P ← P + SIZE(P). (The operation NODE(Q) ← NODE(P) implies the movement of SIZE(P) words; we always have Q ≤ P, so it is safe to move the words in order from smallest location to largest.)

[This method is called the "LISP 2 garbage collector." Another, somewhat more complicated compacting algorithm has been described by B. K. Haddon and W. M. Waite, *Comp. J.* **10** (1967), 162–165.]

34. Let TOP ≡ rI1, Q ≡ rI2, P ≡ rI3, k ≡ rI4, SIZE(P) ≡ rI5. Assume further that Λ = 0, and LINK(0) ≠ 0 to simplify step G4. Step G1 is omitted.

```
01    LINK  EQU   4:5
02    INFO  EQU   0:3
03    SIZE  EQU   1:2
04    T     EQU   3:3
05    G2    LD1   USE          1       G2. Initialize marking phase. TOP ← USE.
06          LD2   AVAIL        1
07          ST2   0,1(LINK)    1       LINK (TOP) ← AVAIL.
08          STZ   0,2(LINK)    1       LINK(AVAIL) ← Λ.
09    G3    ENT3  0,1          a+1     G3. Pop up stack. P ← TOP.
10          LD1   0,1(LINK)    a+1     TOP ← LINK(TOP).
11          J1Z   G5           a+1     To G5 if TOP = Λ.
12    G4    LD4   0,3(T)       a       G4. Put new links on stack. k ← T(P).
13    1H    J4Z   G3           b+a     k = 0?
14          INC3  1            b       P ← P + 1.
15          DEC4  1            b       k ← k − 1.
16          LD2   0,3(LINK)    b       Q ← LINK(P).
17          LDA   0,2(LINK)    b
18          JANZ  1B           b       Jump if LINK(Q) ≠ Λ.
19          ST1   0,2(LINK)    a−1     Otherwise set LINK(Q) ← TOP,
20          ENT1  0,2          a−1           TOP ← Q.
21          JMP   1B           a−1
22    G5    ENT2  1            1       G5. Initialize next phase. Q ← 1.
23          ST2   0,3          1       LINK(AVAIL) ← 1, SIZE(AVAIL) ← 0.
```

```
24          ENT3  1                1          P ← 1.
25          JMP   G6               1
26    1H    ST2   0,3(LINK)        a          LINK(P) ← Q.
27          INC2  0,5              a          Q ← Q + SIZE(P).
28          INC3  0,5              a          P ← P + SIZE(P).
29    G6    LDA   0,3(LINK)        a + 1      G6. Assign new addresses.
30    G6A   LD5   0,3(SIZE)        a + c + 1
31          JAZ   G7               a + c + 1  Jump if LINK(P) = Λ.
32          J5NZ  1B               a + 1      Jump if SIZE(P) ≠ 0.
33    G8    LD1   USE              1          G8. Translate all links.
34          LDA   0,1(LINK)        1
35          STA   USE              1          USE ← LINK(USE).
36          ST2   AVAIL            1          AVAIL ← Q.
37          ENT3  1                1          P ← 1.
38          JMP   G8P              1
39    1H    LD6   0,6(SIZE)        d
40          INC5  0,6              d          rI5 ← rI5 + SIZE(P + SIZE(P)).
41    G7    ENT6  0,3              c + d      G7. Collapse available areas.
42          INC6  0,5              c + d      rI6 ← P + SIZE(P).
43          LDA   0,6(LINK)        c + d
44          JAZ   1B               c + d      Jump if LINK(rI6) = Λ.
45          ST5   0,3(SIZE)        c          SIZE(P) ← rI5.
46          INC3  0,5              c          P ← P + SIZE(P).
47          JMP   G6A              c
48    2H    DEC4  1                b          k ← k − 1.
49          INC2  1                b          Q ← Q + 1.
50          LD6   0,2(LINK)        b
51          LDA   0,6(LINK)        b
52          STA   0,2(LINK)        b          LINK(Q) ← LINK(LINK(Q)).
53    1H    J4NZ  2B               a + b      Jump if k ≠ 0.
54    3H    INC3  0,5              a + c      P ← P + SIZE(P).
55    G8P   LDA   0,3(LINK)        1 + a + c
56          LD5   0,3(SIZE)        1 + a + c
57          JAZ   3B               1 + a + c  Is LINK(P) = Λ?
58          LD4   0,3(T)           1 + a      k ← T(P).
59          ENT2  0,3              1 + a      Q ← P.
60          J5NZ  1B               1 + a      Jump unless SIZE(P) = 0.
61    G9    ENT3  1                1          G9. Move. P ← 1.
62          ENT1  1                1          Set rI1 for MOVE instructions.
63          JMP   G9P              1
64    1H    STZ   0,3(LINK)        a          LINK(P) ← Λ.
65          ST5   *+1(4:4)         a
66          MOVE  0,3(*)           a          NODE(rI1) ← NODE(P), rI1 ← rI1 + SIZE(P).
67    3H    INC3  0,5              a + c      P ← P + SIZE(P).
68    G9P   LDA   0,3(LINK)        1 + a + c
69          LD5   0,3(SIZE)        1 + a + c
70          JAZ   3B               1 + a + c  Jump if LINK(P) = Λ.
71          J5NZ  1B               1 + a      Jump unless SIZE(P) = 0. ∎
```

Note that in line 66 we are assuming that the size of each node is sufficiently small that it can be moved with a single MOVE instruction; this seems a fair assumption for most cases when this kind of garbage collection is applicable.

The total running time for this program is $(44a + 17b + 2w + 25c + 8d + 47)u$ where a is the number of accessible nodes, b is the number of link fields therein, c is the number of inaccessible nodes which are *not* preceded by an inaccessible node, d is the number of inaccessible nodes which *are* preceded by an inaccessible node, and w is the total number of words in the accessible nodes. If the memory contains n nodes, with ρn of these inaccessible, then we may estimate $a = (1 - \rho)n$, $c = (1 - \rho)\rho n$, $d = \rho^2 n$. Example: five-word nodes (on the average), with two link fields per node (on the average), and a memory of 1000 nodes. Then when $\rho = \frac{1}{5}$, it takes $374u$ per available node recovered; when $\rho = \frac{1}{2}$, it takes $104u$; and when $\rho = \frac{4}{5}$, it takes only $33u$.

37. First request $n + 2$ 1-cell blocks. If the middle cell a_{n+1} is unavailable, liberate all others and request a block of size $n + 1$. If a_{n+1} is available, find the maximum j and minimum k such that a_j and a_k are unavailable, $1 \leq j \leq n$, $n + 2 \leq k \leq 2n + 1$. Then $k - j \leq n$, so if we liberate all but a_j and a_k we cannot fill a request for n cells in a row.

Note: Considerably more is true; let $N(M, m)$ denote the minimum number of storage cells such that requests can always be filled when at most M cells are requested at a time and no blocks of size $> m$ are requested. Then either the first fit method or the buddy system can be used to show that $N(M, m) = O(M \log (m + 1))$; conversely, lower bounds can be established to show that $N(M, m)$ is asymptotically $cM \log m$ for large m and M. [These results are due to R. L. Graham and J. M. Robson (to be published).]

INDEX TO NOTATIONS

In the following formulas, letters which are not further qualified have the following significance:

j, k	integer-valued arithmetic expression
m, n	nonnegative integer-valued arithmetic expression
x, y, z	real-valued arithmetic expression
f	real-valued function
P	pointer-valued expression, i.e., either Λ or an address within a computer.

Formal symbolism	Meaning	Section reference
NODE(P)	the node (group of variables which are individually distinguished by their field names) whose address is P, P $\neq \Lambda$	2.1
F(P)	the variable in NODE(P) whose field name is F	2.1
CONTENTS(P)	contents of computer "word" whose address is P	2.1
LOC(V)	address of variable V within a computer	2.1
A_n	the nth element of linear array A	
A_{mn}	the element in row m, column n of rectangular array A	
$A[n]$	equivalent to A_n	1.1
$A[m, n]$	equivalent to A_{mn}	1.1
$V \leftarrow E$	give variable V the value of expression E	1.1
$U \leftrightarrow V$	interchange the values of variables U and V	1.1
P $\Leftarrow$ AVAIL	set the value of pointer variable P to the address of a new node, or signal memory overflow if there is no room for a new node	2.2.3

Formal symbolism	Meaning	Section reference
AVAIL ⇐ P	NODE(P) is returned to free storage; all its fields lose their identity	2.2.3
top(S)	node at the top of a nonempty stack S	2.2.1
$X \Leftarrow S$	pop up S to X: set $X \leftarrow$ top(S); delete top(S) from nonempty stack S	2.2.1
$S \Leftarrow X$	push down X onto S: insert the value or group of values denoted by X as a new entry on the top of stack S	2.2.1
front(Q)	node at the front of a nonempty queue Q	2.2.1
$X \Leftarrow Q$	set $X \leftarrow$ front(Q); delete front(Q) from nonempty queue Q	2.2.1
$Q \Leftarrow X$	insert the value or group of values denoted by X as a new entry at the rear of queue Q	2.2.1
$(B \Rightarrow E_1; E_2)$	conditional expression: denotes E_1 if B is true, E_2 if B is false	8.1
δ_{jk}	Kronecker delta: $(j = k \Rightarrow 1;\ 0)$	1.2.6
$\sum_{R(k)} f(k)$	sum of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$\prod_{R(k)} f(k)$	product of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$\min_{R(k)} f(k)$	minimum value of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$\max_{R(k)} f(k)$	maximum value of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$j \backslash k$	j divides k: $k \bmod j = 0$	1.2.4
gcd(j, k)	greatest common divisor of j and k: $(j = k = 0 \Rightarrow 0;\quad \max_{d \backslash j, d \backslash k} d)$	1.1
det (A)	determinant of square matrix A	1.2.3
A^T	transpose of rectangular array A: $A^T[j, k] = A[k, j]$	1.2.3
x^y	x to the y power, x positive	1.2.2
x^k	x to the kth power: $\left(k \geq 0 \Rightarrow \prod_{0 \leq j < k} x,\quad 1/x^{-k}\right)$	1.2.2

Formal symbolism	Meaning	Section reference
$x^{\overline{k}}$	x upper k:	
	$\left(k \geq 0 \Rightarrow x(x+1)\cdots(x+k-1) = \prod_{0 \leq j < k} (x+j);\quad 1/(x+k)^{\overline{-k}}\right)$	1.2.6
$x^{\underline{k}}$	x lower k:	
	$\left(k \geq 0 \Rightarrow x(x-1)\cdots(x-k+1) = \prod_{0 \leq j < k} (x-j);\right.$	
	$\left. 1/(x-k)^{\underline{-k}}\right) = (-1)^k(-x)^{\overline{k}}$	1.2.6
$n!$	n factorial: $1 \cdot 2 \cdot \cdots \cdot n = n^{\underline{n}}$	1.2.5
$\binom{x}{k}$	binomial coefficient: $(k < 0 \Rightarrow 0;\ x^{\underline{k}}/k!)$	1.2.6
$\binom{n}{n_1, n_2, \ldots, n_m}$	multinomial coefficient, $n = n_1 + n_2 + \cdots + n_m$	1.2.6
$\begin{bmatrix} n \\ m \end{bmatrix}$	Stirling number of first kind:	
	$\sum_{0<k_1<k_2<\cdots<k_{n-m}<n} k_1 k_2 \cdots k_{n-m}$	1.2.6
$\begin{Bmatrix} n \\ m \end{Bmatrix}$	Stirling number of second kind:	
	$\sum_{1\leq k_1\leq k_2\leq \cdots \leq k_{n-m}\leq m} k_1 k_2 \cdots k_{n-m}$	1.2.6
$\{a \mid R(a)\}$	set of all a for which the relation $R(a)$ is true	
$\{a_1, \ldots, a_n\}$	the set $\{a_k \mid 1 \leq k \leq n\}$	
$\{x\}$	in contexts where a real value, not a set, is required, denotes fractional part: $x \bmod 1$	1.2.11.2
$\lvert x \rvert$	absolute value of x: $(x < 0 \Rightarrow -x;\ x)$	
$\lfloor x \rfloor$	floor of x, greatest integer function: $\max_{k \leq x} k$	1.2.4
$\lceil x \rceil$	ceiling of x, least integer function: $\min_{k \geq x} k$	1.2.4
$x \bmod y$	mod function: $(y = 0 \Rightarrow x;\ x - y\lfloor x/y \rfloor)$	1.2.4
$x \equiv y \pmod{z}$	relation of congruence: $x \bmod z = y \bmod z$	1.2.4

Formal symbolism	Meaning	Section reference
$\log_b x$	logarithm, base b, of x (real positive $b \neq 1$): $x = b^{\log_b x}$	1.2.2
$\ln x$	natural logarithm: $\log_e x$	1.2.2
$\exp x$	exponential of x: e^x	1.2.2
$\langle X_n \rangle$	the infinite sequence $X_0, X_1, X_2, \ldots$ (here n is a letter which is part of the symbol)	1.2.9
$f'(x)$	derivative of f at x	1.2.9
$f''(x)$	second derivative of f at x	1.2.10
$f^{(n)}(x)$	nth derivative: $(n = 0 \Rightarrow f(x);$ $g'(x) \quad$ where $\quad g(x) = f^{(n-1)}(x))$	1.2.11.2
$H_n^{(x)}$	$1 + 1/2^x + \cdots + 1/n^x = \sum_{1 \leq k \leq n} 1/k^x$	1.2.7
H_n	harmonic number: $H_n^{(1)}$	1.2.7
F_n	Fibonacci number: $(n \leq 1 \Rightarrow n; F_{n-1} + F_{n-2})$	1.2.8
B_n	Bernoulli number	1.2.11.2
$B(x, y)$	Beta function	1.2.6
sign (x)	sign of x: $(x = 0 \Rightarrow 0; (x > 0 \Rightarrow +1; -1))$	
$\zeta(x)$	zeta function: $H_\infty^{(x)}$ when $x > 1$	1.2.7
$\Gamma(x)$	gamma function: $\gamma(x, \infty)$; $(x - 1)!$ when x is a positive integer	1.2.5
$\gamma(x, y)$	incomplete gamma function	1.2.11.3
γ	Euler's constant	1.2.7
e	base of natural logarithms: $\sum_{k \geq 0} 1/k!$	1.2.2
∞	infinity: larger than any number	
Λ	null link (pointer to no address)	2.1
$\emptyset$	empty set (set with no elements)	
ϕ	golden ratio, $\frac{1}{2}(1 + \sqrt{5})$	1.2.8
$\varphi(n)$	Euler's totient function: $\sum_{\substack{0 \leq k < n \\ \gcd(k,n)=1}} 1$	1.2.4
$p(n)$	number of partitions of n	1.2.1
$O(f(n))$	big-oh of $f(n)$ as $n \to \infty$	1.2.11.1

Formal symbolism	Meaning	Section reference
$O(f(x))$	big-oh of $f(x)$, for small x (or for x in some specified range)	1.2.11.1
(min x_1, ave x_2, max x_3, dev x_4)	a random variable having minimum value x_1, average ("expected") value x_2, maximum value x_3, standard deviation x_4	1.2.10
mean (g)	mean value of probability distribution represented by generating function g: $g'(1)$	1.2.10
var (g)	variance of probability distribution represented by generating function g: $g''(1) + g'(1) - g'(1)^2$	1.2.10
P*	address of preorder successor of NODE(P) in a tree or binary tree	2.3.1, 2.3.2
P$	address of postorder successor of NODE(P) in a tree or binary tree	2.3.1, 2.3.2
P#	address of endorder successor of NODE(P) in a tree or binary tree	2.3.1, 2.3.2
*P	address of preorder predecessor of NODE(P) in a tree or binary tree	2.3.1, 2.3.2
$P	address of postorder predecessor of NODE(P) in a tree or binary tree	2.3.1, 2.3.2
#P	address of endorder predecessor of NODE(P) in a tree or binary tree	2.3.1, 2.3.2
∎	end of algorithm, program, or proof	1.1
␣	one blank space	1.3.1
rA	register A (accumulator) of MIX	1.3.1
rX	register X (extension) of MIX	1.3.1
rI1, . . . , rI6	(index) registers I1, . . . , I6 of MIX	1.3.1
rJ	(jump) register J of MIX	1.3.1
(L:R)	partial field of MIX word, $0 \le L \le R \le 5$	1.3.1
OP ADDRESS,I(F)	notation for MIX instruction	1.3.1, 1.3.2
u	unit of time in MIX	1.3.1
*	"self" in MIXAL	1.3.2
0F, 1F, 2F, . . . , 9F	"forward" local symbol in MIXAL	1.3.2
0B, 1B, 2B, . . . , 9B	"backward" local symbol in MIXAL	1.3.2
0H, 1H, 2H, . . . , 9H	"here" local symbol in MIXAL	1.3.2

APPENDIX B

TABLES OF NUMERICAL QUANTITIES

Table 1

Quantities which are frequently used in standard subroutines and in analysis of computer programs. (40 decimal places)

$$\sqrt{2} = 1.41421\ 35623\ 73095\ 04880\ 16887\ 24209\ 69807\ 85697-$$
$$\sqrt{3} = 1.73205\ 08075\ 68877\ 29352\ 74463\ 41505\ 87236\ 69428+$$
$$\sqrt{5} = 2.23606\ 79774\ 99789\ 69640\ 91736\ 68731\ 27623\ 54406+$$
$$\sqrt{10} = 3.16227\ 76601\ 68379\ 33199\ 88935\ 44432\ 71853\ 37196-$$
$$\sqrt[3]{2} = 1.25992\ 10498\ 94873\ 16476\ 72106\ 07278\ 22835\ 05703-$$
$$\sqrt[3]{3} = 1.44224\ 95703\ 07408\ 38232\ 16383\ 10780\ 10958\ 83919-$$
$$\sqrt[4]{2} = 1.18920\ 71150\ 02721\ 06671\ 74999\ 70560\ 47591\ 52930-$$
$$\ln 2 = 0.69314\ 71805\ 59945\ 30941\ 72321\ 21458\ 17656\ 80755+$$
$$\ln 3 = 1.09861\ 22886\ 68109\ 69139\ 52452\ 36922\ 52570\ 46475-$$
$$\ln 10 = 2.30258\ 50929\ 94045\ 68401\ 79914\ 54684\ 36420\ 76011+$$
$$1/\ln 2 = 1.44269\ 50408\ 88963\ 40735\ 99246\ 81001\ 89213\ 74266+$$
$$1/\ln 10 = 0.43429\ 44819\ 03251\ 82765\ 11289\ 18916\ 60508\ 22944-$$
$$\pi = 3.14159\ 26535\ 89793\ 23846\ 26433\ 83279\ 50288\ 41972-$$
$$1^\circ = \pi/180 = 0.01745\ 32925\ 19943\ 29576\ 92369\ 07684\ 88612\ 71344+$$
$$1/\pi = 0.31830\ 98861\ 83790\ 67153\ 77675\ 26745\ 02872\ 40689+$$
$$\pi^2 = 9.86960\ 44010\ 89358\ 61883\ 44909\ 99876\ 15113\ 53137-$$
$$\sqrt{\pi} = \Gamma(1/2) = 1.77245\ 38509\ 05516\ 02729\ 81674\ 83341\ 14518\ 27975+$$
$$\Gamma(1/3) = 2.67893\ 85347\ 07747\ 63365\ 56929\ 40974\ 67764\ 41287-$$
$$\Gamma(2/3) = 1.35411\ 79394\ 26400\ 41694\ 52880\ 28154\ 51378\ 55193+$$
$$e = 2.71828\ 18284\ 59045\ 23536\ 02874\ 71352\ 66249\ 77572+$$
$$1/e = 0.36787\ 94411\ 71442\ 32159\ 55237\ 70161\ 46086\ 74458+$$
$$e^2 = 7.38905\ 60989\ 30650\ 22723\ 04274\ 60575\ 00781\ 31803+$$
$$\gamma = 0.57721\ 56649\ 01532\ 86060\ 65120\ 90082\ 40243\ 10422-$$
$$\ln \pi = 1.14472\ 98858\ 49400\ 17414\ 34273\ 51353\ 05871\ 16473-$$
$$\phi = 1.61803\ 39887\ 49894\ 84820\ 45868\ 34365\ 63811\ 77203+$$
$$e^\gamma = 1.78107\ 24179\ 90197\ 98523\ 65041\ 03107\ 17954\ 91696+$$
$$e^{\pi/4} = 2.19328\ 00507\ 38015\ 45655\ 97696\ 59278\ 73822\ 34616+$$
$$\sin 1 = 0.84147\ 09848\ 07896\ 50665\ 25023\ 21630\ 29899\ 96226-$$
$$\cos 1 = 0.54030\ 23058\ 68139\ 71740\ 09366\ 07442\ 97660\ 37323+$$
$$\zeta(3) = 1.20205\ 69031\ 59594\ 28539\ 97381\ 61511\ 44999\ 07650-$$
$$\ln \phi = 0.48121\ 18250\ 59603\ 44749\ 77589\ 13424\ 36842\ 31352-$$
$$1/\ln \phi = 2.07808\ 69212\ 35027\ 53760\ 13226\ 06117\ 79576\ 77422-$$

Table 2

Quantities which are frequently used in standard subroutines and in analysis of computer programs, in *octal* notation. The *name* of each quantity, appearing at the left of the equal sign, is given in decimal notation.

$0.1 =$	*0.06314 63146 31463 14631 46314 63146 31463 14631 4632*
$0.01 =$	*0.00507 53412 17270 24365 60507 53412 17270 24365 6051*
$0.001 =$	*0.00040 61115 64570 65176 76355 44264 16254 02030 4467*
$0.0001 =$	*0.00003 21556 13530 70414 54512 75170 33021 15002 3522*
$0.00001 =$	*0.00000 24761 32610 70664 36041 06077 17401 56063 3442*
$0.000001 =$	*0.00000 02061 57364 05536 66151 55323 07746 44470 2603*
$0.0000001 =$	*0.00000 00153 27745 15274 53644 12741 72312 20354 0215*
$0.00000001 =$	*0.00000 00012 57143 56106 04303 47374 77341 01512 6333*
$0.000000001 =$	*0.00000 00001 04560 27640 46655 12262 71426 40124 2174*
$0.0000000001 =$	*0.00000 00000 06676 33766 35367 55653 37265 34642 0163*
$\sqrt{2} =$	*1.32404 74631 77167 46220 42627 66115 46725 12575 1744*
$\sqrt{3} =$	*1.56663 65641 30231 25163 54453 50265 60361 34073 4222*
$\sqrt{5} =$	*2.17067 36334 57722 47602 57471 63003 00563 55620 3202*
$\sqrt{10} =$	*3.12305 40726 64555 22444 02242 57101 41466 33775 2253*
$\sqrt[3]{2} =$	*1.20505 05746 15345 05342 10756 65334 25574 22415 0303*
$\sqrt[3]{3} =$	*1.34233 50444 22175 73134 67363 76133 05334 31147 6012*
$\sqrt[4]{2} =$	*1.14067 74050 61556 12455 72152 64430 60271 02755 7314*
$\ln 2 =$	*0.54271 02775 75071 73632 57117 07316 30007 71366 5364*
$\ln 3 =$	*1.06237 24752 55006 05227 32440 63065 25012 35574 5534*
$\ln 10 =$	*2.23273 06735 52524 25405 56512 66542 56026 46050 5071*
$1/\ln 2 =$	*1.34252 16624 53405 77027 35750 37766 40644 35175 0435*
$1/\ln 10 =$	*0.33626 75425 11562 41614 52325 33525 27655 14756 0622*
$\pi =$	*3.11037 55242 10264 30215 14230 63050 56006 70163 21122* 0111G
$1^\circ = \pi/180 =$	*0.01073 72152 11224 72344 25603 54276 63351 22056 1154*
$1/\pi =$	*0.24276 30155 62344 20251 23760 47257 50765 15156 7007*
$\pi^2 =$	*11.67517 14467 62135 71322 25561 15466 30021 40654 3410*
$\sqrt{\pi} = \Gamma(1/2) =$	*1.61337 61106 64736 65247 47035 40510 15273 34470 1776*
$\Gamma(1/3) =$	*2.53347 35234 51013 61316 73106 47644 54653 00106 6605*
$\Gamma(2/3) =$	*1.26523 57112 14154 74312 54572 37655 60126 23231 0245*
$e =$	*2.55760 52130 50535 51246 52773 42542 00471 72363 6166*
$1/e =$	*0.27426 53066 13167 46761 52726 75436 02440 52371 0336*
$e^2 =$	*7.30714 45615 23355 33460 63507 35040 32664 25356 5022*
$\gamma =$	*0.44742 14770 67666 06172 23215 74376 01002 51313 2552*
$\ln \pi =$	*1.11206 40443 47503 36413 65374 52661 52410 37511 4606*
$\phi =$	*1.47433 57156 27751 23701 27634 71401 40271 66710 1501*
$e^\gamma =$	*1.61772 13452 61152 65761 22477 36553 53327 17554 2126*
$e^{\pi/4} =$	*2.14275 31512 16162 52370 35530 11342 53525 44307 0217*
$\sin 1 =$	*0.65665 24436 04414 73402 03067 23644 11612 07474 1451*
$\cos 1 =$	*0.42450 50037 32406 42711 07022 14666 27320 70675 1232*
$\zeta(3) =$	*1.14735 00023 60014 20470 15613 42561 31715 10177 0662*
$\ln \phi =$	*0.36630 26256 61213 01145 13700 41004 52264 30700 4065*
$1/\ln \phi =$	*2.04776 60111 17144 41512 11436 16575 00355 43630 4065*

Tables 1 and 2 contain several hitherto unpublished 40-digit values which have been furnished to the author by Dr. John W. Wrench, Jr.

For high-precision values of constants not found in this list, see J. Peters, *Ten Place Logarithms of the Numbers from 1 to 100000*, Appendix to Volume 1 (New York: F. Ungar Publ. Co., 1957); and *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun (Washington, D.C.: U. S. Govt. Printing Office, 1964), Chapter 1.

Table 3

Values of harmonic numbers, Bernoulli numbers, and Fibonacci numbers for small values of n.

n	H_n	B_n	F_n	n
0	0	1	0	0
1	1	−1/2	1	1
2	3/2	1/6	1	2
3	11/6	0	2	3
4	25/12	−1/30	3	4
5	137/60	0	5	5
6	49/20	1/42	8	6
7	363/140	0	13	7
8	761/280	−1/30	21	8
9	7129/2520	0	34	9
10	7381/2520	5/66	55	10
11	83711/27720	0	89	11
12	86021/27720	−691/2730	144	12
13	1145993/360360	0	233	13
14	1171733/360360	7/6	377	14
15	1195757/360360	0	610	15
16	2436559/720720	−3617/510	987	16
17	42142223/12252240	0	1597	17
18	14274301/4084080	43867/798	2584	18
19	275295799/77597520	0	4181	19
20	55835135/15519504	−174611/330	6765	20
21	18858053/5173168	0	10946	21
22	19093197/5173168	854513/138	17711	22
23	444316699/118982864	0	28657	23
24	1347822955/356948592	−236364091/2730	46368	24
25	34052522467/8923714800	0	75025	25

For any x, let $H_x = \sum_{n\ge 1}\left(\frac{1}{n} - \frac{1}{n+x}\right)$. Then

$$H_{1/2} = 2 - 2\ln 2,$$
$$H_{1/3} = 3 - \tfrac{1}{2}\pi/\sqrt{3} - \tfrac{3}{2}\ln 3,$$
$$H_{2/3} = \tfrac{3}{2} + \tfrac{1}{2}\pi/\sqrt{3} - \tfrac{3}{2}\ln 3,$$
$$H_{1/4} = 4 - \tfrac{1}{2}\pi - 3\ln 2,$$
$$H_{3/4} = \tfrac{4}{3} + \tfrac{1}{2}\pi - 3\ln 2,$$
$$H_{1/5} = 5 - \tfrac{1}{2}\pi\phi\sqrt{\frac{2+\phi}{5}} - \tfrac{1}{2}(3-\phi)\ln 5 - (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{2/5} = \tfrac{5}{2} - \tfrac{1}{2}\pi/\phi\sqrt{2+\phi} - \tfrac{1}{2}(2+\phi)\ln 5 + (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{3/5} = \tfrac{5}{3} + \tfrac{1}{2}\pi/\phi\sqrt{2+\phi} - \tfrac{1}{2}(2+\phi)\ln 5 + (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{4/5} = \tfrac{5}{4} + \tfrac{1}{2}\pi\phi\sqrt{\frac{2+\phi}{5}} - \tfrac{1}{2}(3-\phi)\ln 5 - (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{1/6} = 6 - \tfrac{1}{2}\pi\sqrt{3} - 2\ln 2 - \tfrac{3}{2}\ln 3,$$
$$H_{5/6} = \tfrac{6}{5} + \tfrac{1}{2}\pi\sqrt{3} - 2\ln 2 - \tfrac{3}{2}\ln 3,$$

and, in general, when $0 < p < q$ (cf. exercise 1.2.9–19),

$$H_{p/q} = \frac{q}{p} - \tfrac{1}{2}\pi\cot\frac{p}{q}\pi - \ln 2q + 2\sum_{1\le n<q/2}\cos\frac{2\pi np}{q}\ln\sin\frac{n}{q}\pi.$$

INDEX AND GLOSSARY

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information; an answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.

Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer. For additional definitions of computer terminology, see the *IFIP-ICC Vocabulary of Information Processing* (Amsterdam: North Holland Publishing Co., 1966).

THE ART OF COMPUTER PROGRAMMING

SECOND EDITION

DONALD E. KNUTH *Stanford University*

ADDISON-WESLEY PUBLISHING COMPANY

Volume 2 / **Seminumerical Algorithms**

THE ART OF COMPUTER PROGRAMMING

SECOND EDITION

Reading, Massachusetts
Menlo Park, California · London · Amsterdam · Don Mills, Ontario · Sydney

This book is in the

ADDISON-WESLEY SERIES IN
COMPUTER SCIENCE AND INFORMATION PROCESSING

MICHAEL A. HARRISON, Consulting Editor

Library of Congress Cataloging in Publication Data

Knuth, Donald Ervin (1938–
The Art of Computer Programming. 2d ed.

(Addison-Wesley series in computer science and information processing)
Includes index.
Contents: v. 1. Fundamental algorithms.—v. 2. Seminumerical algorithms.
1. Electronic digital computers—Programming. I. Title.
QA76.6.K64 001.6'42 73-1830
ISBN 0-201-03822-6 (v. 2)

ISBN 0-201-03822-6
BCDEFGHIJ-MA-898765432

PREFACE

O dear Ophelia!
I am ill at these numbers:
I have not art to reckon my groans.

—Hamlet (Act II, Sc. 2, Line 120)

THE ALGORITHMS discussed in this book deal directly with numbers; yet I believe they are properly called *seminumerical*, because they lie on the borderline between numeric and symbolic calculation. Each algorithm not only computes the desired answers to a problem, it also is intended to blend well with the internal operations of a digital computer. In many cases a person will not be able to appreciate the beauty of such an algorithm unless he or she also has some knowledge of a computer's machine language; the efficiency of the corresponding machine program is a vital factor that cannot be divorced from the algorithm itself. The problem is to find the best ways to make computers deal with numbers, and this involves tactical as well as numerical considerations. Therefore the subject matter of this book is unmistakably a part of computer science, as well as of numerical mathematics.

Some people working in "higher levels" of numerical analysis will regard the topics treated here as the domain of system programmers. Other people working in "higher levels" of system programming will regard the topics treated here as the domain of numerical analysts. But I hope that there are a few people left who will want to look carefully at these basic methods; although the methods reside perhaps on a low level, they underlie all of the more grandiose applications of computers to numerical problems, so it is important to know them well. We are concerned here with the interface between numerical mathematics and computer programming, and it is the mating of both types of skills that makes the subject so interesting.

There is a noticeably higher percentage of mathematical material in this book than in other volumes of this series, because of the nature of the subjects treated. In most cases the necessary mathematical topics are developed here starting almost from scratch (or from results proved in Volume 1), but in some easily recognizable sections a knowledge of calculus has been assumed.

This volume comprises Chapters 3 and 4 of the complete series. Chapter 3 is concerned with "random numbers": it is not only a study of various methods for generating random sequences, it also investigates statistical tests for randomness, as well as the transformation of uniform random numbers into other types of random quantities; the latter subject illustrates how random numbers are used in practice. I have also included a section about the nature of randomness itself. Chapter 4 is my attempt to tell the fascinating story of what mankind has been able to learn about the processes of arithmetic, after centuries of progress. It discusses various systems for representing numbers, and how to convert between them; and it treats arithmetic on floating point numbers, high-precision integers, rational fractions, polynomials, and power series, including the questions of factoring and finding greatest common divisors.

Each of Chapters 3 and 4 can be used as the basis of a one-semester college course at the junior to graduate level. Although courses on "Random Numbers" and on "Arithmetic" are not presently a part of many college curricula, I believe the reader will find that the subject matter of these chapters lends itself nicely to a unified treatment of material that has real educational value. My own experience has been that these courses are a good means of introducing elementary probability theory and number theory to college students; nearly all of the topics usually treated in such introductory courses arise naturally in connection with applications, and the presence of these applications can be an important motivation that helps the student to learn and to appreciate the theory. Furthermore, each chapter gives a few hints of more advanced topics that will whet the appetite of many students for further mathematical study.

For the most part this book is self-contained, except for occasional discussions relating to the MIX computer explained in Volume 1. Appendix B contains a summary of the mathematical notations used, some of which are a little different from those found in traditional mathematics books.

In addition to the acknowledgments made in the preface to Volume 1, I would like to express deep appreciation to Elwyn R. Berlekamp, John Brillhart, George E. Collins, Stephen A. Cook, D. H. Lehmer, M. Donald MacLaren, Mervin E. Muller, Kenneth B. Stolarsky, and H. Zassenhaus, who have generously devoted considerable time to reading portions of the preliminary manuscript, and who have suggested many valuable improvements.

Princeton, New Jersey D. E. K.
October 1968

Preface to the Second Edition

My first plan, when beginning to prepare this new edition, was to make it like the second edition of Volume 1: I went through the entire book and tried to improve every page without greatly perturbing the page numbering. But the number of improvements turned out to be so great that the entire book needed to be typeset again. As a result, I decided to make this book the first test case

for a new computer typesetting system I have been developing. I hope that most readers will like the slight changes in format, since my aim has been to produce a book whose typography is of the highest possible quality—superior even to the fine appearance of the previous editions, in spite of the fact that a computer is now involved. If all goes well, the third edition of Volume 1 and the second edition of Volume 3, and all editions of Volumes 4 through 7, will be published in the present style.

The decision to reset this entire book has freed me from the shackles of the previous page numbering, so I have been able to make major refinements and to insert a lot of new material. I estimate that about 45 percent of the book has changed. I did try, however, to keep the exercise numbers from being substantially altered; although many of the old exercises have been replaced by new and better ones, the new exercises tend to relate to the same idea as before. The explosive growth of seminumerical research in recent years has of course made it impossible for me to insert all of the beautiful ideas in this field that have been discovered since 1968; but I think that this edition does contain an up-to-date survey of all the major paradigms and basic theory of the subject, and it seems reasonable to believe that very few of the topics discussed here will ever become obsolete.

The National Science Foundation and the Office of Naval Research have been particularly generous in their support of my research as I work on these books. I am also deeply grateful for the advice and unselfish assistance of many readers, too numerous to mention. In this regard I want to acknowledge especially the help of several people whose contributions have been really major: B. I. Aspvall, R. P. Brent, U. Dieter, M. J. Fischer, R. W. Gosper, D. C. Hoaglin, W. M. Kahan, F. M. Liang, J. F. Reiser, A. G. Waterman, S. Winograd, and M. C. Wunderlich. Furthermore Marion Howe and other people in the Addison–Wesley production department have been enormously helpful in untangling literally thousands of hand-written inserts so that a very chaotic manuscript has come out looking reasonably well-organized. I suppose some mistakes still remain, or have crept in, and I would like to fix them; therefore I will cheerfully pay $2.00 reward to the first finder of each technical, typographical, or historical error.

Stanford, California D. E. K.
July 1980

'Defendit numerus,' [there is safety in numbers]
is the maxim of the foolish;
'Deperdit numerus,' [there is ruin in numbers]
of the wise.

—C. C. COLTON (1820)

NOTES ON THE EXERCISES

THE EXERCISES in this set of books have been designed for self-study as well as classroom study. It is difficult, if not impossible, for anyone to learn a subject purely by reading about it, without applying the information to specific problems and thereby being encouraged to think about what has been read. Furthermore, we all learn best the things that we have discovered for ourselves. Therefore the exercises form a major part of this work; a definite attempt has been made to keep them as informative as possible and to select problems that are enjoyable to solve.

In many books, easy exercises are found mixed randomly among extremely difficult ones. This is sometimes unfortunate because readers like to know in advance how long a problem ought to take—otherwise they may just skip over all the problems. A classic example of such a situation is the book *Dynamic Programming* by Richard Bellman; this is an important, pioneering work in which a group of problems is collected together at the end of some chapters under the heading "Exercises and Research Problems," with extremely trivial questions appearing in the midst of deep, unsolved problems. It is rumored that someone once asked Dr. Bellman how to tell the exercises apart from the research problems, and he replied, "If you can solve it, it is an exercise; otherwise it's a research problem."

Good arguments can be made for including both research problems and very easy exercises in a book of this kind; therefore, to save the reader from the possible dilemma of determining which are which, *rating numbers* have been provided to indicate the level of difficulty. These numbers have the following general significance:

Rating *Interpretation*

00 An extremely easy exercise that can be answered immediately if the material of the text has been understood; such an exercise can almost always be worked "in your head."

10 A simple problem that makes you think over the material just read, but it is by no means difficult. It should be possible to do this in one minute at most; pencil and paper may be useful in obtaining the solution.

20 An average problem that tests basic understanding of the text material, but you may need about fifteen or twenty minutes to answer it completely.

30 A problem of moderate difficulty and/or complexity; this one may involve over two hours' work to solve satisfactorily.

40 Quite a difficult or lengthy problem that would be suitable for a term project in classroom situations. It is expected that a student will be able to solve the problem in a reasonable amount of time, but the solution is not trivial.

50 A research problem that has not yet been solved satisfactorily, as far as the author knew at the time of writing, although many people have tried. If you have found an answer to such a problem, you ought to write it up for publication; furthermore, the author of this book would appreciate hearing about the solution as soon as possible (provided that it is correct).

By interpolation in this "logarithmic" scale, the significance of other rating numbers becomes clear. For example, a rating of *17* would indicate an exercise that is a bit simpler than average. Problems with a rating of *50* that are subsequently solved by some reader may appear with a *45* rating in later editions of the book.

The author has earnestly tried to assign accurate rating numbers, but it is difficult for the person who makes up a problem to know just how formidable it will be for someone else to find a solution; and everyone has more aptitude for certain types of problems than for others. It is hoped that the rating numbers represent a good guess as to the level of difficulty, but they should be taken as general guidelines, not as absolute indicators.

This book has been written for readers with varying degrees of mathematical training and sophistication; as a result, some of the exercises are intended only for the use of more mathematically inclined readers. The rating is preceded by an *M* if the exercise involves mathematical concepts or motivation to a greater extent than necessary for someone who is primarily interested only in programming the algorithms themselves. An exercise is marked with the letters "*HM*" if its solution necessarily involves a knowledge of calculus or other higher mathematics not developed in this book. An "*HM*" designation does *not* necessarily imply difficulty.

Some exercises are preceded by an arrowhead, "▶"; this designates problems that are especially instructive and that are especially recommended. Of course, no reader/student is expected to work *all* of the exercises, so those that seem to be the most valuable have been singled out. (This is not meant to detract from the other exercises!) Each reader should at least make an attempt to solve all of the problems whose rating is *10* or less; and the arrows may help to indicate which of the problems with a higher rating should be given priority.

Solutions to most of the exercises appear in the answer section. Please use them wisely; do not turn to the answer until you have made a genuine effort to solve the problem by yourself, or unless you do not have time to work this particular problem. *After* getting your own solution or giving the problem a decent try, you may find the answer instructive and helpful. The solution given will often be quite short, and it will sketch the details under the assumption that you have earnestly tried to solve it by your own means first. Sometimes the solution gives less information than was asked; often it gives more. It is quite

possible that you may have a better answer than the one published here, or you may have found an error in the published solution; in such a case, the author will be pleased to know the details. Later editions of this book will give the improved solutions together with the solver's name where appropriate.

When working an exercise you may generally use the answers to previous exercises, unless specifically forbidden from doing so. The rating numbers have been assigned with this in mind; thus it is possible for exercise $n + 1$ to have a lower rating than exercise n, even though it includes the result of exercise n as a special case.

Summary of codes:			
		00	Immediate
		10	Simple (one minute)
		20	Medium (quarter hour)
▶	Recommended	*30*	Moderately hard
M	Mathematically oriented	*40*	Term project
HM	Requiring "higher math"	*50*	Research problem

EXERCISES

▶ **1.** [*00*] What does the rating "*M20*" mean?

2. [*10*] Of what value can the exercises in a textbook be to the reader?

3. [*M50*] Prove that when n is an integer, $n > 2$, the equation $x^n + y^n = z^n$ has no solution in positive integers x, y, z.

Exercise is the beste intrument in learnyng.

—ROBERT RECORDE (*The Whetstone of Witte*, 1557)

CONTENTS

JMK
JSK

CHAPTER THREE

RANDOM NUMBERS

Anyone who considers arithmetical
methods of producing random digits
is, of course, in a state of sin.

—JOHN VON NEUMANN (1951)

Lest men suspect your tale untrue,
Keep probability in view.

—JOHN GAY (1727)

There wanted not some beams of light
to guide men in the exercise of their Stocastick faculty.

—JOHN OWEN (1662)

3.1. INTRODUCTION

NUMBERS that are "chosen at random" are useful in many different kinds of applications. For example:

a) *Simulation.* When a computer is being used to simulate natural phenomena, random numbers are required to make things realistic. Simulation covers many fields, from the study of nuclear physics (where particles are subject to random collisions) to operations research (where people come into, say, an airport at random intervals).

b) *Sampling.* It is often impractical to examine all possible cases, but a random sample will provide insight into what constitutes "typical" behavior.

c) *Numerical analysis.* Ingenious techniques for solving complicated numerical problems have been devised using random numbers. Several books have been written on this subject.

d) *Computer programming.* Random values make a good source of data for testing the effectiveness of computer algorithms. This is the primary application of interest to us in this series of books; it accounts for the fact that random numbers are already being considered here in Chapter 3, before most of the other computer algorithms have appeared.

Several people experimented with the middle-square method in the early 1950s. Working with numbers that have four digits instead of ten, G. E. Forsythe tried 16 different starting values and found that 12 of them led to sequences ending with the cycle 6100, 2100, 4100, 8100, 6100, ..., while two of them degenerated to zero. N. Metropolis also conducted extensive tests on the middle-square method, mostly in the binary number system. He showed that when 20-bit numbers are being used, there are 13 different cycles into which the sequence might degenerate, the longest of which has a period of length 142.

It is fairly easy to restart the middle-square method on a new value when zero has been detected, but long cycles are somewhat harder to avoid. Exercise 7 discusses some interesting ways to determine the cycles of periodic sequences, using very little memory space.

A theoretical disadvantage of the middle-square method is given in exercises 9 and 10. On the other hand, working with 38-bit numbers, Metropolis obtained a sequence of about 750,000 numbers before degeneracy occurred, and the resulting $750{,}000 \times 38$ bits satisfactorily passed statistical tests for randomness. This shows that the middle-square method *can* give usable results, but it is rather dangerous to put much faith in it until after elaborate computations have been performed.

Many random number generators in use today are not very good. There is a tendency for people to avoid learning anything about such subroutines; quite often we find that some old method that is comparatively unsatisfactory has blindly been passed down from one programmer to another, and today's users have no understanding of its limitations. We shall see in this chapter that it is not difficult to learn the most important facts about random number generators and their proper use.

It is not easy to invent a foolproof source of random numbers. This fact was convincingly impressed upon the author several years ago, when he attempted to create a fantastically good generator using the following peculiar approach:

Algorithm K (*"Super-random" number generator*). Given a 10-digit decimal number X, this algorithm may be used to change X to the number that should come next in a supposedly random sequence. Although the algorithm might be expected to yield quite a random sequence, reasons given below show that it is, in fact, not very good at all. (The reader need not study this algorithm in great detail except to observe how complicated it is; note, in particular, steps K1 and K2.)

K1. [Choose number of iterations.] Set $Y \leftarrow \lfloor X/10^9 \rfloor$, the most significant digit of X. (We will execute steps K2 through K13 exactly $Y + 1$ times; that is, we will apply randomizing transformations a *random* number of times.)

K2. [Choose random step.] Set $Z \leftarrow \lfloor X/10^8 \rfloor \bmod 10$, the second most significant digit of X. Go to step K$(3 + Z)$. (That is, we now jump to a *random* step in the program.)

K3. [Ensure $\geq 5 \times 10^9$.] If $X < 5000000000$, set $X \leftarrow X + 5000000000$.

device. A famous random-number machine called ERNIE has been used to pick the winning numbers in the British Premium Savings Bonds lottery. [See the articles by Kendall and Babington-Smith in *J. Royal Stat. Soc.*, Series A, **101** (1938), 147–166, and Series B, **6** (1939), 51–61; see also the review of the RAND table in *Math. Comp.* **10** (1956), 39–43, and the discussion of ERNIE by W. E. Thomson et al., *J. Royal Stat. Soc.*, Series A, **122** (1959), 301–333.]

Shortly after computers were introduced, people began to search for efficient ways to obtain random numbers within computer programs. A table can be used, but this method is of limited utility because of the memory space and input time requirement, because the table may be too short, and because it is a bit of a nuisance to prepare and maintain the table. Machines such as ERNIE might be attached to the computer, but this would be unsatisfactory since it would be impractical to reproduce calculations exactly a second time when checking out a program; and such machines have tended to suffer from malfunctions that are difficult to detect.

The inadequacy of these methods led to an interest in the production of random numbers using the arithmetic operations of a computer. John von Neumann first suggested this approach in about 1946, using the "middle-square" method. His idea was to take the square of the previous random number and to extract the middle digits; for example, if we are generating 10-digit numbers and the previous value was 5772156649, we square it to get

$$33317792380594909201,$$

and the next number is therefore 7923805949.

There is a fairly obvious objection to this technique: how can a sequence generated in such a way be random, since each number is completely determined by its predecessor? The answer is that this sequence *isn't* random, but it *appears* to be. In typical applications the actual relationship between one number and its successor has no physical significance; hence the nonrandom character is not really undesirable. Intuitively, the middle square seems to be a fairly good scrambling of the previous number.

Sequences generated in a deterministic way such as this are usually called *pseudo-random* or *quasi-random* sequences in the highbrow technical literature, but in this book we shall simply call them random sequences, with the understanding that they only *appear* to be random. Being "apparently random" is perhaps all that can be said about any random sequence anyway. Random numbers generated deterministically on computers have worked quite well in nearly every application, provided that a suitable method has been carefully selected. Of course, deterministic sequences aren't always the answer; they certainly shouldn't replace ERNIE for the lotteries.

Von Neumann's original "middle-square method" has actually proved to be a comparatively poor source of random numbers. The danger is that the sequence tends to get into a rut, a short cycle of repeating elements. For example, if zero ever appears as a number of the sequence, it will continually perpetuate itself.

e) *Decision making.* There are reports that many executives make their decisions by flipping a coin or by throwing darts, etc. It is also rumored that some college professors prepare their grades on such a basis. Sometimes it is important to make a completely "unbiased" decision; this ability is occasionally useful in computer algorithms, for example in situations where a fixed decision made each time would cause the algorithm to run more slowly. Randomness is also an essential part of optimal strategies in the theory of games.

f) *Recreation.* Rolling dice, shuffling decks of cards, spinning roulette wheels, etc., are fascinating pastimes for just about everybody. These traditional uses of random numbers have suggested the name "Monte Carlo method," a general term used to describe any algorithm that employs random numbers.

People who think about this topic almost invariably get into philosophical discussions about what the word "random" means. In a sense, there is no such thing as a random number; for example, is 2 a random number? Rather, we speak of a *sequence of independent random numbers* with a specified *distribution*, and this means loosely that each number was obtained merely by chance, having nothing to do with other numbers of the sequence, and that each number has a specified probability of falling in any given range of values.

A *uniform* distribution on a finite set of numbers is one in which each possible number is equally probable. A distribution is generally understood to be uniform unless some other distribution is specifically mentioned.

Each of the ten digits 0 through 9 will occur about $\frac{1}{10}$ of the time in a (uniform) sequence of random digits. Each pair of two successive digits should occur about $\frac{1}{100}$ of the time, etc. Yet if we take a truly random sequence of a million digits, it will not always have exactly 100,000 zeros, 100,000 ones, etc. In fact, chances of this are quite slim; a *sequence* of such sequences will have this character on the average.

Any specified sequence of a million digits is equally as probable as the sequence consisting of a million *zeros*. Thus, if we are choosing a million digits at random and if the first 999,999 of them happen to come out to be zero, the chance that the final digit is zero is still exactly $\frac{1}{10}$, in a truly random situation. These statements seem paradoxical to many people, but there is really no contradiction involved.

There are several ways to formulate decent abstract definitions of randomness, and we will return to this interesting subject in Section 3.5; but for the moment, let us content ourselves with an intuitive understanding of the concept.

At first, people who needed random numbers in their scientific work would draw balls out of a "well-stirred urn" or would roll dice or deal out cards. A table of over 40,000 random digits, "taken at random from census reports," was published in 1927 by L. H. C. Tippett. Since then, a number of devices have been built to generate random numbers mechanically; the first such machine was used in 1939 by M. G. Kendall and B. Babington-Smith to produce a table of 100,000 random digits, and in 1955 the RAND Corporation published a widely used table of a million random digits obtained with the help of another special

K4. [Middle square.] Replace X by $\lfloor X^2/10^5 \rfloor \bmod 10^{10}$, i.e., by the middle of the square of X.

K5. [Multiply.] Replace X by $(1001001001\,X) \bmod 10^{10}$.

K6. [Pseudo-complement.] If $X < 100000000$, then set $X \leftarrow X + 9814055677$; otherwise set $X \leftarrow 10^{10} - X$.

K7. [Interchange halves.] Interchange the low-order five digits of X with the high-order five digits, i.e., $X \leftarrow 10^5(X \bmod 10^5) + \lfloor X/10^5 \rfloor$, the middle 10 digits of $(10^{10} + 1)X$.

K8. [Multiply.] Same as step K5.

K9. [Decrease digits.] Decrease each nonzero digit of the decimal representation of X by one.

K10. [99999 modify.] If $X < 10^5$, set $X \leftarrow X^2 + 99999$; otherwise set $X \leftarrow X - 99999$.

K11. [Normalize.] (At this point X cannot be zero.) If $X < 10^9$, set $X \leftarrow 10X$ and repeat this step.

K12. [Modified middle square.] Replace X by $\lfloor X(X-1)/10^5 \rfloor \bmod 10^{10}$, i.e., by the middle 10 digits of $X(X-1)$.

K13. [Repeat?] If $Y > 0$, decrease Y by 1 and return to step K2. If $Y = 0$, the algorithm terminates with X as the desired "random" value. ∎

(The machine-language program corresponding to the above algorithm was intended to be so complicated that a person reading a listing of it without explanatory comments wouldn't know what the program was doing.)

Considering all the contortions of Algorithm K, doesn't it seem plausible that it should produce almost an infinite supply of unbelievably random numbers? No! In fact, when this algorithm was first put onto a computer, it almost immediately converged to the 10-digit value 6065038420, which—by extraordinary coincidence—is transformed into itself by the algorithm (see Table 1). With another starting number, the sequence began to repeat after 7401 values, in a cyclic period of length 3178.

The moral of this story is that *random numbers should not be generated with a method chosen at random.* Some theory should be used.

In this chapter, we shall consider random number generators that are superior to the middle-square method and to Algorithm K; the corresponding sequences are guaranteed to have certain desirable random properties, and no degeneracy will occur. We shall explore the reasons for this random behavior in some detail, and we shall also consider techniques for manipulating random numbers. For example, one of our investigations will be the shuffling of a simulated deck of cards within a computer program.

Section 3.6 summarizes this chapter and lists several bibliographic sources.

Table 1

A COLOSSAL COINCIDENCE: THE NUMBER 6065038420 IS TRANSFORMED INTO ITSELF BY ALGORITHM K.

Step	X (after)	
K1	6065038420	
K3	6065038420	
K4	6910360760	
K5	8031120760	
K6	1968879240	
K7	7924019688	
K8	9631707688	
K9	8520606577	
K10	8520506578	
K11	8520506578	
K12	0323372207	$Y = 6$
K6	9676627793	
K7	2779396766	
K8	4942162766	
K9	3831051655	
K10	3830951656	
K11	3830951656	
K12	1905867781	$Y = 5$
K12	3319967479	$Y = 4$
K6	6680032521	
K7	3252166800	
K8	2218966800	
K9	1107855700	
K10	1107755701	
K11	1107755701	
K12	1226919902	$Y = 3$
K5	0048821902	
K6	9862877579	
K7	7757998628	
K8	2384626628	
K9	1273515517	
K10	1273415518	
K11	1273415518	
K12	5870802097	$Y = 2$
K11	5870802097	
K12	3172562687	$Y = 1$
K4	1540029446	
K5	7015475446	
K6	2984524554	
K7	2455429845	
K8	2730274845	
K9	1620163734	
K10	1620063735	
K11	1620063735	
K12	6065038420	$Y = 0$

EXERCISES

▸ **1.** [*20*] Suppose that you wish to obtain a decimal digit at random, not using a computer. Which of the following methods would be suitable?

a) Open a telephone directory to a random place (i.e., stick your finger in it somewhere) and use the units digit of the first number found on the selected page.

b) Same as (a), but use the units digit of the page number.

c) Roll a die that is in the shape of a regular icosahedron, whose twenty faces have been labeled with the digits $0, 0, 1, 1, \ldots, 9, 9$. Use the digit that appears on top, when the die comes to rest. (A felt table with a hard surface is recommended for rolling dice.)

d) Expose a geiger counter to a source of radioactivity for one minute (shielding yourself) and use the units digit of the resulting count. Assume that the geiger counter displays the number of counts in decimal notation, and that the count is initially zero.

e) Glance at your wristwatch; and if the position of the second-hand is between $6n$ and $6(n + 1)$ seconds, choose the digit n.

f) Ask a friend to think of a random digit, and use the digit he names.

g) Ask an enemy to think of a random digit, and use the digit he names.

h) Assume that 10 horses are entered in a race and that you know nothing whatever about their qualifications. Assign to these horses the digits 0 to 9, in arbitrary fashion, and after the race use the winner's digit.

2. [*M22*] In a random sequence of a million decimal digits, what is the probability that there are exactly 100,000 of each possible digit?

3. [*10*] What number follows 1010101010 in the middle-square method?

4. [*10*] Why can't the value of X be zero when step K11 of Algorithm K is performed? What would be wrong with the algorithm if X could be zero?

5. [*15*] Explain why, in any case, Algorithm K should not be expected to provide "infinitely many" random numbers, in the sense that (even if the coincidence given in Table 1 had not occurred) one knows in advance that any sequence generated by Algorithm K will eventually be periodic.

▶ **6.** [*M21*] Suppose that we want to generate a sequence of integers $X_0, X_1, X_2, \ldots$, in the range $0 \le X_n < m$. Let $f(x)$ be any function such that $0 \le x < m$ implies $0 \le f(x) < m$. Consider a sequence formed by the rule $X_{n+1} = f(X_n)$. (Examples are the middle-square method and Algorithm K.)

a) Show that the sequence is ultimately periodic, in the sense that there exist numbers λ and μ for which the values $X_0, X_1, \ldots, X_\mu, \ldots, X_{\mu+\lambda-1}$ are distinct, but $X_{n+\lambda} = X_n$ when $n \ge \mu$. Find the maximum and minimum possible values of μ and λ.

b) (R. W. Floyd.) Show that there exists an $n > 0$ such that $X_n = X_{2n}$; and the smallest such value of n lies in the range $\mu \le n \le \mu + \lambda$. Furthermore the value of X_n is unique in the sense that if $X_n = X_{2n}$ and $X_r = X_{2r}$, then $X_r = X_n$.

c) Use the idea of part (b) to design an algorithm that calculates μ and λ for any given function f and any given X_0, using only $O(\mu + \lambda)$ steps and only a bounded number of memory locations.

▶ **7.** [*M21*] (R. P. Brent, 1977.) Let $\ell(n)$ be the least power of 2 that is less than or equal to n; thus, for example, $\ell(15) = 8$ and $\ell(\ell(n)) = \ell(n)$.

a) Show that, in terms of the notation in exercise 6, there exists an $n > 0$ such that $X_n = X_{\ell(n)-1}$. Find a formula that expresses the least such n in terms of μ and λ.

b) Apply this result to design an algorithm that can be used in conjunction with any random number generator of the type $X_{n+1} = f(X_n)$, to prevent it from cycling indefinitely. Your algorithm should calculate the period length λ, and it should use only a small amount of memory space—you must not simply store all of the computed sequence values!

8. [*28*] Make a complete examination of the middle-square method in the case of two-digit decimal numbers. (a) We might start the process out with any of the 100 possible values 00, 01, ..., 99. How many of these values lead ultimately to the repeating cycle 00, 00, ...? [*Example:* Starting with 43, we obtain the sequence 43, 84, 05, 02, 00, 00, 00, ...] (b) How many possible final cycles are there? How long is the longest cycle? (c) What starting value or values will give the largest number of distinct elements before the sequence repeats?

9. [*M14*] Prove that the middle-square method using $2n$-digit numbers to the base b has the following disadvantage: If the sequence includes any number whose most significant n digits are zero, the succeeding numbers will get smaller and smaller until zero occurs repeatedly.

10. [*M16*] Under the assumptions of the preceding exercise, what can you say about the sequence of numbers following X if the *least* significant n digits of X are zero? What if the least significant $n+1$ digits are zero?

▶ **11.** [*M26*] Consider sequences of random number generators having the form described in exercise 6. If we choose $f(x)$ and X_0 at random, i.e., if we assume that each of the m^m possible functions $f(x)$ is equally probable and that each of the m possible values of X_0 is equally probable, what is the probability that the sequence will eventually degenerate into a cycle of length $\lambda = 1$? (*Note:* The assumptions of this problem give a natural way to think of a "random" random number generator of this type. A method such as Algorithm K may be expected to behave somewhat like the generator considered here; the answer to this problem gives a measure of how "colossal" the coincidence of Table 1 really is.)

▶ **12.** [*M31*] Under the assumptions of the preceding exercise, what is the average length of the final cycle? What is the average length of the sequence before it begins to cycle? (In the notation of exercise 6, we wish to examine the average values of λ and of $\mu+\lambda$.)

13. [*M42*] If $f(x)$ is chosen at random in the sense of exercise 11, what is the average length of the *longest* cycle obtainable by varying the starting value X_0? (*Note:* We have already considered the analogous problem in the case that $f(x)$ is a random permutation; see exercise 1.3.3–23.)

14. [*M38*] If $f(x)$ is chosen at random in the sense of exercise 11, what is the average number of distinct final cycles obtainable by varying the starting value? [Cf. exercise 8(b).]

15. [*M15*] If $f(x)$ is chosen at random in the sense of exercise 11, what is the probability that none of the final cycles has length 1, regardless of the choice of X_0?

16. [*15*] A sequence generated as in exercise 6 must begin to repeat after at most m values have been generated. Suppose we generalize the method so that X_{n+1} depends on X_{n-1} as well as on X_n; formally, let $f(x,y)$ be a function such that $0 \le x, y < m$ implies $0 \le f(x,y) < m$. The sequence is constructed by selecting X_0 and X_1 arbitrarily, and then letting

$$X_{n+1} = f(X_n, X_{n-1}), \qquad \text{for } n > 0.$$

What is the maximum period conceivably attainable in this case?

17. [*10*] Generalize the situation in the previous exercise so that X_{n+1} depends on the preceding k values of the sequence.

18. [*M20*] Invent a method analogous to that of exercise 7 for finding cycles in the general form of random number generator discussed in exercise 17.

19. [*M48*] Solve the problems of exercises 11 through 15 for the more general case that X_{n+1} depends on the preceding k values of the sequence; each of the m^{m^k} functions $f(x_1, \ldots, x_k)$ is to be considered equally probable. (*Note:* The number of functions that yield the *maximum* period is analyzed in exercise 2.3.4.2–23.)

3.2. GENERATING UNIFORM RANDOM NUMBERS

IN THIS SECTION we shall consider methods for generating a sequence of random fractions, i.e., random *real numbers U_n, uniformly distributed between zero and one.* Since a computer can represent a real number with only finite accuracy, we shall actually be generating integers X_n between zero and some number m; the fraction

$$U_n = X_n/m$$

will then lie between zero and one. Usually m is the word size of the computer, so X_n may be regarded (conservatively) as the integer contents of a computer word with the radix point assumed at the extreme right, and U_n may be regarded (liberally) as the contents of the same word with the radix point assumed at the extreme left.

3.2.1. The Linear Congruential Method

By far the most popular random number generators in use today are special cases of the following scheme, introduced by D. H. Lehmer in 1949. [See *Proc. 2nd Symp. on Large-Scale Digital Calculating Machinery* (Cambridge: Harvard University Press, 1951), 141–146.] We choose four "magic numbers":

$$\begin{array}{rll} m, & \text{the modulus;} & m > 0. \\ a, & \text{the multiplier;} & 0 \le a < m. \\ c, & \text{the increment;} & 0 \le c < m. \\ X_0, & \text{the starting value;} & 0 \le X_0 < m. \end{array} \tag{1}$$

The desired sequence of random numbers $\langle X_n \rangle$ is then obtained by setting

$$X_{n+1} = (aX_n + c) \bmod m, \qquad n \ge 0. \tag{2}$$

This is called a *linear congruential sequence.* Taking the remainder mod m is somewhat like determining where a ball will land in a spinning roulette wheel.

For example, the sequence obtained when $m = 10$ and $X_0 = a = c = 7$ is

$$7, 6, 9, 0, 7, 6, 9, 0, \ldots. \tag{3}$$

As this example shows, the sequence is not always "random" for all choices of m, a, c, and X_0; the principles of choosing the magic numbers appropriately will be investigated carefully in later parts of this chapter.

Example (3) illustrates the fact that the congruential sequences always "get into a loop"; i.e., there is ultimately a cycle of numbers that is repeated endlessly. This property is common to all sequences having the general form $X_{n+1} = f(X_n)$; see exercise 3.1–6. The repeating cycle is called the *period*; sequence (3) has a period of length 4. A useful sequence will of course have a relatively long period.

The special case $c = 0$ deserves explicit mention, since the number generation process is a little faster when $c = 0$ than it is when $c \neq 0$. We shall see later that the restriction $c = 0$ cuts down the length of the period of the sequence, but it is still possible to make the period reasonably long. Lehmer's original generation method had $c = 0$, although he mentioned $c \neq 0$ as a possibility; the idea of taking $c \neq 0$ to obtain longer periods is due to Thomson [*Comp. J.* **1** (1958), 83, 86] and, independently, to Rotenberg [*JACM* **7** (1960), 75–77]. The terms *multiplicative congruential method* and *mixed congruential method* are used by many authors to denote linear congruential sequences with $c = 0$ and $c \neq 0$, respectively.

The letters m, a, c, and X_0 will be used throughout this chapter in the sense described above. Furthermore, we will find it useful to define

$$b = a - 1, \tag{4}$$

in order to simplify many of our formulas.

We can immediately reject the case $a = 1$, for this would mean that $X_n = (X_0 + nc) \bmod m$, and the sequence would certainly not behave as a random sequence. The case $a = 0$ is even worse. Hence for practical purposes we may assume that

$$a \geq 2, \qquad b \geq 1. \tag{5}$$

Now we can prove a generalization of Eq. (2),

$$X_{n+k} = \big(a^k X_n + (a^k - 1)c/b\big) \bmod m, \qquad k \geq 0, \quad n \geq 0, \tag{6}$$

which expresses the $(n+k)$th term directly in terms of the nth term. (The special case $n = 0$ in this equation is worthy of note.) It follows that the subsequence consisting of every kth term of $\langle X_n \rangle$ is another linear congruential sequence, having the multiplier $a^k \bmod m$ and the increment $\big((a^k - 1)c/b\big) \bmod m$.

An important corollary of (6) is that the general sequence defined by m, a, c, and X_0 can be expressed very simply in terms of the special case where $c = 1$ and $X_0 = 0$. Let

$$Y_0 = 0, \qquad Y_{n+1} = (aY_n + 1) \bmod m. \tag{7}$$

According to Eq. (6) we will have $Y_k \equiv (a^k - 1)/b$ (modulo m), hence the general sequence defined in (2) satisfies

$$X_n = (AY_n + X_0) \bmod m, \qquad \text{where } A = (X_0 b + c) \bmod m. \tag{8}$$

EXERCISES

1. [*10*] Example (3) shows a situation in which $X_4 = X_0$, so the sequence begins again from the beginning. Give an example of a linear congruential sequence with $m = 10$ for which X_0 never appears again in the sequence.

▸ **2.** [*M20*] Show that if a and m are relatively prime, the number X_0 will always appear in the period.

3. [*M10*] If a and m are not relatively prime, explain why the sequence will be somewhat handicapped and probably not very random; hence we will generally want the multiplier a to be relatively prime to the modulus m.

4. [*11*] Prove Eq. (6).

5. [*M20*] Equation (6) holds for $k \geq 0$. If possible, give a formula that expresses X_{n+k} in terms of X_n for *negative* values of k.

3.2.1.1. Choice of modulus. Our current goal is to find good values for the parameters that define a linear congruential sequence. Let us first consider the proper choice of the number m. We want m to be rather large, since the period cannot have more than m elements. (Even if a person wants to generate only random zeros and ones, he should *not* take $m = 2$, for then the sequence would at best have the form $\ldots, 0, 1, 0, 1, 0, 1, \ldots$! Methods for modifying random numbers to get random zeros and ones are discussed in Section 3.4.)

Another factor that influences our choice of m is speed of generation: We want to pick a value so that the computation of $(aX_n + c) \bmod m$ is fast.

Consider MIX as an example. We can compute $y \bmod m$ by putting y in registers A and X and dividing by m; assuming that y and m are positive, we see that $y \bmod m$ will then appear in register X. But division is a comparatively slow operation, and it can be avoided if we take m to be a value that is especially convenient, such as the *word size* of our computer.

Let w be the computer's word size, namely, 2^e on an e-bit binary computer or 10^e on an e-digit decimal machine. (In this book we shall often use the letter e to denote an arbitrary integer exponent, instead of the base of natural logarithms, hoping that the context will make our notation unambiguous. Physicists have a similar problem when they use e for the charge on an electron.) The result of an addition operation is usually given modulo w, except on ones'-complement machines; and multiplication mod w is also quite simple, since the desired result is the lower half of the product. Thus, the following program computes the quantity $(aX + c) \bmod w$ efficiently:

```
LDA   A   rA ← a.
MUL   X   rAX ← (rA) · X.
SLAX  5   rA ← rAX mod w.
ADD   C   rA ← (rA + c) mod w.   ∎
```
(1)

The result appears in register A. The overflow toggle might be on at the conclusion of the above sequence of instructions, and if this is undesirable, the code should be followed by, e.g., "JOV *+1" to turn it off.

A clever technique that is less commonly known can be used to perform computations modulo $(w+1)$. For reasons to be explained later, we will generally want $c = 0$ when $m = w + 1$, so we merely need to compute $(aX) \bmod (w + 1)$.

The following program does this:

```
01   LDAN X          rA ← −X.
02   MUL  A          rAX ← (rA) · a.
03   STX  TEMP
04   SUB  TEMP       rA ← rA − rX.                          (2)
05   JANN *+3        Exit if rA ≥ 0.
06   INCA 2          rA ← rA + 2.
07   ADD  =w − 1=    rA ← rA + w − 1.  (Cf. exercise 3.)  ∎
```

Register A now contains the value $(aX) \bmod (w+1)$. Of course, this value might lie anywhere between 0 and w, inclusive, so the reader may legitimately wonder how we can represent so many values in the A-register! (The register obviously cannot hold a number larger than $w-1$.) The answer is that overflow will be on after the above program if and only if the result equals w, assuming that overflow was initially off. It is convenient simply to reject the value w if it appears in the congruential sequence modulo $w+1$; this will happen if lines 05 and 06 of (2) are replaced by "`JANN *+4; INCA 2; JAP *-5`".

To prove that code (2) actually does determine $(aX) \bmod (w+1)$, note that in line 04 we are subtracting the lower half of the product from the upper half. No overflow can occur at this step; and if $aX = qw + r$, with $0 \le r < w$, we will have the quantity $r - q$ in register A after line 04. Now

$$aX = q(w+1) + (r-q),$$

and since $q < w$, we have $-w < r - q < w$; hence $(aX) \bmod (w+1)$ equals either $r-q$ or $r-q+(w+1)$, depending on whether $r-q \ge 0$ or $r-q < 0$.

A similar technique can be used to get the product of two numbers modulo $(w-1)$; see exercise 8.

In later sections we shall require a knowledge of the prime factors of m in order to choose the multiplier a correctly. Table 1 lists the complete factorization of $w \pm 1$ into primes for nearly every known computer word size; the methods of Section 4.5.4 can be used to extend this table if desired.

The reader may well ask why we bother to consider using $m = w \pm 1$, when the choice $m = w$ is so manifestly convenient. The reason is that *when $m = w$, the right-hand digits of X_n are much less random than the left-hand digits.* If d is a divisor of m, and if

$$Y_n = X_n \bmod d, \tag{3}$$

we can easily show that

$$Y_{n+1} = (aY_n + c) \bmod d. \tag{4}$$

(For, $X_{n+1} = aX_n + c - qm$ for some integer q, and taking both sides mod d causes the quantity qm to drop out when d is a factor of m.)

To illustrate the significance of Eq. (4), let us suppose, for example, that we have a binary computer. If $m = w = 2^e$, the low-order four bits of X_n are

Table 1

PRIME FACTORIZATIONS OF $w \pm 1$

$2^e - 1$	e	$2^e + 1$
$7 \cdot 31 \cdot 151$	15	$3^2 \cdot 11 \cdot 331$
$3 \cdot 5 \cdot 17 \cdot 257$	16	65537
131071	17	$3 \cdot 43691$
$3^3 \cdot 7 \cdot 19 \cdot 73$	18	$5 \cdot 13 \cdot 37 \cdot 109$
524287	19	$3 \cdot 174763$
$3 \cdot 5^2 \cdot 11 \cdot 31 \cdot 41$	20	$17 \cdot 61681$
$7^2 \cdot 127 \cdot 337$	21	$3^2 \cdot 43 \cdot 5419$
$3 \cdot 23 \cdot 89 \cdot 683$	22	$5 \cdot 397 \cdot 2113$
$47 \cdot 178481$	23	$3 \cdot 2796203$
$3^2 \cdot 5 \cdot 7 \cdot 13 \cdot 17 \cdot 241$	24	$97 \cdot 257 \cdot 673$
$31 \cdot 601 \cdot 1801$	25	$3 \cdot 11 \cdot 251 \cdot 4051$
$3 \cdot 2731 \cdot 8191$	26	$5 \cdot 53 \cdot 157 \cdot 1613$
$7 \cdot 73 \cdot 262657$	27	$3^4 \cdot 19 \cdot 87211$
$3 \cdot 5 \cdot 29 \cdot 43 \cdot 113 \cdot 127$	28	$17 \cdot 15790321$
$233 \cdot 1103 \cdot 2089$	29	$3 \cdot 59 \cdot 3033169$
$3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$	30	$5^2 \cdot 13 \cdot 41 \cdot 61 \cdot 1321$
2147483647	31	$3 \cdot 715827883$
$3 \cdot 5 \cdot 17 \cdot 257 \cdot 65537$	32	$641 \cdot 6700417$
$7 \cdot 23 \cdot 89 \cdot 599479$	33	$3^2 \cdot 67 \cdot 683 \cdot 20857$
$3 \cdot 43691 \cdot 131071$	34	$5 \cdot 137 \cdot 953 \cdot 26317$
$31 \cdot 71 \cdot 127 \cdot 122921$	35	$3 \cdot 11 \cdot 43 \cdot 281 \cdot 86171$
$3^3 \cdot 5 \cdot 7 \cdot 13 \cdot 19 \cdot 37 \cdot 73 \cdot 109$	36	$17 \cdot 241 \cdot 433 \cdot 38737$
$223 \cdot 616318177$	37	$3 \cdot 1777 \cdot 25781083$
$3 \cdot 174763 \cdot 524287$	38	$5 \cdot 229 \cdot 457 \cdot 525313$
$7 \cdot 79 \cdot 8191 \cdot 121369$	39	$3^2 \cdot 2731 \cdot 22366891$
$3 \cdot 5^2 \cdot 11 \cdot 17 \cdot 31 \cdot 41 \cdot 61681$	40	$257 \cdot 4278255361$
$13367 \cdot 164511353$	41	$3 \cdot 83 \cdot 8831418697$
$3^2 \cdot 7^2 \cdot 43 \cdot 127 \cdot 337 \cdot 5419$	42	$5 \cdot 13 \cdot 29 \cdot 113 \cdot 1429 \cdot 14449$
$431 \cdot 9719 \cdot 2099863$	43	$3 \cdot 2932031007403$
$3 \cdot 5 \cdot 23 \cdot 89 \cdot 397 \cdot 683 \cdot 2113$	44	$17 \cdot 353 \cdot 2931542417$
$7 \cdot 31 \cdot 73 \cdot 151 \cdot 631 \cdot 23311$	45	$3^3 \cdot 11 \cdot 19 \cdot 331 \cdot 18837001$
$3 \cdot 47 \cdot 178481 \cdot 2796203$	46	$5 \cdot 277 \cdot 1013 \cdot 1657 \cdot 30269$
$2351 \cdot 4513 \cdot 13264529$	47	$3 \cdot 283 \cdot 165768537521$
$3^2 \cdot 5 \cdot 7 \cdot 13 \cdot 17 \cdot 97 \cdot 241 \cdot 257 \cdot 673$	48	$193 \cdot 65537 \cdot 22253377$
$179951 \cdot 3203431780337$	59	$3 \cdot 2833 \cdot 37171 \cdot 1824726041$
$3^2 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 31 \cdot 41 \cdot 61 \cdot 151 \cdot 331 \cdot 1321$	60	$17 \cdot 241 \cdot 61681 \cdot 4562284561$
$7^2 \cdot 73 \cdot 127 \cdot 337 \cdot 92737 \cdot 649657$	63	$3^3 \cdot 19 \cdot 43 \cdot 5419 \cdot 77158673929$
$3 \cdot 5 \cdot 17 \cdot 257 \cdot 641 \cdot 65537 \cdot 6700417$	64	$274177 \cdot 67280421310721$

$10^e - 1$	e	$10^e + 1$
$3^3 \cdot 7 \cdot 11 \cdot 13 \cdot 37$	6	$101 \cdot 9901$
$3^2 \cdot 239 \cdot 4649$	7	$11 \cdot 909091$
$3^2 \cdot 11 \cdot 73 \cdot 101 \cdot 137$	8	$17 \cdot 5882353$
$3^4 \cdot 37 \cdot 333667$	9	$7 \cdot 11 \cdot 13 \cdot 19 \cdot 52579$
$3^2 \cdot 11 \cdot 41 \cdot 271 \cdot 9091$	10	$101 \cdot 3541 \cdot 27961$
$3^2 \cdot 21649 \cdot 513239$	11	$11^2 \cdot 23 \cdot 4093 \cdot 8779$
$3^3 \cdot 7 \cdot 11 \cdot 13 \cdot 37 \cdot 101 \cdot 9901$	12	$73 \cdot 137 \cdot 99990001$
$3^2 \cdot 11 \cdot 17 \cdot 73 \cdot 101 \cdot 137 \cdot 5882353$	16	$353 \cdot 449 \cdot 641 \cdot 1409 \cdot 69857$

the numbers $Y_n = X_n \bmod 2^4$. The gist of Eq. (4) is that the low-order four bits of $\langle X_n \rangle$ form a congruential sequence that has a period of length 16 or less. Similarly, the low-order five bits are periodic with a period of at most 32; and the least significant bit of X_n is either constant or strictly alternating.

This situation does not occur when $m = w \pm 1$; in such a case, the low-order bits of X_n will behave just as randomly as the high-order bits do. If, for example, $w = 2^{35}$ and $m = 2^{35} - 1$, the numbers of the sequence will not be very random if we consider only their remainders mod 31, 71, 127, or 122921 (cf. Table 1); but the low-order bit, which represents the numbers of the sequence taken mod 2, would be satisfactorily random.

Another alternative is to let m be the largest prime number less than w. This prime may be found by using the techniques of Section 4.5.4, and a table of suitably large primes appears in that section.

In most applications, the low-order bits are insignificant, and the choice $m = w$ is quite satisfactory—provided that the programmer using the random numbers does so wisely.

Our discussion so far has been based on a "signed magnitude" computer like MIX. Similar ideas apply to machines that use complement notations, although there are some instructive variations. For example, a DEC 20 computer has 36 bits with two's complement arithmetic; when it computes the product of two nonnegative integers, the lower half contains the least significant 35 bits with a plus sign. On this machine we should therefore take $w = 2^{35}$, not 2^{36}. The 32-bit two's complement arithmetic on IBM System/370 computers is different: the lower half of a product contains a full 32 bits. Some programmers have felt that this is a disadvantage, since the lower half can be negative when the operands are positive, and it is a nuisance to correct this; but actually it is a distinct *advantage* from the standpoint of random number generation, since we can take $m = 2^{32}$ instead of 2^{31} (see exercise 4).

EXERCISES

1. [*M12*] In exercise 3.2.1–3 we concluded that the best congruential generators will have the multiplier a relatively prime to m. Show that when $m = w$ in this case it is possible to compute $(aX + c) \bmod w$ in just *three* MIX instructions, rather than the four in (1), with the result appearing in register X.

2. [*16*] Write a MIX subroutine having the following characteristics:

Calling sequence:	JMP RANDM
Entry conditions:	Location XRAND contains an integer X.
Exit conditions:	$X \leftarrow \mathrm{rA} \leftarrow (aX + c) \bmod w$, $\mathrm{rX} \leftarrow 0$, overflow off.

(Thus a call on this subroutine will produce the next random number of a linear congruential sequence.)

3. [*20*] How can the constant $(w - 1)$ be specified in general in the MIX assembly language, regardless of the value of the byte size?

▶ **4.** [*21*] Discuss the calculation of linear congruential sequences with $m = 2^{32}$ on two's-complement machines such as the System/370 series.

5. [*20*] Given that m is less than the word size, and that x, y are nonnegative integers less than m, show that the difference $(x - y) \bmod m$ may be computed in just four MIX instructions, without requiring any division. What is the best code for the sum $(x + y) \bmod m$?

▶ **6.** [*20*] The previous exercise suggests that subtraction mod m is easier to perform than addition mod m. Discuss sequences generated by the rule

$$X_{n+1} = (aX_n - c) \bmod m.$$

Are these sequences essentially different from linear congruential sequences as defined in the text? Are they more suited to efficient computer calculation?

7. [*M24*] What patterns can you spot in Table 1?

▶ **8.** [*20*] Write a MIX program analogous to (2) that computes $(aX) \bmod (w - 1)$. The values 0 and $w - 1$ are to be treated as equivalent in the input and output of your program.

9. [*23*] Write a MIX program analogous to the one in exercise 8, but it should compute $(aX) \bmod (w - 2)$.

3.2.1.2. Choice of multiplier. In this section we shall show how to choose the multiplier a so as to give the *period of maximum length*. A long period is essential for any sequence that is to be used as a source of random numbers; indeed, we would hope that the period contains considerably more numbers than will ever be used in a single application. Therefore we shall concern ourselves in this section with the question of period length. The reader should keep in mind, however, that a long period is only one desirable criterion for the randomness of our sequence. For example, when $a = c = 1$, the sequence is simply $X_{n+1} = (X_n + 1) \bmod m$, and this obviously has a period of length m, yet it is anything but random. Other considerations affecting the choice of a multiplier will be given later in this chapter.

Since only m different values are possible, the period surely cannot be longer than m. Can we achieve the maximum length, m? The example above shows that it is always possible, although the choice $a = c = 1$ does not yield a desirable sequence. Let us investigate *all* possible choices of a, c, and X_0 that give a period of length m. It turns out that all such values of the parameters can be characterized very simply; when m is the product of distinct primes, only $a = 1$ will produce the full period, but when m is divisible by a high power of some prime there is considerable latitude in the choice of a. The following theorem makes it easy to tell if the maximum period is achieved.

Theorem A. *The linear congruential sequence defined by m, a, c, and X_0 has period length m if and only if*

i) *c is relatively prime to m;*

ii) *$b = a - 1$ is a multiple of p, for every prime p dividing m;*

iii) *b is a multiple of 4, if m is a multiple of 4.*

The ideas used in the proof of this theorem go back at least a hundred years. The first proof of the theorem in this particular form was given by M. Greenberger in the special case $m = 2^e$ [see *JACM* **8** (1961), 383–389], and the sufficiency of conditions (i), (ii), and (iii) in the general case was shown by Hull and Dobell [see *SIAM Review* **4** (1962), 230–254]. To prove the theorem we will first consider some auxiliary number-theoretic results that are of interest in themselves.

Lemma P. *Let p be a prime number, and let e be a positive integer, where $p^e > 2$. If*

$$x \equiv 1 \text{ (modulo } p^e), \qquad x \not\equiv 1 \text{ (modulo } p^{e+1}), \tag{1}$$

then

$$x^p \equiv 1 \text{ (modulo } p^{e+1}), \qquad x^p \not\equiv 1 \text{ (modulo } p^{e+2}). \tag{2}$$

Proof. We have $x = 1 + qp^e$ for some integer q that is not a multiple of p. By the binomial formula

$$\begin{aligned} x^p &= 1 + \binom{p}{1} qp^e + \cdots + \binom{p}{p-1} q^{p-1} p^{(p-1)e} + q^p p^{pe} \\ &= 1 + qp^{e+1}\left(1 + \frac{1}{p}\binom{p}{2} qp^e + \frac{1}{p}\binom{p}{3} q^2 p^{2e} + \cdots + \frac{1}{p}\binom{p}{p} q^{p-1} p^{(p-1)e}\right). \end{aligned}$$

The quantity in parentheses is an integer, and, in fact, every term inside the parentheses is a multiple of p except the first term. For if $1 < k < p$, the binomial coefficient $\binom{p}{k}$ is divisible by p (cf. exercise 1.2.6–10), hence

$$\frac{1}{p}\binom{p}{k} q^{k-1} p^{(k-1)e}$$

is divisible by $p^{(k-1)e}$; and the last term is $q^{p-1}p^{(p-1)e-1}$, which is divisible by p since $(p-1)e > 1$ when $p^e > 2$. So $x^p \equiv 1 + qp^{e+1}$ (modulo p^{e+2}), and this completes the proof. (*Note:* A generalization of this result appears in exercise 3.2.2–11(a).) ∎

Lemma Q. *Let the decomposition of m into prime factors be*

$$m = p_1^{e_1} \ldots p_t^{e_t}. \tag{3}$$

The length λ of the period of the linear congruential sequence determined by (X_0, a, c, m) is the least common multiple of the lengths λ_j of the periods of the linear congruential sequences $(X_0 \bmod p_j^{e_j}, a \bmod p_j^{e_j}, c \bmod p_j^{e_j}, p_j^{e_j})$, $1 \le j \le t$.

Proof. By induction on t, it suffices to prove that if m_1 and m_2 are relatively prime, the length λ of the linear congruential sequence determined by the parameters $(X_0, a, c, m_1 m_2)$ is the least common multiple of the lengths λ_1 and λ_2 of the periods of the sequences determined by $(X_0 \bmod m_1,\ a \bmod m_1,\ c \bmod m_1,\ m_1)$ and $(X_0 \bmod m_2,\ a \bmod m_2,\ c \bmod m_2,\ m_2)$. We observed in the previous section, Eq. (4), that if the elements of these three sequences are respectively denoted by X_n, Y_n, and Z_n, we will have

$$Y_n = X_n \bmod m_1 \qquad \text{and} \qquad Z_n = X_n \bmod m_2, \qquad \text{for all } n \ge 0.$$

Therefore, by Law D of Section 1.2.4, we find that

$$X_n = X_k \qquad \text{if and only if} \qquad Y_n = Y_k \quad \text{and} \quad Z_n = Z_k. \tag{4}$$

Let λ' be the least common multiple of λ_1 and λ_2; we wish to prove that $\lambda' = \lambda$. Since $X_n = X_{n+\lambda}$ for all suitably large n, we have $Y_n = Y_{n+\lambda}$ (hence λ is a multiple of λ_1) and $Z_n = Z_{n+\lambda}$ (hence λ is a multiple of λ_2), so we must have $\lambda \ge \lambda'$. Furthermore, we know that $Y_n = Y_{n+\lambda'}$ and $Z_n = Z_{n+\lambda'}$ for all suitably large n; therefore, by (4), $X_n = X_{n+\lambda'}$. This proves $\lambda \le \lambda'$. ∎

Now we are ready to prove Theorem A. Because of Lemma Q, it suffices to prove the theorem when m is a power of a prime number. For

$$p_1^{e_1} \ldots p_t^{e_t} = \lambda = \operatorname{lcm}(\lambda_1, \ldots, \lambda_t) \le \lambda_1 \ldots \lambda_t \le p_1^{e_1} \ldots p_t^{e_t}$$

can be true if and only if $\lambda_j = p_j^{e_j}$ for $1 \le j \le t$.

Therefore, assume that $m = p^e$, where p is prime and e is a positive integer. The theorem is obviously true when $a = 1$, so we may take $a > 1$. The period can be of length m if and only if each possible integer $0 \le x < m$ occurs in the period, since no value occurs in the period more than once. Therefore the period is of length m if and only if the period of the sequence with $X_0 = 0$ is of length m, and we are justified in supposing that $X_0 = 0$. By formula 3.2.1–6 we have

$$X_n = \left(\frac{a^n - 1}{a - 1}\right) c \bmod m. \tag{5}$$

If c is not relatively prime to m, this value X_n could never be equal to 1, so condition (i) of the theorem is necessary. The period has length m if and only if the smallest positive value of n for which $X_n = X_0 = 0$ is $n = m$. By (5) and condition (i), our theorem now reduces to proving the following fact:

Lemma R. *Assume that $1 < a < p^e$, where p is prime. If λ is the smallest positive integer for which $(a^\lambda - 1)/(a-1) \equiv 0$ (modulo p^e), then*

$$\lambda = p^e \quad \text{if and only if} \quad \begin{cases} a \equiv 1 \text{ (modulo } p) & \text{when } p > 2, \\ a \equiv 1 \text{ (modulo } 4) & \text{when } p = 2. \end{cases}$$

Proof. Assume that $\lambda = p^e$. If $a \not\equiv 1$ (modulo p), then $(a^n - 1)/(a-1) \equiv 0$ (modulo p^e) if and only if $a^n - 1 \equiv 0$ (modulo p^e). The condition $a^{p^e} - 1 \equiv 0$ (modulo p^e) then implies that $a^{p^e} \equiv 1$ (modulo p); but by Theorem 1.2.4F we have $a^{p^e} \equiv a$ (modulo p), hence $a \not\equiv 1$ (modulo p) leads to a contradiction. And if $p = 2$ and $a \equiv 3$ (modulo 4), we have $(a^{2^{e-1}} - 1)/(a-1) \equiv 0$ (modulo 2^e) by exercise 8. These arguments show that it is necessary in general to have $a = 1 + qp^f$, where $p^f > 2$ and q is not a multiple of p, whenever $\lambda = p^e$.

It remains to be shown that this condition is *sufficient* to make $\lambda = p^e$. By repeated application of Lemma P, we find that

$$a^{p^g} \equiv 1 \text{ (modulo } p^{f+g}), \qquad a^{p^g} \not\equiv 1 \text{ (modulo } p^{f+g+1}),$$

for all $g \geq 0$, and therefore

$$\begin{aligned} (a^{p^g} - 1)/(a-1) &\equiv 0 \text{ (modulo } p^g), \\ (a^{p^g} - 1)/(a-1) &\not\equiv 0 \text{ (modulo } p^{g+1}). \end{aligned} \tag{6}$$

In particular, $(a^{p^e} - 1)/(a-1) \equiv 0$ (modulo p^e). Now the congruential sequence $(0, a, 1, p^e)$ has $X_n = (a^n - 1)/(a-1) \bmod p^e$; therefore it has a period of length λ, that is, $X_n = 0$ if and only if n is a multiple of λ. Hence p^e is a multiple of λ. This can happen only if $\lambda = p^g$ for some g, and the relations in (6) imply that $\lambda = p^e$, completing the proof. ∎

The proof of Theorem A is now complete. ∎

We will conclude this section by considering the special case of pure multiplicative generators, when $c = 0$. Although the random number generation process is slightly faster in this case, Theorem A shows us that the maximum period length cannot be achieved. In fact, this is quite obvious, since the sequence now satisfies the relation

$$X_{n+1} = aX_n \bmod m, \tag{7}$$

and the value $X_n = 0$ should never appear, lest the sequence degenerate to zero. In general, if d is any divisor of m and if X_n is a multiple of d, all succeeding elements $X_{n+1}, X_{n+2}, \ldots$ of the multiplicative sequence will be multiples of d. So when $c = 0$, we will want X_n to be relatively prime to m for all n, and this limits the length of the period to at most $\varphi(m)$, the number of integers between 0 and m that are relatively prime to m.

It may be possible to achieve an acceptably long period even if we stipulate that $c = 0$. Let us now try to find conditions on the multiplier so that the period is as long as possible in this special case.

According to Lemma Q, the period of the sequence depends entirely on the periods of the sequences when $m = p^e$, so let us consider that situation. We have $X_n = a^n X_0 \bmod p^e$, and it is clear that the period will be of length 1 if a is a multiple of p, so we take a to be relatively prime to p. Then the period is the smallest integer λ such that $X_0 = a^\lambda X_0 \bmod p^e$. If the greatest common divisor of X_0 and p^e is p^f, this condition is equivalent to

$$a^\lambda \equiv 1 \text{ (modulo } p^{e-f}). \tag{8}$$

By Euler's theorem (exercise 1.2.4–28), $a^{\varphi(p^{e-f})} \equiv 1$ (modulo p^{e-f}); hence λ is a divisor of

$$\varphi(p^{e-f}) = p^{e-f-1}(p-1).$$

When a is relatively prime to m, the smallest integer λ for which $a^\lambda \equiv 1$ (modulo m) is conventionally called *the order of a modulo m*. Any such value of a that has the *maximum* possible order modulo m is called a *primitive element* modulo m.

Let $\lambda(m)$ denote the order of a primitive element, i.e., the maximum possible order, modulo m. The remarks above show that $\lambda(p^e)$ is a divisor of $p^{e-1}(p-1)$; with a little care (see exercises 11 through 16 below) we can give the precise value of $\lambda(m)$ in all cases as follows:

$$\begin{aligned} \lambda(2) = 1, \qquad \lambda(4) = 2, \qquad \lambda(2^e) = 2^{e-2} \quad \text{if} \quad e \geq 3. \\ \lambda(p^e) = p^{e-1}(p-1), \qquad \text{if} \quad p > 2. \\ \lambda(p_1^{e_1} \ldots p_t^{e_t}) = \operatorname{lcm}(\lambda(p_1^{e_1}), \ldots, \lambda(p_t^{e_t})). \end{aligned} \tag{9}$$

Our remarks may be summarized in the following theorem:

Theorem B. [R. D. Carmichael, *Bull. Amer. Math. Soc.* **16** (1910), 232–238.] *The maximum period possible when $c = 0$ is $\lambda(m)$, where $\lambda(m)$ is defined in* (9). *This period is achieved if*

i) *X_0 is relatively prime to m;*
ii) *a is a primitive element modulo m.* ∎

Note that we can obtain a period of length $m - 1$ if m is prime; this is just one less than the maximum length, so for all practical purposes such a period is as long as we want.

The question now is, how can we find primitive elements modulo m? The exercises at the close of this section tell us that there is a fairly simple answer when m is prime or a power of a prime, namely the results stated in our next theorem.

Theorem C. *The number a is a primitive element modulo p^e if and only if*

i) $p^e = 2$, a *is odd;* *or* $p^e = 4$, $a \bmod 4 = 3$;
or $p^e = 8$, $a \bmod 8 = 3, 5, 7$; *or* $p = 2$, $e \geq 4$, $a \bmod 8 = 3, 5$;

or

ii) p *is odd,* $e = 1$, $a \not\equiv 0$ (modulo p), *and* $a^{(p-1)/q} \not\equiv 1$ (modulo p) *for any prime divisor* q *of* $p - 1$;

or

iii) p *is odd,* $e > 1$, a *satisfies (ii), and* $a^{p-1} \not\equiv 1$ (modulo p^2). ∎

Conditions (ii) and (iii) of this theorem are readily tested on a computer for large values of p, by using the efficient methods for evaluating powers discussed in Section 4.6.3. Theorem C applies to powers of primes only; if we are given values a_j that are primitive modulo $p_j^{e_j}$, it is possible to find a single value a such that $a \equiv a_j$ (modulo $p_j^{e_j}$), for $1 \leq j \leq t$, using the "Chinese remainder algorithm" discussed in Section 4.3.2, and this number a will be a primitive element modulo $p_1^{e_1} \ldots p_t^{e_t}$. Hence there is a reasonably efficient way to construct multipliers satisfying the condition of Theorem B, for any desired value of m, although the calculations can be somewhat lengthy in the general case.

In the common case $m = 2^e$, with $e \geq 4$, the conditions above simplify to the single requirement that $a \equiv 3$ or 5 (modulo 8). In this case, one-fourth of all possible multipliers give the maximum period.

The second most common case is when $m = 10^e$. Using Lemmas P and Q, it is not difficult to obtain necessary and sufficient conditions for the achievement of the maximum period in the case of a decimal computer (cf. exercise 18):

Theorem D. *If $m = 10^e$, $e \geq 5$, $c = 0$, and X_0 is not a multiple of 2 or 5, the period of the linear congruential sequence is $5 \times 10^{e-2}$ if and only if $a \bmod 200$ equals one of the following 32 values:*

$$\begin{array}{l} 3, 11, 13, 19, 21, 27, 29, 37, 53, 59, 61, 67, 69, 77, 83, 91, 109, 117, \\ 123, 131, 133, 139, 141, 147, 163, 171, 173, 179, 181, 187, 189, 197. \end{array} \quad (10)$$

∎

EXERCISES

1. [*10*] What is the length of the period of the linear congruential sequence with $X_0 = 5772156648$, $a = 3141592621$, $c = 2718281829$, and $m = 10000000000$?

2. [*10*] Are the following two conditions sufficient to guarantee the maximum length period, when m is a power of 2? "(i) c is odd; (ii) $a \bmod 4 = 1$."

3. [*15*] Suppose that $m = 10^e$, where $e \geq 2$, and suppose further that c is odd and not a multiple of 5. Show that the linear congruential sequence will have the maximum length period if and only if $a \bmod 20 = 1$.

4. [*M20*] When a and c satisfy the conditions of Theorem A, and when $m = 2^e$, $X_0 = 0$, what is the value of $X_{2^{e-1}}$?

5. [*14*] Find all multipliers a that satisfy the conditions of Theorem A when $m = 2^{35} + 1$. (The prime factors of m may be found in Table 3.2.1.1–1.)

▶ **6.** [*20*] Find all multipliers a that satisfy the conditions of Theorem A when $m = 10^6 - 1$. (See Table 3.2.1.1–1.)

▶ **7.** [*M23*] The period of a congruential sequence need not start with X_0, but we can always find indices $\mu \geq 0$ and $\lambda > 0$ such that $X_{n+\lambda} = X_n$ whenever $n \geq \mu$, and for which μ and λ are the smallest possible values with this property. (Cf. exercises 3.1–6 and 3.2.1–1.) If μ_j and λ_j are the indices corresponding to the sequences $(X_0 \bmod p_j^{e_j}, a \bmod p_j^{e_j}, c \bmod p_j^{e_j}, p_j^{e_j})$, and if μ and λ correspond to the sequence $(X_0, a, c, p_1^{e_1} \ldots p_t^{e_t})$, Lemma Q states that λ is the least common multiple of $\lambda_1, \ldots, \lambda_t$. What is the value of μ in terms of the values of $\mu_1, \ldots, \mu_t$? What is the maximum possible value of μ obtainable by varying X_0, a, and c, when $m = p_1^{e_1} \ldots p_t^{e_t}$ is fixed?

8. [*M20*] Show that if $a \bmod 4 = 3$, we have $(a^{2^{e-1}} - 1)/(a - 1) \equiv 0$ (modulo 2^e) when $e > 1$. (Use Lemma P.)

▶ **9.** [*M22*] (W. E. Thomson.) When $c = 0$ and $m = 2^e \geq 16$, Theorems B and C say that the period has length 2^{e-2} if and only if the multiplier a satisfies $a \bmod 8 = 3$ or $a \bmod 8 = 5$. Show that every such sequence is essentially a linear congruential sequence with $m = 2^{e-2}$, having *full* period, in the following sense:

a) If $X_{n+1} = (4c + 1)X_n \bmod 2^e$, and $X_n = 4Y_n + 1$, then

$$Y_{n+1} = \big((4c + 1)Y_n + c\big) \bmod 2^{e-2}.$$

b) If $X_{n+1} = (4c - 1)X_n \bmod 2^e$, and $X_n = \big((-1)^n(4Y_n + 1)\big) \bmod 2^e$, then

$$Y_{n+1} = \big((1 - 4c)Y_n - c\big) \bmod 2^{e-2}.$$

[*Note:* In these formulas, c is an odd integer. The literature contains several statements to the effect that sequences with $c = 0$ satisfying Theorem B are somehow more random than sequences satisfying Theorem A, in spite of the fact that the period is only one-fourth as long in the case of Theorem B. This exercise refutes such statements; in essence, one gives up two bits of the word length in order to save the addition of c, when m is a power of 2.]

10. [*M21*] For what values of m is $\lambda(m) = \varphi(m)$?

▶ **11.** [*M28*] Let x be an odd integer greater than 1. (a) Show that there exists a unique integer $f > 1$ such that $x \equiv 2^f \pm 1$ (modulo 2^{f+1}). (b) Given that $1 < x < 2^e - 1$ and that f is the corresponding integer from part (a), show that the order of x modulo 2^e is 2^{e-f}. (c) In particular, this proves Theorem C(i).

12. [*M26*] Let p be an odd prime. If $e > 1$, prove that a is a primitive element modulo p^e if and only if a is a primitive element modulo p and $a^{p-1} \not\equiv 1$ (modulo p^2). (For the purposes of this exercise, assume that $\lambda(p^e) = p^{e-1}(p-1)$. This fact is proved in exercises 14 and 16 below.)

13. [*M22*] Let p be prime. Given that a is not a primitive element modulo p, show that either a is a multiple of p or $a^{(p-1)/q} \equiv 1$ (modulo p) for some prime number q that divides $p - 1$.

14. [*M18*] If $e > 1$ and p is an odd prime, and if a is a primitive element modulo p, prove that either a or $a + p$ is a primitive element modulo p^e. [*Hint:* See exercise 12.]

15. [*M29*] (a) Let a_1, a_2 be relatively prime to m, and let their orders modulo m be λ_1, λ_2, respectively. If λ is the least common multiple of λ_1 and λ_2, prove that $a_1^{\kappa_1} a_2^{\kappa_2}$ has order λ modulo m, for suitable integers κ_1, κ_2. [*Hint:* Consider first the case that λ_1 is relatively prime to λ_2.] (b) Let $\lambda(m)$ be the maximum order of any element modulo m. Prove that $\lambda(m)$ is a multiple of the order of each element modulo m; that is, prove that $a^{\lambda(m)} \equiv 1$ (modulo m) whenever a is relatively prime to m.

▶ **16.** [*M24*] Let p be a prime number. (a) Let $f(x) = x^n + c_1 x^{n-1} + \cdots + c_n$, where the c's are integers. Given that a is an integer for which $f(a) \equiv 0$ (modulo p), show that there exists a polynomial $q(x) = x^{n-1} + q_1 x^{n-2} + \cdots + q_{n-1}$ with integer coefficients such that $f(x) \equiv (x - a)q(x)$ (modulo p) for all integers x. (b) Let $f(x)$ be a polynomial as in (a). Show that $f(x)$ has at most n distinct "roots" modulo p; that is, there are at most n integers a, with $0 \le a < p$, such that $f(a) \equiv 0$ (modulo p). (c) Because of exercise 15(b), the polynomial $f(x) = x^{\lambda(p)} - 1$ has $p - 1$ distinct roots; hence there is an integer a with order $p - 1$.

17. [*M26*] Not all of the values listed in Theorem D would be found by the text's construction; for example, 11 is not primitive modulo 5^e. How can this be possible, when 11 *is* primitive modulo 10^e, according to Theorem D? Which of the values listed in Theorem D are primitive elements modulo *both* 2^e and 5^e?

18. [*M25*] Prove Theorem D. (Cf. the previous exercise.)

19. [*40*] Make a table of some suitable multipliers, a, for each of the values of m listed in Table 3.2.1.1–1, assuming that $c = 0$.

▶ **20.** [*M24*] (G. Marsaglia.) The purpose of this exercise is to study the period length of an *arbitrary* linear congruential sequence. Let $Y_n = 1 + a + \cdots + a^{n-1}$, so that $X_n = (AY_n + X_0) \bmod m$ for some constant A by Eq. 3.2.1–8. (a) Prove that the period length of $\langle X_n \rangle$ is the period length of $\langle Y_n \bmod m' \rangle$, where $m' = m/\gcd(A, m)$. (b) Prove that the period length of $\langle Y_n \bmod p^e \rangle$ satisfies the following when p is prime: (i) If $a \bmod p = 0$, it is 1. (ii) If $a \bmod p = 1$, it is p^e, except when $p = 2$ and $e \ge 2$ and $a \bmod 4 = 3$. (iii) If $p = 2$, $e \ge 2$, and $a \bmod 4 = 3$, it is twice the order of a modulo p^e (cf. exercise 11), unless $a \equiv -1$ (modulo 2^e) when it is 2. (iv) If $a \bmod p > 1$, it is the order of a modulo p^e.

21. [*M25*] In a linear congruential sequence of maximum period, let $X_0 = 0$ and let s be the least positive integer such that $a^s \equiv 1$ (modulo m). Prove that $\gcd(X_s, m) = s$.

3.2.1.3. Potency. In the preceding section, we showed that the maximum period can be obtained when $b = a - 1$ is a multiple of each prime dividing m; and b must also be a multiple of 4 if m is a multiple of 4. If z is the radix of the machine being used—so that $z = 2$ for a binary computer, and $z = 10$ for a decimal computer—and if m is the word size z^e, the multiplier

$$a = z^k + 1, \qquad 2 \le k < e \tag{1}$$

satisfies these conditions. Theorem 3.2.1.2A also says that we may take $c = 1$. The recurrence relation now has the form

$$X_{n+1} = \big((z^k + 1)X_n + 1\big) \bmod z^e, \tag{2}$$

and this equation suggests that we can avoid the multiplication; merely shifting and adding will suffice.

For example, suppose that $a = B^2 + 1$, where B is the byte size of MIX. The code

```
LDA  X
SLA  2
ADD  X
INCA 1
```
(3)

can be used in place of the instructions given in Section 3.2.1.1, and the execution time decreases from $16u$ to $7u$.

For this reason, multipliers having form (1) have been widely discussed in the literature, and indeed they have been recommended by many authors. However, the early years of experimentation with this method showed that *multipliers having the simple form in* (1) *should be avoided.* The generated numbers just aren't random enough.

Later in this chapter we shall be discussing some rather sophisticated theory that accounts for the badness of all the linear congruential random number generators known to be bad. However, some generators (such as (2)) are sufficiently awful that a comparatively simple theory can be used to dispense with them. This simple theory is related to the concept of "potency," which we shall now discuss.

The *potency* of a linear congruential sequence with maximum period is defined to be the least integer s such that

$$b^s \equiv 0 \text{ (modulo } m). \tag{4}$$

(Such an integer s will always exist when the multiplier satisfies the conditions of Theorem 3.2.1.2A, since b is a multiple of every prime dividing m.)

We may analyze the randomness of the sequence by taking $X_0 = 0$, since 0 occurs somewhere in the period. With this assumption, we have

$$X_n = \big((a^n - 1)c/b\big) \bmod m,$$

and if we expand $a^n - 1 = (b+1)^n - 1$ by the binomial theorem, we find that

$$X_n = c\left(n + \binom{n}{2}b + \cdots + \binom{n}{s}b^{s-1}\right) \bmod m. \tag{5}$$

All terms in b^s, b^{s+1}, etc., may be ignored, since they are multiples of m.

Equation (5) can be instructive, so we shall consider some special cases. If $a = 1$, the potency is 1; and $X_n \equiv cn$ (modulo m), as we have already observed, so the sequence is surely not random. If the potency is 2, we have $X_n \equiv cn + cb\binom{n}{2}$, and again the sequence is not very random; indeed,

$$X_{n+1} - X_n \equiv c + cbn$$

in this case, so the differences between consecutively generated numbers change in a simple way from one value of n to the next. The point (X_n, X_{n+1}, X_{n+2}) always lies on one of the four planes

$$x - 2y + z = d + m, \qquad x - 2y + z = d - m,$$
$$x - 2y + z = d, \qquad x - 2y + z = d - 2m,$$

in three-dimensional space, where $d = cb \bmod m$.

If the potency is 3, the sequence begins to look somewhat more random, but there is a high degree of dependency between X_n, X_{n+1}, and X_{n+2}; tests show that sequences with potency 3 are still not sufficiently good. Reasonable results have been reported when the potency is 4 or more, but these have been disputed by other people. A potency of at least 5 would seem to be required for sufficiently random values.

Suppose, for example, that $m = 2^{35}$ and $a = 2^k + 1$. Then $b = 2^k$, so we find that when $k \geq 18$, the value $b^2 = 2^{2k}$ is a multiple of m: the potency is 2. If $k = 17, 16, \ldots, 12$, the potency is 3, and a potency of 4 is achieved for $k = 11, 10, 9$. The only acceptable multipliers, from the standpoint of potency, therefore have $k \leq 8$. This means $a \leq 257$, and we shall see later that *small* multipliers are also to be avoided. We have now eliminated all multipliers of the form $2^k + 1$ when $m = 2^{35}$.

When m is equal to $w \pm 1$, where w is the word size, m is generally not divisible by high powers of primes, and a high potency is impossible (see exercise 6). So in this case, the maximum-period method should *not* be used; the pure-multiplication method with $c = 0$ should be applied instead.

It must be emphasized that high potency is necessary but not sufficient for randomness; we use the concept of potency only to reject impotent generators, not to accept the potent ones. Linear congruential sequences should pass the "spectral test" discussed in Section 3.3.4 before they are considered to be acceptably random.

EXERCISES

1. [*M10*] Show that, no matter what the byte size B of MIX happens to be, the code (3) yields a random number generator of maximum period.

2. [*10*] What is the potency of the generator represented by the MIX code (3)?

3. [*11*] When $m = 2^{35}$, what is the potency of the linear congruential sequence with $a = 3141592621$? What is the potency if the multiplier is $a = 2^{23} + 2^{14} + 2^2 + 1$?

4. [*15*] Show that if $m = 2^e \geq 8$, maximum potency is achieved when $a \bmod 8 = 5$.

5. [*M20*] Given that $m = p_1^{e_1} \ldots p_t^{e_t}$ and $a = 1 + kp_1^{f_1} \ldots p_t^{f_t}$, where a satisfies the conditions of Theorem 3.2.1.2A and k is relatively prime to m, show that the potency is $\max(\lceil e_1/f_1 \rceil, \ldots, \lceil e_t/f_t \rceil)$.

▶ **6.** [*20*] Which of the values of $m = w \pm 1$ in Table 3.2.1.1–1 can be used in a linear congruential sequence of maximum period whose potency is 4 or more? (Use the result of exercise 5.)

7. [*M20*] When a satisfies the conditions of Theorem 3.2.1.2A, it is relatively prime to m; hence there is a number a' such that $aa' \equiv 1$ (modulo m). Show that a' can be expressed simply in terms of b.

▶ **8.** [*M26*] A random number generator defined by $X_{n+1} = (2^{17} + 3)X_n \bmod 2^{35}$ and $X_0 = 1$ was subjected to the following test: Let $Y_n = \lfloor 10X_n/2^{35} \rfloor$; then Y_n should be a random digit between 0 and 9, and the triples $(Y_{3n}, Y_{3n+1}, Y_{3n+2})$ should take on each of the 1000 possible values from (0, 0, 0) to (9, 9, 9) with equal probability. But with 30000 values of n tested, some triples hardly ever occurred, and others occurred much more often than they should have. Can you account for this failure?

3.2.2. Other Methods

Of course, linear congruential sequences are not the only sources of random numbers that have been proposed for computer use. In this section we shall review the most significant alternatives; some of these methods are quite important, while others are interesting chiefly because they are not as good as a person might expect.

One of the common fallacies encountered in connection with random number generation is the idea that we can take a good generator and modify it a little, in order to get an "even-more-random" sequence. This is often false. For example, we know that

$$X_{n+1} = (aX_n + c) \bmod m \tag{1}$$

leads to reasonably good random numbers; wouldn't the sequence produced by

$$X_{n+1} = \big((aX_n) \bmod (m + 1) + c\big) \bmod m \tag{2}$$

be even *more* random? The answer is, the new sequence is probably a great deal *less* random. For the whole theory breaks down, and in the absence of any theory about the behavior of the sequence (2), we come into the area of generators of the type $X_{n+1} = f(X_n)$ with the function f chosen at random; exercises 3.1–11 through 3.1–15 show that these sequences probably behave much more poorly than the sequences obtained from the more disciplined function (1).

Let us consider another approach, in an attempt to get "more random" numbers. The linear congruential method can be generalized to, say, a quadratic congruential method:

$$X_{n+1} = (dX_n^2 + aX_n + c) \bmod m. \tag{3}$$

Exercise 8 generalizes Theorem 3.2.1.2A to obtain necessary and sufficient conditions on a, c, and d such that the sequence defined by (3) has a period of the maximum length m; the restrictions are not much more severe than in the linear method.

An interesting quadratic method has been proposed by R. R. Coveyou when m is a power of two; let

$$X_0 \bmod 4 = 2, \qquad X_{n+1} = X_n(X_n + 1) \bmod 2^e, \qquad n \geq 0. \tag{4}$$

This sequence can be computed with about the same efficiency as (1), without any worries of overflow. It has an interesting connection with von Neumann's original middle-square method: If we let Y_n be $2^e X_n$, so that Y_n is a double-precision number obtained by placing e zeros to the right of the binary representation of X_n, then Y_{n+1} consists of precisely the middle $2e$ digits of $Y_n^2 + 2^e Y_n$! In other words, Coveyou's method is almost identical to a somewhat degenerate double-precision middle-square method, yet it is guaranteed to have a long period; further evidence of its randomness is proved in exercise 3.3.4–25.

Other generalizations of Eq. (1) also suggest themselves; for example, we might try to extend the period length of the sequence. The period of a linear congruential sequence is extremely long; when m is approximately the word size of the computer, we usually get periods on the order of 10^9 or more, so that typical calculations will use only a very small portion of the sequence. On the other hand, when we discuss the idea of "accuracy" in Section 3.3.4 we will see that the period length influences the degree of randomness achievable in a sequence. Therefore it is occasionally desirable to seek a longer period, and several methods are available for this purpose. One technique is to make X_{n+1} depend on both X_n and X_{n-1}, instead of just on X_n; then the period length can be as high as m^2, since the sequence will not begin to repeat until we have $(X_{n+\lambda}, X_{n+\lambda+1}) = (X_n, X_{n+1})$.

The simplest sequence in which X_{n+1} depends on more than one of the preceding values is the Fibonacci sequence,

$$X_{n+1} = (X_n + X_{n-1}) \bmod m. \tag{5}$$

This generator was considered in the early 1950s, and it usually gives a period length greater than m; but tests have shown that the numbers produced by the Fibonacci recurrence (5) are definitely *not* satisfactorily random, and so at the present time the main interest in (5) as a source of random numbers is that it makes a nice "bad example." We may also consider generators of the form

$$X_{n+1} = (X_n + X_{n-k}) \bmod m, \tag{6}$$

when k is a comparatively large value. These were introduced by Green, Smith, and Klem [*JACM* **6** (1959), 527–537], who reported that, when $k \leq 15$, the sequence fails to pass the "gap test" described in Section 3.3.2, although when $k = 16$ the test was satisfactory.

A much better type of additive generator was devised in 1958 by G. J. Mitchell and D. P. Moore [unpublished], who suggested the somewhat unusual sequence defined by

$$X_n = (X_{n-24} + X_{n-55}) \bmod m, \qquad n \geq 55, \tag{7}$$

where m is even, and where $X_0, \ldots, X_{54}$ are arbitrary integers not all even. The constants 24 and 55 in this definition were not chosen at random, they are special values that happen to have the property that the least significant bits $\langle X_n \bmod 2 \rangle$ will have a period of length $2^{55} - 1$. Therefore the sequence $\langle X_n \rangle$ must have a period at least this long. Exercise 11, which explains how to calculate the period length of such sequences, proves that (7) has a period of length $2^f(2^{55} - 1)$ for some f, $0 \le f < e$, when $m = 2^e$.

At first glance Eq. (7) may not seem to be extremely well suited to machine implementation, but in fact there is a very efficient way to generate the sequence using a cyclic list:

Algorithm A (*Additive number generator*). Memory cells $Y[1]$, $Y[2]$, ..., $Y[55]$ are initially set to the values X_{54}, X_{53}, ..., X_0, respectively; j is initially equal to 24 and k is 55. Successive performances of this algorithm will produce the numbers X_{55}, X_{56}, ... as output.

A1. [Add.] (If we are about to output X_n at this point, $Y[j]$ now equals X_{n-24} and $Y[k]$ equals X_{n-55}.) Set $Y[k] \leftarrow (Y[k] + Y[j]) \bmod 2^e$, and output $Y[k]$.

A2. [Advance.] Decrease j and k by 1. If now $j = 0$, set $j \leftarrow 55$; otherwise if $k = 0$, set $k \leftarrow 55$. ∎

This algorithm in MIX is simply the following:

Program A (*Additive number generator*). Assuming that index registers 5 and 6 are not touched by the remainder of the program in which this routine is embedded, the following code performs Algorithm A and leaves the result in register A. rI5 ≡ j, rI6 ≡ k.

```
LDA   Y,6   A1. Add.
ADD   Y,5   Y_k + Y_j (overflow possible)
STA   Y,6       → Y_k.
DEC5  1     A2. Advance. j ← j − 1.
DEC6  1     k ← k − 1.
J5P   *+2
ENT5  55    If j = 0, set j ← 55.
J6P   *+2
ENT6  55    If k = 0, set k ← 55.  ∎
```

This generator is usually faster than the previous methods we have been discussing, since it does not require any multiplication. Besides its speed, it has the longest period we have seen yet; and it has consistently produced reliable results, in extensive tests since its invention in 1958. Furthermore, as Richard Brent has observed, it can be made to work correctly with floating point numbers, avoiding the need to convert between integers and fractions (cf. exercise 23). Therefore it may well prove to be the very *best* source of random numbers for practical purposes. The only reason it is difficult to recommend sequence (7) wholeheartedly is that there is still very little theory to prove that it does or does not have desirable randomness properties; essentially all we know for sure

Table 1

SUBSCRIPT PAIRS YIELDING LONG PERIODS MOD 2

(1, 2)	(1, 15)	(5, 23)	(7, 31)	(5, 47)	(21, 52)	(18, 65)	(28, 73)	(2, 93)
(1, 3)	(4, 15)	(9, 23)	(13, 31)	(14, 47)	(24, 55)	(32, 65)	(31, 73)	(21, 94)
(1, 4)	(7, 15)	(3, 25)	(13, 33)	(20, 47)	(7, 57)	(9, 68)	(9, 79)	(11, 95)
(2, 5)	(3, 17)	(7, 25)	(2, 35)	(21, 47)	(22, 57)	(33, 68)	(19, 79)	(17, 95)
(1, 6)	(5, 17)	(3, 28)	(11, 36)	(9, 49)	(19, 58)	(6, 71)	(4, 81)	(6, 97)
(1, 7)	(6, 17)	(9, 28)	(4, 39)	(12, 49)	(1, 60)	(9, 71)	(16, 81)	(12, 97)
(3, 7)	(7, 18)	(13, 28)	(8, 39)	(15, 49)	(11, 60)	(18, 71)	(35, 81)	(33, 97)
(4, 9)	(3, 20)	(2, 29)	(14, 39)	(22, 49)	(1, 63)	(20, 71)	(13, 84)	(34, 97)
(3, 10)	(2, 21)	(3, 31)	(3, 41)	(3, 52)	(5, 63)	(35, 71)	(13, 87)	(11, 98)
(2, 11)	(1, 22)	(6, 31)	(20, 41)	(19, 52)	(31, 63)	(25, 73)	(38, 89)	(27, 98)

For each pair (l, k), the pair $(k-l, k)$ is also valid (see exercise 24), hence only values of $l \leq k/2$ are listed here. For extensions of this table, see N. Zierler and J. Brillhart, *Information and Control* **13** (1968), 541–554; **14** (1969), 566–569; **15** (1969), 67–69.

is that the period is very long, and this is not enough. John Reiser (Ph. D. thesis, Stanford Univ., 1977) has shown, however, that an additive sequence like (7) will be well distributed in high dimensions, provided that a certain plausible conjecture is true (cf. exercise 26).

The fact that the special numbers (24, 55) in (7) work so well follows from theoretical results developed in some of the exercises below. Table 1 lists all pairs (l, k) for which the sequence $X_n = (X_{n-l}+X_{n-k}) \bmod 2$ has period length 2^k-1, when $k < 100$. The pairs (l, k) for small as well as larger k are shown, for the sake of completeness; the pair (1, 2) corresponds to the Fibonacci sequence mod 2, whose period has length 3. However, only pairs (l, k) for relatively large k should be used to generate random numbers in practice.

Instead of considering only additive sequences, we can construct useful random number generators by taking general linear combinations of $X_{n-1}, \ldots, X_{n-k}$ for small k. In this case the best results occur when the modulus m is a large *prime*; for example, m can be chosen to be the largest prime number that fits in a single computer word (see Table 4.5.4–1). When $m = p$ is prime, the theory of finite fields tells us that it is possible to find multipliers $a_1, \ldots, a_k$ such that the sequence defined by

$$X_n = (a_1X_{n-1} + \cdots + a_kX_{n-k}) \bmod p \tag{8}$$

has period length $p^k - 1$; here $X_0, \ldots, X_{k-1}$ may be chosen arbitrarily but not all zero. (The special case $k = 1$ corresponds to a multiplicative congruential sequence with prime modulus, with which we are already familiar.) The constants $a_1, \ldots, a_k$ in (8) have the desired property if and only if the polynomial

$$f(x) = x^k - a_1x^{k-1} - \cdots - a_k \tag{9}$$

is a "primitive polynomial modulo p," that is, if and only if this polynomial has a root that is a primitive element of the field with p^k elements (see exercise 4.6.2–16).

Of course, the mere fact that suitable constants $a_1, \ldots, a_k$ *exist* giving a period of length $p^k - 1$ is not enough for practical purposes; we must be able to *find* them, and we can't simply try all p^k possibilities, since p is on the order of the computer's word size. Fortunately there are exactly $\varphi(p^k - 1)/k$ suitable choices of $(a_1, \ldots, a_k)$, so there is a fairly good chance of hitting one after making a few random tries. But we also need a way to tell quickly whether or not (9) is a primitive polynomial modulo p; it is certainly unthinkable to generate up to $p^k - 1$ elements of the sequence and wait for a repetition! Methods of testing for primitivity modulo p are discussed by Alanen and Knuth in *Sankhyā* (A) **26** (1964), 305–328; the following criteria can be used: Let $r = (p^k - 1)/(p - 1)$.

i) $(-1)^{k-1}a_k$ must be a primitive root modulo p. (Cf. Section 3.2.1.2.)
ii) The polynomial x^r must be congruent to $(-1)^{k-1}a_k$, modulo $f(x)$ and p.
iii) The degree of $x^{r/q} \bmod f(x)$, using polynomial arithmetic modulo p, must be positive, for each prime divisor q of r.

Efficient ways to compute the polynomial $x^n \bmod f(x)$, using polynomial arithmetic modulo a given prime p, are discussed in Section 4.6.2.

In order to carry out this test, we need to know the prime factorization of $r = (p^k - 1)/(p - 1)$, and this is the limiting factor in the calculation; r can be factored in a reasonable amount of time when $k = 2$, 3, and perhaps 4, but higher values of k are difficult to handle when p is large. Even $k = 2$ essentially doubles the number of "significant random digits" over what is achievable with $k = 1$, so larger values of k will rarely be necessary.

An adaptation of the spectral test (Section 3.3.4) can be used to rate the sequence of numbers generated by (8); see exercise 3.3.4–26. The considerations of that section show that we should *not* make the obvious choice of $a_1 = +1$ or -1 when it is possible to do so; it is better to pick large, essentially "random," values of $a_1, \ldots, a_k$ that satisfy the conditions, and to verify the choice by applying the spectral test. A significant amount of computation is involved in finding $a_1, \ldots, a_k$, but all known evidence indicates that the result will be a very satisfactory source of random numbers. We essentially achieve the randomness of a linear congruential generator with k-tuple precision, using only single precision operations.

The special case $p = 2$ is of independent interest. Sometimes a random number generator is desired that merely produces a random sequence of *bits*—zeros and ones—instead of fractions between zero and one. There is a simple way to generate a highly random bit sequence on a binary computer, manipulating k-bit words: Start with an arbitrary nonzero binary word X. To get the next random bit of the sequence, do the following operations, shown in MIX's language (see exercise 16):

```
LDA  X     (Assume that overflow is now "off.")
ADD  X     Shift left one bit.
JNOV *+2   Jump if high bit was originally zero.          (10)
XOR  A     Otherwise adjust number with "exclusive or."
STA  X                 ∎
```

The fourth instruction here is the "exclusive or" operation found on nearly all binary computers (cf. exercise 2.5–28 and Section 7.1); it changes each bit position in which location A has a "1" bit. The value in location A is the binary constant $(a_1 \ldots a_k)_2$, where $x^k - a_1 x^{k-1} - \cdots - a_k$ is a primitive polynomial modulo 2 as above. After the code (10) has been executed, the next bit of the generated sequence may be taken as the least significant bit of word X (or, alternatively, we could consistently use the most significant bit of X, if it were more convenient to do so).

For example, consider Fig. 1, which illustrates the sequence generated for $k = 4$ and CONTENTS(A) $= (0011)_2$. This is, of course, an unusually small value for k. The right-hand column shows the sequence of bits of the sequence, namely 1101011110001001..., repeating in a period of length $2^k - 1 = 15$. This sequence is quite random, considering that it was generated with only four bits of memory; to see this, consider the adjacent sets of four bits occurring in the period, namely 1101, 1010, 0101, 1011, 0111, 1111, 1110, 1100, 1000, 0001, 0010, 0100, 1001, 0011, 0110. In general, every possible adjacent set of k bits occurs exactly once in the period, except the set of all zeros, since the period length is $2^k - 1$; thus, adjacent sets of k bits are essentially independent. We shall see in Section 3.5 that this is a very strong criterion for randomness when k is, say, 30 or more. Theoretical results illustrating the randomness of this sequence are given in an article by R. C. Tausworthe, *Math. Comp.* **19** (1965), 201–209.

Primitive polynomials modulo 2 of degree ≤ 100 have been tabulated by E. J. Watson, *Math. Comp.* **16** (1962), 368–369. When $k = 35$, we may take

$$\text{CONTENTS(A)} = (00000000000000000000000000000101)_2,$$

but the considerations of exercises 18 and 3.3.4–26 imply that it would be better to find "random" constants that define primitive polynomials modulo 2.

Caution: Several people have been trapped into believing that this random bit-generation technique can be used to generate random whole-word fractions $(.X_0X_1 \ldots X_{k-1})_2, (.X_kX_{k+1} \ldots X_{2k-1})_2, \ldots$; but it is actually a poor source of random fractions, even though the bits are individually quite random. Exercise 18 explains why.

Mitchell and Moore's additive generator (7) is essentially based on the concept of primitive polynomials; the polynomial $x^{55} + x^{24} + 1$ is primitive, and Table 1 is essentially a listing of all the primitive trinomials modulo 2. A generator almost identical to that of Mitchell and Moore was independently discovered in 1971 by T. G. Lewis and W. H. Payne [cf. *JACM* **20** (1973), 456–468], but using "exclusive or" instead of addition so that the period is exactly $2^{55} - 1$; each bit position in their generated numbers runs through the same periodic sequence, but has its own starting point. (See Bright and Enison, *Computing Surveys* **11** (1979), 357–370, for further discussion of Lewis and Payne's method.)

We have now seen that sequences with $0 \leq X_n < m$ and period $m^k - 1$ can be found, when X_n is a suitable function of $X_{n-1}, \ldots, X_{n-k}$ and when m is prime. The highest conceivable period for *any* sequence defined by a relation

1011
0101
1010
0111
1110
1111
1101
1001
0001
0010
0100
1000
0011
0110
1100
1011

Fig. 1. Successive contents of the computer word X in the binary method, assuming that $k = 4$ and CONTENTS(A) $= (0011)_2$.

of the form

$$X_n = f(X_{n-1}, \ldots, X_{n-k}), \qquad 0 \le X_n < m, \tag{11}$$

is easily seen to be m^k. M. H. Martin [*Bull. Amer. Math. Soc.* **40** (1934), 859–864] was the first person to show that functions achieving this maximum period are possible for all m and k; his method is easy to state, but it is unfortunately not suitable for programming (see exercise 17). From a computational standpoint, the simplest known functions f that yield the maximum period m^k appear in exercise 21; the corresponding programs are, in general, not as efficient for random number generation as other methods we have described, but they do give demonstrable randomness when the period as a whole is considered.

Another important class of techniques deals with the *combination* of random number generators, to get "more random" sequences. There will always be people who feel that the linear congruential methods, additive methods, etc., are all too simple to give sufficiently random sequences; and it may never be possible to *prove* that their skepticism is unjustified (although we believe it is), so it is pretty useless to argue the point. There are reasonably efficient methods for combining two sequences into a third one that should be haphazard enough to satisfy all but the most hardened skeptic.

Suppose we have two sequences $X_0, X_1, \ldots$ and $Y_0, Y_1, \ldots$ of random numbers between 0 and $m - 1$, preferably generated by two unrelated methods. One suggestion has been to add them together, mod m, obtaining the sequence $Z_n = (X_n + Y_n) \bmod m$; in this case, the period will be quite long if the period lengths of $\langle X_n \rangle$ and $\langle Y_n \rangle$ are relatively prime to each other (see exercise 13). Another approach, based on circular shifting and "exclusive or-ing", has been suggested by W. J. Westlake, *JACM* **14** (1967), 337–340.

A considerably different method has been suggested by M. D. MacLaren and G. Marsaglia [*JACM* **12** (1965), 83–89; *CACM* **11** (1968), 759], who use one random sequence to permute the elements of another:

Algorithm M (*Randomizing by shuffling*). Given methods for generating two sequences $\langle X_n \rangle$ and $\langle Y_n \rangle$, this algorithm will successively output the terms of a "considerably more random" sequence. We use an auxiliary table $V[0]$, $V[1]$, $\ldots$, $V[k-1]$, where k is some number chosen for convenience, usually in the neighborhood of 100. Initially, the V-table is filled with the first k values of the X-sequence.

M1. [Generate X, Y.] Set X and Y equal to the next members of the sequences $\langle X_n \rangle$ and $\langle Y_n \rangle$, respectively.

M2. [Extract j.] Set $j \leftarrow \lfloor kY/m \rfloor$, where m is the modulus used in the sequence $\langle Y_n \rangle$; that is, j is a random value, $0 \le j < k$, determined by Y.

M3. [Exchange.] Output $V[j]$ and then set $V[j] \leftarrow X$. ∎

As an example, assume that Algorithm M is applied to the following two sequences, with $k = 64$:

$$\begin{aligned} X_0 &= 5772156649, & X_{n+1} &= (3141592653X_n + 2718281829) \bmod 2^{35}; \\ Y_0 &= 1781072418, & Y_{n+1} &= (2718281829Y_n + 3141592653) \bmod 2^{35}. \end{aligned} \tag{12}$$

On intuitive grounds it appears safe to predict that the sequence obtained by applying Algorithm M will satisfy virtually *anyone's* requirements for randomness in a computer-generated sequence, because the relationship between nearby terms of the output has been almost entirely obliterated. Furthermore, the time required to generate this sequence is only slightly more than twice as long as it takes to generate the sequence $\langle X_n \rangle$ alone.

Exercise 15 proves that the period length of Algorithm M's output will be the least common multiple of the period lengths of $\langle X_n \rangle$ and $\langle Y_n \rangle$, in most situations of practical interest. In particular, the above example will have a period of length 2^{35}.

However, there is an even better way to shuffle the elements of a sequence, discovered by Carter Bays and S. D. Durham [*ACM Trans. Math. Software* **2** (1976), 59-64]. Their approach, although it appears to be superficially similar to Algorithm M, can give surprisingly better performance even though it requires only one input sequence $\langle X_n \rangle$ instead of two:

Algorithm B (*Randomizing by shuffling*). Given a method for generating a sequence $\langle X_n \rangle$, this algorithm will successively output the terms of a "considerably more random" sequence, using an auxiliary table $V[0]$, $V[1]$, $\ldots$, $V[k-1]$ as in Algorithm M. Initially the V-table is filled with the first k values of the X-sequence, and an auxiliary variable Y is set equal to the $(k+1)$st value.

B1. [Extract j.] Set $j \leftarrow \lfloor kY/m \rfloor$, where m is the modulus used in the sequence $\langle X_n \rangle$; that is, j is a random value, $0 \le j < k$, determined by Y.

B2. [Exchange.] Set $Y \leftarrow V[j]$, output Y, and then set $V[j]$ to the next member of the sequence $\langle X_n \rangle$. ∎

The reader is urged to work exercise 3, in order to get a feeling for the difference between Algorithms M and B.

On MIX we may implement Algorithm B by taking k equal to the byte size, obtaining the following simple generation scheme once the initialization has been done:

```
LD6  Y(1:1)   j ← high-order byte of Y.
LDA  X        rA ← X_n.
INCA 1        (cf. exercise 3.2.1.1-1)
MUL  A        rX ← X_{n+1}.
STX  X        "n ← n + 1."
LDA  V,6
STA  Y        Y ← V[j].
STX  V,6      V[j] ← X_n.  ∎
```
(13)

The output appears in register A. Note that Algorithm B requires only four instructions of overhead per generated number.

F. Gebhardt [*Math. Comp.* **21** (1967), 708–709] found that satisfactory random sequences were produced by Algorithm M even when it was applied to a sequence as nonrandom as the Fibonacci sequence, with $X_n = F_{2n} \bmod m$ and $Y_n = F_{2n+1} \bmod m$. However, it is also possible for Algorithm M to produce a sequence *less* random than the original sequences, if $\langle X_n \rangle$ and $\langle Y_n \rangle$ are strongly related, as shown in exercise 3. Such problems do not seem to arise with Algorithm B. Since Algorithm B won't make a sequence any less random, and since it probably enhances the randomness substantially with very little extra cost, it can be recommended for use in combination with any other random number generator.

EXERCISES

▸ **1.** [*12*] In practice, we form random numbers using $X_{n+1} = (aX_n + c) \bmod m$, where the X's are *integers*, afterwards treating them as the *fractions* $U_n = X_n/m$. The recurrence relation for U_n is actually

$$U_{n+1} = (aU_n + c/m) \bmod 1.$$

Discuss the generation of random sequences using this relation *directly*, by making use of floating point arithmetic on the computer.

▸ **2.** [*M20*] A good source of random numbers will have $X_{n-1} < X_{n+1} < X_n$ about one-sixth of the time, since each of the six possible relative orders of X_{n-1}, X_n, and X_{n+1} should be equally probable. However, show that the above ordering *never* occurs if the Fibonacci sequence (5) is used.

3. [*23*] (a) What sequence comes from Algorithm M if

$$X_0 = 0, \quad X_{n+1} = (5X_n + 3) \bmod 8, \qquad Y_0 = 0, \quad Y_{n+1} = (5Y_n + 1) \bmod 8,$$

and $k = 4$? (Note that the potency is two, so $\langle X_n \rangle$ and $\langle Y_n \rangle$ aren't extremely random to start with.) (b) What happens if Algorithm B is applied to this same sequence $\langle X_n \rangle$ with $k = 4$?

4. [*00*] Why is the most significant byte used in the first line of program (13), instead of some other byte?

▸ **5.** [*20*] Discuss using $X_n = Y_n$ in Algorithm M, in order to improve the speed of generation. Is the result analogous to Algorithm B?

6. [*10*] In the binary method (10), the text states that the low-order bit of X is random, if the code is performed repeatedly. Why isn't the entire *word* X random?

7. [*20*] Show that the full sequence of length 2^e (i.e., a sequence in which each of the 2^e possible sets of e adjacent bits occurs just once in the period) may be obtained if program (10) is changed to the following:

```
LDA  X
JANZ *+2
LDA  A
ADD  X
JNOV *+3
JAZ  *+2
XOR  A
STA  X
```

8. [*M39*] Prove that the quadratic congruential sequence (3) has period length m if and only if the following conditions are satisfied:

i) *c is relatively prime to m;*
ii) *d and $a - 1$ are both multiples of p, for all odd primes p dividing m;*
iii) *d is even, and $d \equiv a - 1$ (modulo 4), if m is a multiple of 4;*
$d \equiv a - 1$ (modulo 2), if m is a multiple of 2;
iv) *either $d \equiv 0$ or both $a \equiv 1$ and $cd \equiv 6$ (modulo 9), if m is a multiple of 9.*

[*Hint:* The sequence defined by $X_0 = 0$, $X_{n+1} = dX_n^2 + aX_n + c$ has a period of length m, modulo m, only if its period length is r modulo any divisor r of m.]

▸ **9.** [*M24*] (R. R. Coveyou.) Use the result of exercise 8 to prove that the modified middle-square method (4) has a period of length 2^{e-2}.

10. [*M29*] Show that if X_0 and X_1 are not both even and if $m = 2^e$, the period of the Fibonacci sequence (5) is $3 \cdot 2^{e-1}$.

11. [*M36*] The purpose of this exercise is to analyze certain properties of integer sequences satisfying the recurrence relation

$$X_n = a_1 X_{n-1} + \cdots + a_k X_{n-k}, \qquad n \geq k;$$

if we can calculate the period length of this sequence modulo $m = p^e$, when p is prime, the period length with respect to an arbitrary modulus m is the least common multiple of the period lengths for the prime power factors of m.

a) If $f(z)$, $a(z)$, $b(z)$ are polynomials with integer coefficients, let us write $a(z) \equiv b(z)$ (modulo $f(z)$ and m) if $a(z) = b(z) + f(z)u(z) + mv(z)$ for some polynomials $u(z)$, $v(z)$ with integer coefficients. Prove that when $f(0) = 1$ and $p^e > 2$, "If $z^\lambda \equiv 1$ (modulo $f(z)$ and p^e), $z^\lambda \not\equiv 1$ (modulo $f(z)$ and p^{e+1}), then $z^{p\lambda} \equiv 1$ (modulo $f(z)$ and p^{e+1}), $z^{p\lambda} \not\equiv 1$ (modulo $f(z)$ and p^{e+2})."

b) Let $f(z) = 1 - a_1 z - \cdots - a_k z^k$, and let

$$G(z) = 1/f(z) = A_0 + A_1 z + A_2 z^2 + \cdots.$$

Let $\lambda(m)$ denote the period length of $\langle A_n \bmod m\rangle$. Prove that $\lambda(m)$ is the smallest positive integer λ such that $z^\lambda \equiv 1$ (modulo $f(z)$ and m).

c) Given that p is prime, $p^e > 2$, and $\lambda(p^e) \neq \lambda(p^{e+1})$, prove that $\lambda(p^{e+r}) = p^r\lambda(p^e)$ for all $r \geq 0$. (Thus, to find the period length of the sequence $\langle A_n \bmod 2^e\rangle$, we can compute $\lambda(4)$, $\lambda(8)$, $\lambda(16)$, ... until we find the smallest $e \geq 3$ such that $\lambda(2^e) \neq \lambda(4)$; then the period length is determined mod 2^e for all e. Exercise 4.6.3–26 explains how to calculate X_n for large n in $O(\log n)$ operations.)

d) Show that any sequence of integers satisfying the recurrence stated at the beginning of this exercise has the generating function $g(z)/f(z)$, for some polynomial $g(z)$ with integer coefficients.

e) Given that the polynomials $f(z)$ and $g(z)$ in part (d) are relatively prime modulo p (cf. Section 4.6.1), prove that the sequence $\langle X_n \bmod p^e\rangle$ has exactly the same period length as the special sequence $\langle A_n \bmod p^e\rangle$ in (b). (No longer period could be obtained by *any* choice of $X_0, \ldots, X_{k-1}$, since the general sequence is a linear combination of "shifts" of the special sequence.) [*Hint:* By exercise 4.6.2–22 (Hensel's lemma), there exist polynomials such that $a(z)f(z) + b(z)g(z) \equiv 1$ (modulo p^e).]

▶ **12.** [*M28*] Find integers X_0, X_1, a, b, and c such that the sequence

$$X_{n+1} = (aX_n + bX_{n-1} + c) \bmod 2^e, \qquad n \geq 1,$$

has the longest period length of all sequences of this type. [*Hint:* It follows that $X_{n+2} = ((a+1)X_{n+1} + (b-a)X_n - bX_{n-1}) \bmod 2^e$; see exercise 11(c).]

13. [*M20*] Let $\langle X_n\rangle$ and $\langle Y_n\rangle$ be sequences of integers mod m with periods of lengths λ_1 and λ_2, and form the sequence $Z_n = (X_n + Y_n) \bmod m$. Show that if λ_1 and λ_2 are relatively prime, the sequence $\langle Z_n\rangle$ has a period of length $\lambda_1\lambda_2$.

14. [*M24*] Let X_n, Y_n, Z_n, λ_1, λ_2 be as in the previous exercise. Suppose that the prime factorization of λ_1 is $2^{e_2}3^{e_3}5^{e_5}\ldots$, and similarly suppose that $\lambda_2 = 2^{f_2}3^{f_3}5^{f_5}\ldots$. Let $g_p = (\max(e_p, f_p)$ if $e_p \neq f_p$, otherwise $0)$, and let $\lambda_0 = 2^{g_2}3^{g_3}5^{g_5}\ldots$. Show that the period length λ' of the sequence $\langle Z_n\rangle$ is a multiple of λ_0, but it is a divisor of $\lambda = \operatorname{lcm}(\lambda_1, \lambda_2)$. In particular, $\lambda' = \lambda$ if ($e_p \neq f_p$ or $e_p = f_p = 0$) for each prime p.

15. [*M27*] Let the sequence $\langle X_n\rangle$ in Algorithm M have period length λ_1, and assume that all elements of its period are distinct. Let $q_n = \min\{\, r \mid r > 0 \text{ and } \lfloor kY_{n-r}/m\rfloor = \lfloor kY_n/m\rfloor \,\}$. Assume that $q_n < \frac{1}{2}\lambda_1$ for all $n \geq n_0$, and that the sequence $\langle q_n\rangle$ has period length λ_2. Let λ be the least common multiple of λ_1 and λ_2. Prove that the output sequence $\langle Z_n\rangle$ produced by Algorithm M has a period of length λ.

▶ **16.** [*M28*] Let CONTENTS(A) in method (10) be $(a_1a_2\ldots a_k)_2$ in binary notation. Show that the generated sequence of low-order bits X_0, X_1, ... satisfies the relation

$$X_n = (a_1X_{n-1} + a_2X_{n-2} + \cdots + a_kX_{n-k}) \bmod 2.$$

[This may be regarded as another way to define the sequence, although the connection between this relation and the efficient code (10) is not apparent at first glance!]

17. [*M33*] (M. H. Martin, 1934.) Let m and k be positive integers, and let $X_1 = X_2 = \cdots = X_k = 0$. For all $n > 0$, set X_{n+k} equal to the largest nonnegative value $y < m$ such that the k-tuple $(X_{n+1}, \ldots, X_{n+k-1}, y)$ has not already occurred in the sequence; in other words, $(X_{n+1}, \ldots, X_{n+k-1}, y)$ must differ from $(X_{r+1}, \ldots, X_{r+k})$ for $0 \leq r < n$. In this way, each possible k-tuple will occur at most once in the

sequence. Eventually the process will terminate, when we reach a value of n such that $(X_{n+1}, \ldots, X_{n+k-1}, y)$ has already occurred in the sequence for all nonnegative $y < m$. For example, if $m = k = 3$ the sequence is 0002221220211210201200111010 0, and the process terminates at this point. (a) Prove that when the sequence terminates, we have $X_{n+1} = \cdots = X_{n+k-1} = 0$. (b) Prove that *every* k-tuple $(a_1, a_2, \ldots, a_k)$ of elements with $0 \le a_j < m$ occurs in the sequence; hence the sequence terminates when $n = m^k$. [*Hint:* Prove that the k-tuple $(a_1, \ldots, a_s, 0, \ldots, 0)$ appears, when $a_s \neq 0$, by induction on s.] Note that if we now define $f(X_n, \ldots, X_{n+k-1}) = X_{n+k}$ for $1 \le n \le m^k$, setting $X_{m^k+k} = 0$, we obtain a function of maximum possible period.

18. [*M22*] Let $\langle X_n \rangle$ be the sequence of bits generated by method (10), with $k = 35$ and CONTENTS(A) $= (00000000000000000000000000000000101)_2$. Let U_n be the binary fraction $(.X_{nk}X_{nk+1}\ldots X_{nk+k-1})_2$; show that this sequence $\langle U_n \rangle$ fails the serial test on pairs (Section 3.3.2B) when $d = 8$.

19. [*M41*] For each prime p specified in the first column of Table 1 in Section 4.5.4, find suitable constants a_1, a_2 as suggested in the text, such that the period length of (8), when $k = 2$, is $p^2 - 1$. (See Eq. 3.3.4–39 for an example.)

20. [*M40*] Calculate constants suitable for use as CONTENTS(A) in method (10), having approximately the same number of zeros as ones, for $2 \le k \le 64$.

21. [*M35*] (D. Rees.) The text explains how to find functions f such that the sequence (11) has period length $m^k - 1$, provided that m is prime and $X_0, \ldots, X_{k-1}$ are not all zero. Show that such functions can be modified to obtain sequences of type (11) with period length m^k, for *all* integers m. [*Hints:* Consider Lemma 3.2.1.2Q, the trick of exercise 7, and sequences such as $\langle pX_{2n} + X_{2n+1} \rangle$.]

▶ **22.** [*M24*] The text restricts discussion of the extended linear sequences (8) to the case that m is prime. Prove that reasonably long periods can also be obtained when m is "square-free," i.e., the product of distinct primes. (Examination of Table 3.2.1.1–1 shows that $m = w \pm 1$ often satisfies this hypothesis; many of the results of the text can therefore be carried over to that case, which is somewhat more convenient for calculation.)

▶ **23.** [*20*] Discuss the sequence defined by $X_n = (X_{n-55} - X_{n-24}) \bmod m$ as an alternative to (7).

24. [*M20*] Let $0 < k < m$. Prove that the sequence of bits defined by the recurrence $X_n = (X_{n-m+k} + X_{n-m}) \bmod 2$ has period length $2^m - 1$ whenever the sequence defined by $Y_n = (Y_{n-k} + Y_{n-m}) \bmod 2$ does.

25. [*26*] Discuss the alternative to Program A that changes all 55 entries of the Y table every 55th time a random number is required.

26. [*M48*] (J. F. Reiser.) Let p be prime and let k be a positive integer. Given integers $a_1, \ldots, a_k$ and $x_1, \ldots, x_k$, let λ_α be the period of the sequence $\langle X_n \rangle$ generated by the recurrence

$$X_n = x_n \bmod p^\alpha, \quad 0 \le n < k; \qquad X_n = (a_1 X_{n-1} + \cdots + a_k X_{n-k}) \bmod p^\alpha, \quad n \ge k;$$

and let N_α be the number of 0's that occur in the period (i.e., the number of indices j such that $\mu_\alpha \le j < \mu_\alpha + \lambda_\alpha$ and $X_j = 0$). Prove or disprove the following conjecture: There exists a constant c (depending possibly on p and k and $a_1, \ldots, a_k$) such that $N_\alpha \le c p^{\alpha(k-2)/(k-1)}$ for all α and all $x_1, \ldots, x_k$.

[*Notes:* Reiser has proved that if the recurrence has maximum period length mod p (i.e., if $\lambda_1 = p^k - 1$), and if the conjecture holds, then the k-dimensional discrepancy of $\langle X_n \rangle$ will be $O(\alpha^k p^{-\alpha/(k-1)})$ as $\alpha \to \infty$; thus an additive generator like (7) would be well distributed in 55 dimensions, when $m = 2^e$ and the entire period is considered. (See Section 3.3.4 for the definition of discrepancy in k dimensions.) The conjecture is a very weak condition, for if $\langle X_n \rangle$ takes on each value about equally often and if $\lambda_\alpha = p^{\alpha-1}(p^k - 1)$, the quantity $N_\alpha \approx (p^k - 1)/p$ does not grow at all as α increases. Reiser has verified the conjecture for $k = 3$. On the other hand he has shown that it is possible to find unusually bad starting values $x_1, \ldots, x_k$ (depending on α) so that $N_{2\alpha} \geq p^\alpha$, provided that $\lambda_\alpha = p^{\alpha-1}(p^k - 1)$ and $k \geq 3$ and α is sufficiently large.]

27. [*M30*] Suppose Algorithm B is being applied to a sequence $\langle X_n \rangle$ whose period length is λ, where $\lambda \gg k$. Show that for fixed k and all sufficiently large λ, the output of the sequence will eventually be periodic with the same period length λ, unless $\langle X_n \rangle$ isn't very random to start with. [*Hint:* Find a pattern of consecutive values of $\lfloor kX_n/m \rfloor$ that causes Algorithm B to "synchronize" its subsequent behavior.]

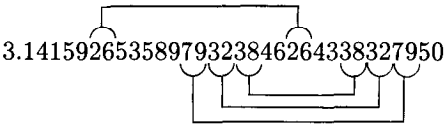
28. [*40*] (A. G. Waterman.) Experiment with linear congruential sequences with m the square or cube of the computer word size, while a and c are single-precision numbers.

3.3. STATISTICAL TESTS

OUR MAIN PURPOSE is to obtain sequences that behave as if they are random. So far we have seen how to make the period of a sequence so long that for practical purposes it never will repeat; this is an important criterion, but it by no means guarantees that the sequence will be useful in applications. How then are we to decide whether a sequence is sufficiently random?

If we were to give some man a pencil and paper and ask him to write down 100 random decimal digits, chances are very slim that he will give a satisfactory result. People tend to avoid things that seem nonrandom, such as pairs of equal adjacent digits (although about one out of every 10 digits should equal its predecessor). And if we would show someone a table of truly random digits, he would quite probably tell us they are not random at all; his eye would spot certain apparent regularities.

According to Dr. I. J. Matrix (as quoted by Martin Gardner in *Scientific American*, January, 1965), "Mathematicians consider the decimal expansion of π a random series, but to a modern numerologist it is rich with remarkable patterns." Dr. Matrix has pointed out, for example, that the first repeated two-digit number in π's expansion is 26, and its second appearance comes in the middle of a curious repetition pattern:



(1)

After listing a dozen or so further properties of these digits, he observed that π, when correctly interpreted, conveys the entire history of the human race!

We all notice patterns in our telephone numbers, license numbers, etc., as aids to memory. The point of these remarks is that we cannot be trusted to judge by ourselves whether a sequence of numbers is random or not. Some unbiased mechanical tests must be applied.

The theory of statistics provides us with some quantitative measures for randomness. There is literally no end to the number of tests that can be conceived; we will discuss those tests that have proved to be most useful, most instructive, and most readily adapted to computer calculation.

If a sequence behaves randomly with respect to tests T_1, T_2, ..., T_n, we cannot be *sure* in general that it will not be a miserable failure when it is subjected to a further test T_{n+1}; yet each test gives us more and more confidence in the randomness of the sequence. In practice, we apply about half a dozen different kinds of statistical tests to a sequence, and if it passes these satisfactorily we consider it to be random—it is then presumed innocent until proven guilty.

Every sequence that is to be used extensively should be tested carefully, so the following sections explain how to carry out these tests in the right way. Two kinds of tests are distinguished: *empirical tests*, for which the computer manipulates groups of numbers of the sequence and evaluates certain statistics; and *theoretical tests*, for which we establish characteristics of the sequence by

using number-theoretic methods based on the recurrence rule used to form the sequence.

If the evidence doesn't come out as desired, the reader may wish to try the techniques in *How to Lie With Statistics* by Darrell Huff (Norton, 1954).

3.3.1. General Test Procedures for Studying Random Data

A. "Chi-square" tests. The chi-square test (χ^2 test) is perhaps the best known of all statistical tests, and it is a basic method that is used in connection with many other tests. Before considering the method in general, let us consider a particular example of the chi-square test as it might be applied to dice throwing. Using two "true" dice (each of which, independently, is assumed to register 1, 2, 3, 4, 5, or 6 with equal probability), the following table gives the probability of obtaining a given total, s, on a single throw:

$$\begin{array}{rccccccccccc} \text{value of } s = & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \text{probability, } p_s = & \frac{1}{36} & \frac{1}{18} & \frac{1}{12} & \frac{1}{9} & \frac{5}{36} & \frac{1}{6} & \frac{5}{36} & \frac{1}{9} & \frac{1}{12} & \frac{1}{18} & \frac{1}{36} \end{array} \tag{1}$$

(For example, a value of 4 can be thrown in three ways: $1+3$, $2+2$, $3+1$; this constitutes $\frac{3}{36} = \frac{1}{12} = p_4$ of the 36 possible outcomes.)

If we throw the dice n times, we should obtain the value s approximately np_s times on the average. For example, in 144 throws we should get the value 4 about 12 times. The following table shows what results were *actually* obtained in a particular sequence of 144 throws of the dice:

$$\begin{array}{rccccccccccc} \text{value of } s = & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \text{observed number, } Y_s = & 2 & 4 & 10 & 12 & 22 & 29 & 21 & 15 & 14 & 9 & 6 \\ \text{expected number, } np_s = & 4 & 8 & 12 & 16 & 20 & 24 & 20 & 16 & 12 & 8 & 4 \end{array} \tag{2}$$

Note that the observed number is different from the expected number in all cases; in fact, random throws of the dice will hardly ever come out with *exactly* the right frequencies. There are 36^{144} possible sequences of 144 throws, all of which are equally likely. One of these sequences consists of all 2's ("snake eyes"), and anyone throwing 144 snake eyes in a row would be convinced that the dice were loaded. Yet the sequence of all 2's is just as probable as any other particular sequence if we specify the outcome of each throw of each die.

In view of this, how can we test whether or not a given pair of dice is loaded? The answer is that we can't make a definite yes-no statement, but we can give a *probabilistic* answer. We can say how probable or improbable certain types of events are.

A fairly natural way to proceed in the above example is to consider the squares of the differences between the observed numbers Y_s and the expected numbers np_s. We can add these together, obtaining

$$V = (Y_2 - np_2)^2 + (Y_3 - np_3)^2 + \cdots + (Y_{12} - np_{12})^2. \tag{3}$$

A bad set of dice should result in a relatively high value of V; and for any given value of V we can ask, "What is the probability that V is this high, using true dice?" If this probability is very small, say $\frac{1}{100}$, we would know that only about one time in 100 would true dice give results so far away from the expected numbers, and we would have definite grounds for suspicion. (Remember, however, that even *good* dice would give such a high value of V about one time in a hundred, so a cautious person would repeat the experiment to see if the high value of V is repeated.)

The statistic V in (3) gives equal weight to $(Y_7 - np_7)^2$ and $(Y_2 - np_2)^2$, although $(Y_7 - np_7)^2$ is likely to be a good deal higher than $(Y_2 - np_2)^2$ since 7's occur about six times as often as 2's. It turns out that the "right" statistic, at least one that has proved to be most important, will give $(Y_7 - np_7)^2$ only $\frac{1}{6}$ as much weight as $(Y_2 - np_2)^2$, and we should change (3) to the following formula:

$$V = \frac{(Y_2 - np_2)^2}{np_2} + \frac{(Y_3 - np_3)^2}{np_3} + \cdots + \frac{(Y_{12} - np_{12})^2}{np_{12}}. \tag{4}$$

This is called the "chi-square" statistic of the observed quantities $Y_2, \ldots, Y_{12}$ in this dice-throwing experiment. For the data in (2), we find that

$$V = \frac{(2-4)^2}{4} + \frac{(4-8)^2}{8} + \cdots + \frac{(9-8)^2}{8} + \frac{(6-4)^2}{4} = 7\frac{7}{48}. \tag{5}$$

The important question now is, of course, "does $7\frac{7}{48}$ constitute an improbably high value for V to assume?" Before answering this question, let us consider the general application of the chi-square method.

In general, suppose that every observation can fall into one of k categories. We take n *independent observations*; this means that the outcome of one observation has absolutely no effect on the outcome of any of the others. Let p_s be the probability that each observation falls into category s, and let Y_s be the number of observations that actually *do* fall into category s. We form the statistic

$$V = \sum_{1 \le s \le k} \frac{(Y_s - np_s)^2}{np_s}. \tag{6}$$

In our example above, there are eleven possible outcomes of each throw of the dice, so $k = 11$. (Eq. (6) is a slight change of notation from Eq. (4), since we are numbering the possibilities from 1 to k instead of from 2 to 12.)

By expanding $(Y_s - np_s)^2 = Y_s^2 - 2np_sY_s + n^2p_s^2$ in (6), and using the facts that

$$\begin{aligned} Y_1 + Y_2 + \cdots + Y_k &= n, \\ p_1 + p_2 + \cdots + p_k &= 1, \end{aligned} \tag{7}$$

we arrive at the formula

$$V = \frac{1}{n} \sum_{1 \le s \le k} \left(\frac{Y_s^2}{p_s} \right) - n, \tag{8}$$

which often makes the computation of V somewhat easier.

Table 1

SELECTED PERCENTAGE POINTS OF THE CHI-SQUARE DISTRIBUTION

	$p = 1\%$	$p = 5\%$	$p = 25\%$	$p = 50\%$	$p = 75\%$	$p = 95\%$	$p = 99\%$
$\nu = 1$	0.00016	0.00393	0.1015	0.4549	1.323	3.841	6.635
$\nu = 2$	0.02010	0.1026	0.5753	1.386	2.773	5.991	9.210
$\nu = 3$	0.1148	0.3518	1.213	2.366	4.108	7.815	11.34
$\nu = 4$	0.2971	0.7107	1.923	3.357	5.385	9.488	13.28
$\nu = 5$	0.5543	1.1455	2.675	4.351	6.626	11.07	15.09
$\nu = 6$	0.8720	1.635	3.455	5.348	7.841	12.59	16.81
$\nu = 7$	1.239	2.167	4.255	6.346	9.037	14.07	18.48
$\nu = 8$	1.646	2.733	5.071	7.344	10.22	15.51	20.09
$\nu = 9$	2.088	3.325	5.899	8.343	11.39	16.92	21.67
$\nu = 10$	2.558	3.940	6.737	9.342	12.55	18.31	23.21
$\nu = 11$	3.053	4.575	7.584	10.34	13.70	19.68	24.73
$\nu = 12$	3.571	5.226	8.438	11.34	14.84	21.03	26.22
$\nu = 15$	5.229	7.261	11.04	14.34	18.25	25.00	30.58
$\nu = 20$	8.260	10.85	15.45	19.34	23.83	31.41	37.57
$\nu = 30$	14.95	18.49	24.48	29.34	34.80	43.77	50.89
$\nu = 50$	29.71	34.76	42.94	49.33	56.33	67.50	76.15
$\nu > 30$	$\nu + \sqrt{2\nu}x_p + \frac{2}{3}x_p^2 - \frac{2}{3} + O(1/\sqrt{\nu})$						
$x_p =$	−2.33	−1.64	−.675	0.00	0.675	1.64	2.33

(For further values, see *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun (Washington, D.C.: U.S. Government Printing Office, 1964), Table 26.8.)

Now we turn to the important question, what constitutes a reasonable value of V? This is found by referring to a table such as Table 1, which gives values of "the chi-square distribution with ν degrees of freedom" for various values of ν. The line of the table with $\nu = k - 1$ is to be used; *the number of "degrees of freedom" is $k-1$, one less than the number of categories.* (Intuitively, this means that $Y_1, Y_2, \ldots, Y_k$ are not completely independent, since Eq. (7) shows that Y_1 can be computed if $Y_2, \ldots, Y_k$ are known; hence, $k - 1$ degrees of freedom are present. This argument is not rigorous, but the theory below justifies it.)

If the table entry in row ν under column p is x, it means, "The quantity V in Eq. (8) will be less than or equal to x with approximate probability p, if n is large enough." For example, the 95 percent entry in row 10 is 18.31; this says we will have $V > 18.31$ only about 5 percent of the time.

Let us assume that the above dice-throwing experiment is simulated on a computer using some sequence of supposedly random numbers, with the following results:

value of $s =$	2	3	4	5	6	7	8	9	10	11	12
Experiment 1, $Y_s =$	4	10	10	13	20	18	18	11	13	14	13
Experiment 2, $Y_s =$	3	7	11	15	19	24	21	17	13	9	5

(9)

We can compute the chi-square statistic in the first case, getting the value $V_1 = 29\frac{59}{120}$, and in the second case we get $V_2 = 1\frac{17}{120}$. Referring to the table entries for 10 degrees of freedom, we see that V_1 is *much too high*; V will be greater than 23.21 only about one percent of the time! (By using more extensive tables, we find in fact that V will be as high as V_1 only 0.1 percent of the time.) Therefore Experiment 1 represents a significant departure from random behavior.

On the other hand, V_2 is quite low, since the observed values Y_s in Experiment 2 are quite close to the expected values np_s in (2). The chi-square table tells us, in fact, that V_2 is *much too low*: the observed values are so close to the expected values, we cannot consider the result to be random! (Indeed, reference to other tables shows that such a low value of V occurs only 0.03 percent of the time when there are 10 degrees of freedom.) Finally, the value $V = 7\frac{7}{48}$ computed in (5) can also be checked with Table 1. It falls between the entries for 25 percent and 50 percent, so we cannot consider it to be significantly high or significantly low; thus the observations in (2) are satisfactorily random with respect to this test.

It is somewhat remarkable that the same table entries are used no matter what the value of n is, and no matter what the probabilities p_s are. Only the number $\nu = k - 1$ affects the results. In actual fact, however, the table entries are not exactly correct: *the chi-square distribution is an approximation that is valid only for large enough values of* n. How large should n be? A common rule of thumb is to take n large enough so that each of the expected values np_s is five or more; preferably, however, take n much larger than this, to get a more powerful test. Note that in our examples above we took $n = 144$, so np_2 was only 4, violating the stated "rule of thumb." This was done only because the author tired of throwing the dice; it makes the entries in Table 1 less accurate for our application. Experiments run on a computer, with $n = 1000$, or 10000, or even 100000, would be much better than this. We could also combine the data for $s = 2$ and $s = 12$; then the test would have only nine degrees of freedom but the chi-square approximation would be more accurate.

We can get an idea of how crude an approximation is involved by considering the case when there are only two categories, having respective probabilities p_1 and p_2. Suppose $p_1 = \frac{1}{4}$ and $p_2 = \frac{3}{4}$. According to the stated rule of thumb, we should have $n \geq 20$ to have a satisfactory approximation, so let's check this out. When $n = 20$, the possible values of V are $\frac{4}{15}r^2$ for $-5 \leq r \leq 15$; we wish to know how well the $\nu = 1$ row of Table 1 describes the distribution of V. The chi-square distribution varies continuously, while the actual distribution

of V has rather big jumps, so we need some convention for representing the exact distribution. If the distinct possible outcomes of the experiment lead to the values $V_0 \le V_1 \le \cdots \le V_n$ with respective probabilities $\pi_0, \pi_1, \ldots, \pi_n$, suppose that a given percentage p falls in the range $\pi_0 + \cdots + \pi_{j-1} < p < \pi_0 + \cdots + \pi_{j-1} + \pi_j$. We would like to represent p by a "percentage point" x such that V is less than x with probability $\le p$ and V is greater than x with probability $\le 1 - p$. It is not difficult to see that the only such number is $x = V_j$. In our example for $n = 20$ and $\nu = 1$, it turns out that the percentage points of the exact distribution, corresponding to the approximations in Table 1 for $p = 1\%$, 5%, 25%, 50%, 75%, 95%, and 99%, respectively, are

$$0, \quad 0, \quad .27, \quad .27, \quad 1.07, \quad 4.27, \quad 6.67.$$

For example, the percentage point for $p = 95\%$ is 4.27, while Table 1 gives the estimate 3.841. The latter value is too low; it tells us (incorrectly) to reject the value $V = 4.27$ at the 95% level, while in fact the probability that $V \ge 4.27$ is more than 6.5%. When $n = 21$, the situation changes slightly because the expected values $np_1 = 5.25$ and $np_2 = 15.75$ can never be obtained exactly; the percentage points for $n = 21$ are

$$.02, \quad .02, \quad .14, \quad .40, \quad 1.29, \quad 3.57, \quad 5.73.$$

We would expect Table 1 to be a better approximation when $n = 50$, but the corresponding tableau actually turns out to be further from Table 1 in some respects than it was for $n = 20$:

$$.03, \quad .03, \quad .03, \quad .67, \quad 1.31, \quad 3.23, \quad 6.0.$$

Here are the values when $n = 300$:

$$0, \quad 0, \quad .07, \quad .44, \quad 1.44, \quad 4.0, \quad 6.42.$$

Even in this case, when np_s is ≥ 75 in each category, the entries in Table 1 are good to only about one significant digit.

The proper choice of n is somewhat obscure. If the dice are actually biased, the fact will be detected as n gets larger and larger. (Cf. exercise 12.) But large values of n will tend to smooth out *locally* nonrandom behavior, i.e., blocks of numbers with a strong bias followed by blocks of numbers with the opposite bias. This type of behavior would not happen when actual dice are rolled, since the same dice are used throughout the test, but a sequence of numbers generated on a computer might very well display such locally nonrandom behavior. Perhaps a chi-square test should be made for a number of different values of n. At any rate, n should always be rather large.

We can summarize the chi-square test as follows. A fairly large number, n, of independent observations is made. (It is important to avoid using the chi-square method unless the observations are independent. See, for example,

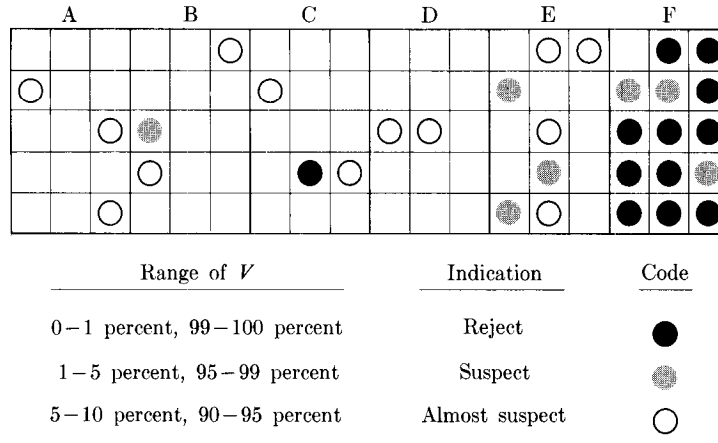


Fig. 2. Indications of "significant" deviations in 90 chi-square tests (cf. also Fig. 5).

exercise 10, which considers the case when half of the observations depend on the other half.) We count the number of observations falling into each of k categories and compute the quantity V given in Eqs. (6) and (8). Then V is compared with the numbers in Table 1, with $\nu = k - 1$. If V is less than the 1% entry or greater than the 99% entry, we reject the numbers as not sufficiently random. If V lies between the 1% and 5% entries or between the 95% and 99% entries, the numbers are "suspect"; if (by interpolation in the table) V lies between the 5% and 10% entries, or the 90% and 95% entries, the numbers might be "almost suspect." The chi-square test is often done at least three times on different sets of data, and if at least two of the three results are suspect the numbers are regarded as not sufficiently random.

For example, see Fig. 2, which shows schematically the results of applying five different types of chi-square tests on each of six sequences of random numbers. Each test has been applied to three different blocks of numbers of the sequence. Generator A is the MacLaren–Marsaglia method (Algorithm 3.2.2M applied to the sequences in 3.2.2–12), Generator E is the Fibonacci method, and the other generators are linear congruential sequences with the following parameters:

Generator B: $X_0 = 0, \quad a = 3141592653, \quad c = 2718281829, \quad m = 2^{35}.$
Generator C: $X_0 = 0, \quad a = 2^7 + 1, \quad c = 1, \quad m = 2^{35}.$
Generator D: $X_0 = 47594118, \quad a = 23, \quad c = 0, \quad m = 10^8 + 1.$
Generator F: $X_0 = 314159265, \quad a = 2^{18} + 1, \quad c = 1, \quad m = 2^{35}.$

From Fig. 2 we conclude that (so far as these tests are concerned) Generators A, B, D are satisfactory, Generator C is on the borderline and should probably be rejected, Generators E and F are definitely unsatisfactory. Generator F has, of course, low potency; Generators C and D have been discussed in the

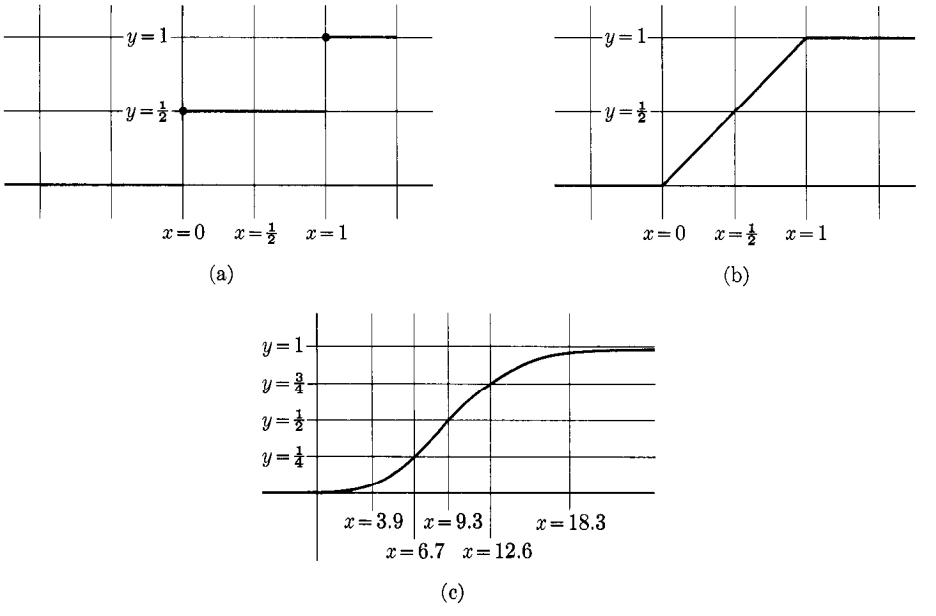


Fig. 3. Examples of distribution functions.

literature, but their multipliers are too small. (Generator D is the original multiplicative generator proposed by Lehmer in 1948; Generator C is the original linear congruential generator with $c \neq 0$ proposed by Rotenberg in 1960.)

Instead of using the "suspect," "almost suspect," etc., criteria for judging the results of chi-square tests, there is a less *ad hoc* procedure available, which will be discussed later in this section.

B. The Kolmogorov–Smirnov test. As we have seen, the chi-square test applies to the situation when observations can fall into a finite number of categories. It is not unusual, however, to consider random quantities that may assume infinitely many values. For example, a random real number between 0 and 1 may take on infinitely many values; even though only a finite number of these can be represented in the computer, we want our random values to behave essentially as though though they are random real numbers.

A general notation for specifying probability distributions, whether they are finite or infinite, is commonly used in the study of probability and statistics. Suppose we want to specify the distribution of the values of a random quantity, X; we do this in terms of the *distribution function* $F(x)$, where

$$F(x) = \text{probability that } (X \leq x).$$

Three examples are shown in Fig. 3. First we see the distribution function for a *random bit*, i.e., for the case when X takes on only the two values 0 and 1, each with probability $\frac{1}{2}$. Part (b) of the figure shows the distribution function

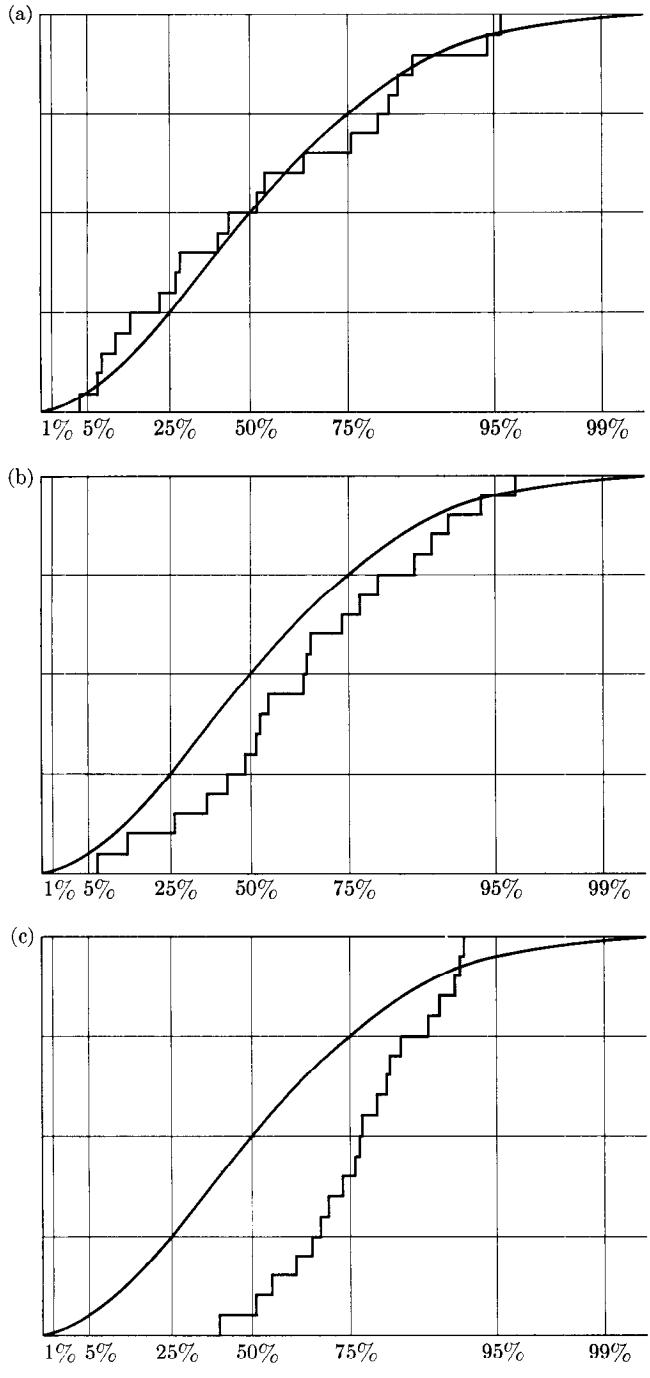


Fig. 4. Examples of empirical distributions.

for a *uniformly distributed random real number* between zero and one, so the probability that $X \le x$ is simply equal to x when $0 \le x \le 1$; for example, the probability that $X \le \frac{2}{3}$ is, naturally, $\frac{2}{3}$. And part (c) shows the limiting distribution of the value V in the chi-square test (shown here with 10 degrees of freedom); this is a distribution that we have already seen represented in another way in Table 1. Note that $F(x)$ always increases from 0 to 1 as x increases from $-\infty$ to $+\infty$.

If we make n independent observations of the random quantity X, thereby obtaining the values $X_1, X_2, \ldots, X_n$, we can form the *empirical distribution function* $F_n(x)$, where

$$F_n(x) = \frac{\text{number of } X_1, X_2, \ldots, X_n \text{ that are } \le x}{n}. \tag{10}$$

Figure 4 illustrates three empirical distribution functions (shown as zigzag lines, although strictly speaking the vertical lines are not part of the graph of $F_n(x)$), superimposed on a graph of the assumed actual distribution function $F(x)$. As n gets large, $F_n(x)$ should be a better and better approximation to $F(x)$.

The Kolmogorov–Smirnov test (KS test) may be used when $F(x)$ has no jumps. It is based on the *difference between* $F(x)$ *and* $F_n(x)$. A bad source of random numbers will give empirical distribution functions that do not approximate $F(x)$ sufficiently well. Figure 4(b) shows an example in which the X_i are consistently too high, so the empirical distribution function is too low. Part (c) of the figure shows an even worse example; it is plain that such great deviations between $F_n(x)$ and $F(x)$ are extremely improbable, and the KS test is used to tell us how improbable they are.

To make the test, we form the following statistics:

$$\begin{aligned} K_n^+ &= \sqrt{n} \max_{-\infty<x<+\infty} \big(F_n(x) - F(x)\big); \\ K_n^- &= \sqrt{n} \max_{-\infty<x<+\infty} \big(F(x) - F_n(x)\big). \end{aligned} \tag{11}$$

Here K_n^+ measures the greatest amount of deviation when F_n is greater than F, and K_n^- measures the maximum deviation when F_n is less than F. The statistics for the examples of Fig. 4 are

	Part (a)	Part (b)	Part (c)
K_{20}^+	0.492	0.134	0.313
K_{20}^-	0.536	1.027	2.101

(12)

(*Note:* The factor $\sqrt{n}$ that appears in Eqs. (11) may seem puzzling at first. Exercise 6 shows that, for fixed x, the standard deviation of $F_n(x)$ is proportional to $1/\sqrt{n}$; hence the factor $\sqrt{n}$ magnifies the statistics K_n^+, K_n^- in such a way that this standard deviation is independent of n.)

Table 2

SELECTED PERCENTAGE POINTS OF THE DISTRIBUTIONS K_n^+ AND K_n^-

	$p = 1\%$	$p = 5\%$	$p = 25\%$	$p = 50\%$	$p = 75\%$	$p = 95\%$	$p = 99\%$
$n = 1$	0.01000	0.05000	0.2500	0.5000	0.7500	0.9500	0.9900
$n = 2$	0.01400	0.06749	0.2929	0.5176	0.7071	1.0980	1.2728
$n = 3$	0.01699	0.07919	0.3112	0.5147	0.7539	1.1017	1.3589
$n = 4$	0.01943	0.08789	0.3202	0.5110	0.7642	1.1304	1.3777
$n = 5$	0.02152	0.09471	0.3249	0.5245	0.7674	1.1392	1.4024
$n = 6$	0.02336	0.1002	0.3272	0.5319	0.7703	1.1463	1.4144
$n = 7$	0.02501	0.1048	0.3280	0.5364	0.7755	1.1537	1.4246
$n = 8$	0.02650	0.1086	0.3280	0.5392	0.7797	1.1586	1.4327
$n = 9$	0.02786	0.1119	0.3274	0.5411	0.7825	1.1624	1.4388
$n = 10$	0.02912	0.1147	0.3297	0.5426	0.7845	1.1658	1.4440
$n = 11$	0.03028	0.1172	0.3330	0.5439	0.7863	1.1688	1.4484
$n = 12$	0.03137	0.1193	0.3357	0.5453	0.7880	1.1714	1.4521
$n = 15$	0.03424	0.1244	0.3412	0.5500	0.7926	1.1773	1.4606
$n = 20$	0.03807	0.1298	0.3461	0.5547	0.7975	1.1839	1.4698
$n = 30$	0.04354	0.1351	0.3509	0.5605	0.8036	1.1916	1.4801
$n > 30$	$y_p - 1/\left(6\sqrt{n}\right) + O(1/n)$, where $y_p^2 = \frac{1}{2}\ln(1/(1-p))$						
$y_p =$	0.07089	0.1601	0.3793	0.5887	0.8326	1.2239	1.5174

As in the chi-square test, we may now look up the values K_n^+, K_n^- in a "percentile" table to determine if they are significantly high or low. Table 2 may be used for this purpose, both for K_n^+ and K_n^-. For example, the probability is 75 percent that K_{20}^- will be 0.7975 or less. Unlike the chi-square test, the table entries are *not* merely approximations that hold for large values of n; Table 2 gives exact values (except, of course, for roundoff error), and the KS test may be reliably used for any value of n.

As they stand, formulas (11) are not readily adapted to computer calculation, since we are asking for a maximum over infinitely many values of x. From the fact that $F(x)$ is increasing and the fact that $F_n(x)$ increases only in finite steps, however, we can derive a simple procedure for evaluating the statistics K_n^+ and K_n^-:

Step 1. Obtain the observations $X_1, X_2, \ldots, X_n$.

Step 2. Rearrange the observations so that they are sorted into ascending order, i.e., so that $X_1 \le X_2 \le \cdots \le X_n$. (Efficient sorting algorithms are the subject

of Chapter 5. On the other hand, it is possible to avoid sorting in this particular application, as shown in exercise 23.)

Step 3. The desired statistics are now given by the formulas

$$K_n^+ = \sqrt{n} \max_{1\le j\le n} \left(\frac{j}{n} - F(X_j)\right);$$
$$K_n^- = \sqrt{n} \max_{1\le j\le n} \left(F(X_j) - \frac{j-1}{n}\right). \tag{13}$$

An appropriate choice of the number of observations, n, is slightly easier to make for this test than it is for the χ^2 test, although some of the considerations are similar. If the random variables X_j actually belong to the probability distribution $G(x)$, while they were assumed to belong to the distribution given by $F(x)$, it will take a comparatively large value of n to reject the hypothesis that $G(x) = F(x)$; for we need n large enough that the empirical distributions $G_n(x)$ and $F_n(x)$ are expected to be observably different. On the other hand, large values of n will tend to average out locally nonrandom behavior, and such behavior is an undesirable characteristic that is of significant importance in most computer applications of random numbers; this makes a case for *smaller* values of n. A good compromise would be to take n equal to, say, 1000, and to make a fairly large number of calculations of K_{1000}^+ on different parts of a random sequence, thereby obtaining values

$$K_{1000}^+(1), \qquad K_{1000}^+(2), \qquad \ldots, \qquad K_{1000}^+(r). \tag{14}$$

We can also apply the KS test *again* to *these* results: Let $F(x)$ now be the distribution function for K_{1000}^+, and determine the empirical distribution $F_r(x)$ obtained from the observed values in (14). Fortunately, the function $F(x)$ in this case is very simple; for a large value of n like $n = 1000$, the distribution of K_n^+ is closely approximated by

$$F_\infty(x) = 1 - e^{-2x^2}, \qquad x \ge 0. \tag{15}$$

The same remarks apply to K_n^-, since K_n^+ and K_n^- have the same expected behavior. *This method of using several tests for moderately large n, then combining the observations later in another KS test, will tend to detect both local and global nonrandom behavior.*

An experiment of this type (although on a much smaller scale) was made by the author as this chapter was being written. The "maximum of 5" test described in the next section was applied to a set of 1000 uniform random numbers, yielding 200 observations X_1, X_2, ..., X_{200} that were supposed to belong to the distribution $F(x) = x^5$ $(0 \le x \le 1)$. The observations were divided into 20 groups of 10 each, and the statistic K_{10}^+ was computed for each group. The 20 values of K_{10}^+, thus obtained, led to the empirical distributions shown in Fig. 4. The smooth curve shown in each of the diagrams in Fig. 4

is the actual distribution the statistic K_{10}^+ should have. Figure 4(a) shows the empirical distribution of K_{10}^+ obtained from the sequence

$$Y_{n+1} = (3141592653Y_n + 2718281829) \bmod 2^{35}, \qquad U_n = Y_n/2^{35},$$

and it is satisfactorily random. Part (b) of the figure came from the Fibonacci method; this sequence has *globally* nonrandom behavior, i.e., it can be shown that the observations X_n in the "maximum of 5" test do not have the correct distribution $F(x) = x^5$. Part (c) came from the notorious and impotent linear congruential sequence $Y_{n+1} = ((2^{18} + 1)Y_n + 1) \bmod 2^{35}$, $U_n = Y_n/2^{35}$.

The KS test applied to the data in Fig. 4 gives the results shown in (12). Referring to Table 2 for $n = 20$, we see that the values of K_{20}^+ and K_{20}^- for Fig. 4(b) are almost suspect (they lie at about the 5 percent and 88 percent levels) but not quite bad enough to be rejected outright. The value of K_{20}^- for Part (c) is, of course, completely out of line, so the "maximum of 5" test shows a definite failure of that random number generator.

We would expect the KS test in this experiment to have more difficulty locating global nonrandomness than local nonrandomness, since the basic observations in Fig. 4 were made on samples of only 10 numbers each. If we were to take 20 groups of 1000 numbers each, part (b) would show a much more significant deviation. To illustrate this point, a *single* KS test was applied to all 200 of the observations that led to Fig. 4, and the following results were obtained:

	Part (a)	Part (b)	Part (c)
K_{200}^+	0.477	1.537	2.819
K_{200}^-	0.817	0.194	0.058

(16)

The global nonrandomness of the Fibonacci generator has definitely been detected here.

We may summarize the Kolmogorov–Smirnov test as follows. We are given n *independent observations* $X_1, \ldots, X_n$ taken from some distribution specified by a *continuous* function $F(x)$. That is, $F(x)$ must be like the functions shown in Fig. 3(b) and 3(c), having no jumps like those in Fig. 3(a). The procedure explained just before Eqs. (13) is carried out on these observations, so we obtain the statistics K_n^+ and K_n^-. These statistics should be distributed according to Table 2.

Some comparisons between the KS test and the χ^2 test can now be made. In the first place, we should observe that the KS test may be used in conjunction with the χ^2 test, to give a better procedure than the *ad hoc* method we mentioned when summarizing the χ^2 test. (That is, there is a better way to proceed than to make three tests and to consider how many of the results were "suspect".) Suppose we have made, say, 10 independent χ^2 tests on different parts of a random sequence, so that values $V_1, V_2, \ldots, V_{10}$ have been obtained. It is not a good policy simply to count how many of the V's are suspiciously large or small. This procedure will work in extreme cases, and very large or very small values

may mean that the sequence has too much local nonrandomness; but a better general method would be to plot the empirical distribution of these 10 values and to compare it to the correct distribution, which may be obtained from Table 1. This would give a clearer picture of the results of the χ^2 tests, and in fact the statistics K_{10}^+ and K_{10}^- could be determined as an indication of the success or failure. With only 10 values or even as many as 100 this could all be done easily by hand, using graphical methods; with a larger number of V's, a computer subroutine for calculating the chi-square distribution would be necessary. Notice that *all 20 of the observations in Fig. 4(c) fall between the 5 and 95 percent levels*, so we would not have regarded *any* of them as suspicious, individually; yet collectively the empirical distribution shows that these observations are not at all right.

An important difference between the KS test and the chi-square test is that the KS test applies to distributions $F(x)$ having no jumps, while the chi-square test applies to distributions having nothing but jumps (since all observations are divided into k categories). The two tests are thus intended for different sorts of applications. Yet it is possible to apply the χ^2 test even when $F(x)$ is continuous, if we divide the domain of $F(x)$ into k parts and ignore all variations within each part. For example, if we want to test whether or not U_1, U_2, ..., U_n can be considered to come from the uniform distribution between zero and one, we want to test if they have the distribution $F(x) = x$ for $0 \le x \le 1$. This is a natural application for the KS test. But we might also divide up the interval from 0 to 1 into $k = 100$ equal parts, count how many U's fall into each part, and apply the chi-square test with 99 degrees of freedom. There are not many theoretical results available at the present time to compare the effectiveness of the KS test versus the chi-square test. The author has found some examples in which the KS test pointed out nonrandomness more clearly than the χ^2 test, and others in which the χ^2 test gave a more significant result. If, for example, the 100 categories mentioned above are numbered 0, 1, ..., 99, and if the deviations from the expected values are positive in compartments 0 to 49 but negative in compartments 50 to 99, then the empirical distribution function will be much further from $F(x)$ than the χ^2 value would indicate; but if the positive deviations occur in compartments 0, 2, ..., 98 and the negative ones occur in 1, 3, ..., 99, the empirical distribution function will tend to hug $F(x)$ much more closely. The kinds of deviations measured are therefore somewhat different. A χ^2 test was applied to the 200 observations that led to Fig. 4, with $k = 10$, and the respective values of V were 9.4, 17.7, and 39.3; so in this particular case the values are quite comparable to the KS values given in (16). Since the χ^2 test is intrinsically less accurate, and since it requires comparatively large values of n, the KS test has several advantages when a continuous distribution is to be tested.

A further example will also be of interest. The data that led to Fig. 2 were chi-square statistics based on $n = 200$ observations of the "maximum-of-t" criterion for $1 \le t \le 5$, with the range divided into 10 equally probable parts. KS statistics K_{200}^+ and K_{200}^- can be computed from exactly the same sets of 200 observations, and the results can be tabulated in just the same way as we did

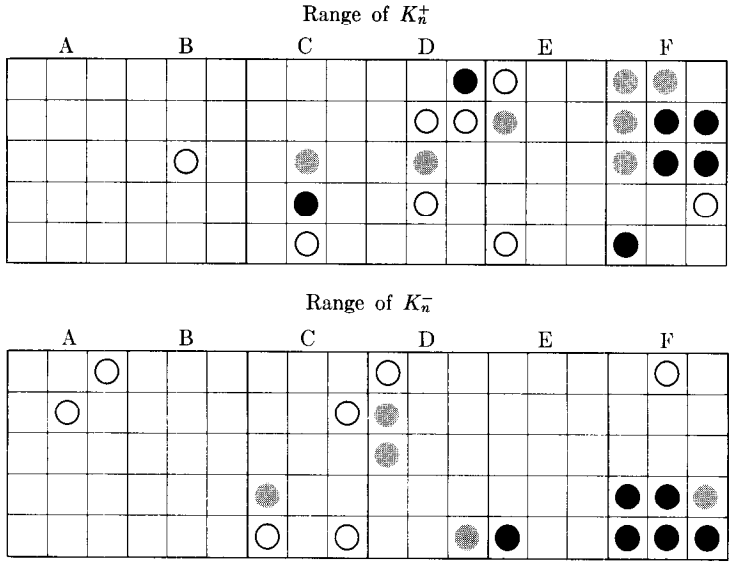


Fig. 5. The KS tests applied to the same data as Fig. 2.

in Fig. 2 (showing which KS values are beyond the 99-percent level, etc.); the results in this case are shown in Fig. 5. Note that Generator D (Lehmer's original method) shows up very badly in Fig. 5, while chi-square tests *on the very same data* revealed no difficulty in Fig. 2; contrariwise, Generator E (the Fibonacci method) does not look so bad in Fig. 5. The good generators, A and B, passed all tests satisfactorily. The reasons for the discrepancies between Fig. 2 and Fig. 5 are primarily that (a) the number of observations, 200, is really not large enough for a powerful test, and (b) the "reject," "suspect," "almost suspect" ranking criterion is itself suspect.

(Incidentally, it is not fair to blame Lehmer for using a "bad" random number generator in the 1940s, since his actual use of Generator D was quite valid. The ENIAC computer was a highly parallel machine, programmed by means of a plugboard; Lehmer set it up so that one of its accumulators was repeatedly multiplying its own contents by 23, mod (10^8+1), yielding a new value every few microseconds. Since this multiplier 23 is too small, we know that each value obtained by such a process was too strongly related to the preceding value to be considered sufficiently random; but the durations of time between actual *uses* of the values in the special accumulator by the accompanying program were comparatively long and subject to some fluctuation. So the effective multiplier was 23^k for large, *varying* values of k.)

C. History, bibliography, and theory. The chi-square test was introduced by Karl Pearson in 1900 [*Philosophical Magazine*, Series 5, **50**, 157–175]. Pearson's important paper is regarded as one of the foundations of modern statistics, since before that time people would simply plot experimental results graphically and assert that they were correct. In his paper, Pearson gave several interesting

examples of the previous misuse of statistics; and he also proved that certain runs at roulette (which he had experienced during two weeks at Monte Carlo in 1892) were so far from the expected frequencies that odds against the assumption of an honest wheel were some 10^{29} to one! A general discussion of the chi-square test and an extensive bibliography appear in the survey article by William G. Cochran, *Annals Math. Stat.* **23** (1952), 315–345.

Let us now consider a brief derivation of the theory behind the chi-square test. The exact probability that $Y_1 = y_1, \ldots, Y_k = y_k$ is easily seen to be

$$\frac{n!}{y_1! \ldots y_k!} p_1^{y_1} \ldots p_k^{y_k}. \tag{17}$$

If we assume that Y_s has the value y_s with the Poisson probability

$$\frac{e^{-np_s}(np_s)^{y_s}}{y_s!},$$

and that the Y's are independent, then $(Y_1, \ldots, Y_k)$ will equal $(y_1, \ldots, y_k)$ with probability

$$\prod_{1 \le s \le k} \frac{e^{-np_s}(np_s)^{y_s}}{y_s!},$$

and $Y_1 + \cdots + Y_k$ will equal n with probability

$$\sum_{\substack{y_1 + \cdots + y_k = n \\ y_1, \ldots, y_k \ge 0}} \prod_{1 \le s \le k} \frac{e^{-np_s}(np_s)^{y_s}}{y_s!} = \frac{e^{-n} n^n}{n!}.$$

If we assume that they are independent *except* for the condition $Y_1 + \cdots + Y_k = n$, the probability that $(Y_1, \ldots, Y_k) = (y_1, \ldots, y_k)$ is the quotient

$$\left(\prod_{1 \le s \le k} \frac{e^{-np_s}(np_s)^{y_s}}{y_s!} \right) \Big/ \left(\frac{e^{-n} n^n}{n!} \right),$$

which equals (17). *We may therefore regard the Y's as independently Poisson distributed, except for the fact that they have a fixed sum.*

It is convenient to make a change of variables,

$$Z_s = \frac{Y_s - np_s}{\sqrt{np_s}}, \tag{18}$$

so that $V = Z_1^2 + \cdots + Z_k^2$. The condition $Y_1 + \cdots + Y_k = n$ is equivalent to requiring that

$$\sqrt{p_1}\, Z_1 + \cdots + \sqrt{p_k}\, Z_k = 0. \tag{19}$$

Let us consider the $(k - 1)$-dimensional space S of all vectors $(Z_1, \ldots, Z_k)$ such that (19) holds. For large values of n, each Z_s has approximately the

normal distribution (cf. exercise 1.2.10–16); therefore points in a differential volume $dz_2 \dots dz_k$ of S occur with probability *approximately* proportional to $\exp\bigl(-(z_1^2 + \cdots + z_k^2)/2\bigr)$. (It is at this point in the derivation that the chi-square method becomes only an approximation for large n.) The probability that $V \le v$ is now

$$\frac{\int_{(z_1,\dots,z_k)\text{ in }S\text{ and }z_1+\cdots+z_k\le v} \exp\bigl(-(z_1^2+\cdots+z_k^2)/2\bigr)\,dz_2\dots dz_k}{\int_{(z_1,\dots,z_k)\text{ in }S} \exp\bigl(-(z_1^2+\cdots+z_k^2)/2\bigr)\,dz_2\dots dz_k}. \tag{20}$$

Since the hyperplane (19) passes through the origin of k-dimensional space, the numerator in (20) is an integration over the interior of a $(k-1)$-dimensional hypersphere centered at the origin. An appropriate transformation to generalized polar coordinates with radius χ and angles $\omega_1, \dots, \omega_{k-2}$ transforms (20) into

$$\frac{\int_{\chi^2\le v} e^{-\chi^2/2}\chi^{k-2} f(\omega_1,\dots,\omega_{k-2})\,d\chi\,d\omega_1\dots d\omega_{k-2}}{\int e^{-\chi^2/2}\chi^{k-2} f(\omega_1,\dots,\omega_{k-2})\,d\chi\,d\omega_1\dots d\omega_{k-2}}$$

for some function f (see exercise 15); then integration over the angles $\omega_1, \dots, \omega_{k-2}$ gives a constant factor that cancels from numerator and denominator. We finally obtain the formula

$$\frac{\int_0^{\sqrt{v}} e^{-\chi^2/2}\chi^{k-2}\,d\chi}{\int_0^{\infty} e^{-\chi^2/2}\chi^{k-2}\,d\chi} \tag{21}$$

for the approximate probability that $V \le v$.

The above derivation uses the symbol χ to stand for the radial length, just as Pearson did in his original paper; this is how the χ^2 test got its name. Substituting $t = \chi^2/2$, the integrals can be expressed in terms of the incomplete gamma function, which we discussed in Section 1.2.11.3:

$$\lim_{n\to\infty} \text{probability that } (V \le v) = \gamma\Bigl(\frac{k-1}{2}, \frac{v}{2}\Bigr)\Big/\Gamma\Bigl(\frac{k-1}{2}\Bigr). \tag{22}$$

This is the definition of the chi-square distribution with $k-1$ degrees of freedom.

We now turn to the KS test. In 1933, A. N. Kolmogorov proposed a test based on the statistic

$$K_n = \sqrt{n} \max_{-\infty<x<+\infty} |F_n(x) - F(x)| = \max(K_n^+, K_n^-). \tag{23}$$

N. V. Smirnov gave several modifications of this test in 1939, including the individual examination of K_n^+ and K_n^- as we have suggested above. There is a large family of similar tests, but the K_n^+ and K_n^- statistics seem to be most convenient for computer application. A comprehensive review of the literature concerning KS tests and their generalizations, including an extensive bibliography, appears in a monograph by J. Durbin, *Regional Conf. Series on Applied Math.* **9** (SIAM, 1973).

To study the distribution of K_n^+ and K_n^-, we begin with the following basic fact: *If X is a random variable with the continuous distribution $F(x)$, then $F(X)$ is a uniformly distributed real number between 0 and 1.* To prove this, we need only verify that if $0 \le y \le 1$ we have $F(X) \le y$ with probability y. Since F is continuous, $F(x_0) = y$ for some x_0; thus the probability that $F(X) \le y$ is the probability that $X \le x_0$. By definition, the latter probability is $F(x_0)$, that is, it is y.

Let $Y_j = nF(X_j)$, for $1 \le j \le n$, where the X's have been sorted as in Step 2 above. Then the variables Y_j are essentially the same as independent, uniformly distributed random numbers between 0 and 1 that have been sorted into nondecreasing order, $Y_1 \le Y_2 \le \cdots \le Y_n$; and the first equation of (13) may be transformed into

$$K_n^+ = \frac{1}{\sqrt{n}} \max(1 - Y_1, 2 - Y_2, \ldots, n - Y_n).$$

If $0 \le t \le n$, the probability that $K_n^+ \le t/\sqrt{n}$ is therefore the probability that $Y_j \ge j - t$ for $1 \le j \le n$. This is not hard to express in terms of n-dimensional integrals,

$$\frac{\int_{\alpha_n}^{n} dy_n \int_{\alpha_{n-1}}^{y_n} dy_{n-1} \ \cdots \int_{\alpha_1}^{y_2} dy_1}{\int_0^n dy_n \int_0^{y_n} dy_{n-1} \ \cdots \int_0^{y_2} dy_1}, \qquad \text{where} \qquad \alpha_j = \max(j - t, 0). \tag{24}$$

The denominator here is immediately evaluated: it is found to be $n^n/n!$, which makes sense since the hypercube of all vectors $(y_1, y_2, \ldots, y_n)$ with $0 \le y_j < n$ has volume n^n, and it can be divided into $n!$ equal parts corresponding to each possible ordering of the y's. The integral in the numerator is a little more difficult, but it yields to the attack suggested in exercise 17, and we get the general formula

$$\text{probability that } \left(K_n^+ \le \frac{t}{\sqrt{n}}\right) = \frac{t}{n^n} \sum_{0 \le k \le t} \binom{n}{k} (k - t)^k (t + n - k)^{n-k-1}. \tag{25}$$

The distribution of K_n^- is exactly the same. Equation (25) was first obtained by Z. W. Birnbaum and Fred H. Tingey [*Annals Math. Stat.* **22** (1951), 592–596]; it may be used to extend Table 2.

In his original paper, Smirnov proved that

$$\lim_{n \to \infty} \text{probability that } (K_n^+ \le s) = 1 - e^{-2s^2}, \qquad \text{if} \qquad s \ge 0. \tag{26}$$

This together with (25) implies that, for all $s \ge 0$, we have

$$\lim_{n \to \infty} \frac{s}{\sqrt{n}} \sum_{\sqrt{n}s < k \le n} \binom{n}{k} \left(\frac{k}{n} - \frac{s}{\sqrt{n}}\right)^k \left(\frac{s}{\sqrt{n}} + 1 - \frac{k}{n}\right)^{n-k-1} = e^{-2s^2}. \tag{27}$$

The more precise asymptotic formulas in Table 2 follow from results obtained by D. A. Darling [*Theory of Prob. and Appl.* **5** (1960), 356-361], who proved among other things that $K_n^+ \le s$ with probability

$$1 - e^{-2s^2}\left(1 - \tfrac{2}{3}s/\sqrt{n} + O(1/n)\right) \tag{28}$$

for all fixed $s \ge 0$.

EXERCISES

1. [*00*] What line of the chi-square table should be used to check whether or not the value $V = 7\frac{7}{48}$ of Eq. (5) is improbably high?

2. [*20*] If two dice are "loaded" so that, on one die, the value 1 will turn up exactly twice as often as any of the other values, and the other die is similarly biased towards 6, compute the probability p_s that a total of exactly s will appear on the two dice, for $2 \le s \le 12$.

▶ **3.** [*23*] Some dice that were loaded as described in the previous exercise were rolled 144 times, and the following values were observed:

value of $s =$	2	3	4	5	6	7	8	9	10	11	12
observed number, $Y_s =$	2	6	10	16	18	32	20	13	16	9	2

Apply the chi-square test to *these* values, using the probabilities in (1), pretending it is not known that the dice are in fact faulty. Does the chi-square test detect the bad dice? If not, explain why not.

▶ **4.** [*23*] The author actually obtained the data in experiment 1 of (9) by simulating dice in which one was normal, the other was loaded so that it always turned up 1 or 6. (The latter two possibilities were equally probable.) Compute the probabilities that replace (1) in this case, and by using a chi-square test decide if the results of that experiment are consistent with the dice being loaded in this way.

5. [*22*] Let $F(x)$ be the uniform distribution, Fig. 3(b). Find K_{20}^+ and K_{20}^- for the following 20 observations:

0.414, 0.732, 0.236, 0.162, 0.259, 0.442, 0.189, 0.693, 0.098, 0.302,

0.442, 0.434, 0.141, 0.017, 0.318, 0.869, 0.772, 0.678, 0.354, 0.718,

and state whether these observations are significantly different from expected behavior with respect to either of these two tests.

6. [*M20*] Consider $F_n(x)$, as given in Eq. (10), for fixed x. What is the probability that $F_n(x) = s/n$, given an integer s? What is the mean value of $F_n(x)$? What is the standard deviation?

7. [*M15*] Show that K_n^+ and K_n^- can never be negative. What is the largest possible value K_n^+ can be?

8. [*00*] The text describes an experiment in which 20 values of the statistic K_{10}^+ were obtained in the study of a random sequence. These values were plotted, to obtain Fig. 4, and a KS statistic was computed from the resulting graph. Why were the table entries for $n = 20$ used to study the resulting statistic, instead of the table entries for $n = 10$?

▶ **9.** [*20*] The experiment described in the text consisted of plotting 20 values of K_{10}^+, computed from the "maximum of 5" test applied to different parts of a random sequence. We could have computed also the corresponding 20 values of K_{10}^-; since K_{10}^- has the same distribution as K_{10}^+, we could lump together the 40 values thus obtained (that is, 20 of the K_{10}^+'s and 20 of the K_{10}^-'s), and a KS test could be applied so that we would get new values K_{40}^+, K_{40}^-. Discuss the merits of this idea.

▶ **10.** [*20*] Suppose a chi-square test is done by making n observations, and the value V is obtained. Now we repeat the test on these same n observations over again (getting, of course, the same results), and we put together the data from both tests, regarding it as a single chi-square test with $2n$ observations. (This procedure violates the text's stipulation that all of the observations must be independent of one another.) How is the second value of V related to the first one?

11. [*10*] Solve exercise 10 substituting the KS test for the chi-square test.

12. [*M28*] Suppose a chi-square test is made on a set of n observations, assuming that p_s is the probability that each observation falls into category s; but suppose that in actual fact the observations have probability $q_s \neq p_s$ of falling into category s. (Cf. exercise 3.) We would, of course, like the chi-square test to detect the fact that the p_s assumption was incorrect. Show that this *will* happen, if n is large enough. Prove also the analogous result for the KS test.

13. [*M24*] Prove that Eqs. (13) are equivalent to Eqs. (11).

▶ **14.** [*HM26*] Let Z_s be given by Eq. (18). Show directly by using Stirling's approximation that the multinomial probability

$$n! p_1^{Y_1} \ldots p_k^{Y_k} / Y_1! \ldots Y_k! = e^{-V/2} / \sqrt{(2n\pi)^{k-1} p_1 \ldots p_k} + O(n^{-k/2}),$$

if $Z_1, Z_2, \ldots, Z_k$ are bounded as $n \to \infty$. (This idea leads to a proof of the chi-square test that is much closer to "first principles," and requires less handwaving, than the derivation in the text.)

15. [*HM24*] Polar coordinates in two dimensions are conventionally defined by the equations $x = r \cos\theta$ and $y = r \sin\theta$. For the purposes of integration, we have $dx\, dy = r\, dr\, d\theta$. More generally, in n-dimensional space we can let

$$x_k = r \sin\theta_1 \ldots \sin\theta_{k-1} \cos\theta_k, \qquad 1 \leq k < n, \qquad x_n = r \sin\theta_1 \ldots \sin\theta_{n-1}.$$

Show that in this case

$$dx_1\, dx_2 \ldots dx_n = \left| r^{n-1} \sin^{n-2}\theta_1 \ldots \sin\theta_{n-2}\, dr\, d\theta_1 \ldots d\theta_{n-1} \right|.$$

▸ **16.** [*HM35*] Generalize Theorem 1.2.11.3A to find the value of

$$\gamma(x+1, x+z\sqrt{2x}+y)/\Gamma(x+1),$$

for large x and fixed y, z. Disregard terms of the answer that are $O(1/x)$. Use this result to find the approximate solution, t, to the equation

$$\gamma\Big(\frac{\nu}{2}, \frac{t}{2}\Big)\Big/\Gamma\Big(\frac{\nu}{2}\Big) = p,$$

for large ν and fixed p, thereby accounting for the asymptotic formulas indicated in Table 1. [*Hint:* See exercise 1.2.11.3–8.]

17. [*HM26*] Let t be a fixed real number. For $0 \le k \le n$, let

$$P_{nk}(x) = \int_{n-t}^{x} dx_n \int_{n-1-t}^{x_n} dx_{n-1} \ \ldots \int_{k+1-t}^{x_{k+2}} dx_{k+1} \int_{0}^{x_{k+1}} dx_k \ \ldots \int_{0}^{x_2} dx_1;$$

by convention, let $P_{00}(x) = 1$. Prove the following relations:

a) $P_{nk}(x) = \displaystyle\int_{n}^{x+t} dx_n \int_{n-1}^{x_n} dx_{n-1} \ \ldots \int_{k+1}^{x_{k+2}} dx_{k+1} \int_{t}^{x_{k+1}} dx_k \ \ldots \int_{t}^{x_2} dx_1.$

b) $P_{n0}(x) = (x+t)^n/n! - (x+t)^{n-1}/(n-1)!.$

c) $P_{nk}(x) - P_{n(k-1)}(x) = \dfrac{(k-t)^k}{k!} P_{(n-k)0}(x-k)$, if $1 \le k \le n$.

d) Obtain a general formula for $P_{nk}(x)$, and apply it to the evaluation of Eq. (24).

18. [*M20*] Give a "simple" reason why K_n^- has the same probability distribution as K_n^+.

19. [*HM48*] Develop tests, analogous to the Kolomogrov–Smirnov test, for use with multivariate distributions $F(x_1, \ldots, x_r)$ = probability that $(X_1 \le x_1, \ldots, X_r \le x_r)$. (Such procedures could be used, for example, in place of the "serial test" in the next section.)

20. [*HM41*] Deduce further terms of the asymptotic behavior of the KS distribution, extending (28).

21. [*M40*] Although the text states that the KS test should be applied only when $F(x)$ is a continuous distribution function, it is, of course, possible to try to compute K_n^+ and K_n^- even when the distribution has jumps. Analyze the probable behavior of K_n^+ and K_n^- for various discontinuous distributions $F(x)$. Compare the effectiveness of the resulting statistical test with the chi-square test on several samples of random numbers.

22. [*HM46*] Investigate the "improved" KS test suggested in the answer to exercise 6.

23. [*M22*] (T. Gonzalez, S. Sahni, and W. R. Franta.) (a) Suppose that the maximum value in formula (13) for the KS statistic K_n^+ occurs at a given index j where $\lfloor n\,F(X_j) \rfloor = k$. Prove that $F(X_j) = \max_{1 \le i \le n} \{\, F(X_i) \mid \lfloor n\,F(X_i) \rfloor = k \,\}$. (b) Design an algorithm that calculates K_n^+ and K_n^- in $O(n)$ steps (without sorting).

▶ **24.** [*40*] Experiment with various probability distributions (p, q, r) on three categories, where $p+q+r=1$, by computing the exact distribution of the chi-square statistic V for various n, thereby determining how accurate an approximation the chi-square distribution with two degrees of freedom really is.

3.3.2. Empirical Tests

In this section we shall discuss ten kinds of specific tests that have been applied to sequences in order to investigate their randomness. The discussion of each test has two parts: (a) a "plug-in" description of how to perform the test; and (b) a study of the theoretical basis for the test. (Readers lacking mathematical training may wish to skip over the theoretical discussions. Conversely, mathematically-inclined readers may find the associated theory quite interesting, even if they never intend to test random number generators, since some instructive combinatorial questions are involved here.)

Each test is applied to a sequence

$$\langle U_n \rangle = U_0, U_1, U_2, \ldots \tag{1}$$

of real numbers, which purports to be independently and uniformly distributed between zero and one. Some of the tests are designed primarily for integer-valued sequences, instead of the real-valued sequence (1). In this case, the auxiliary sequence

$$\langle Y_n \rangle = Y_0, Y_1, Y_2, \ldots, \tag{2}$$

which is defined by the rule

$$Y_n = \lfloor dU_n \rfloor, \tag{3}$$

is used instead. This is a sequence of integers that purports to be independently and uniformly distributed between 0 and $d-1$. The number d is chosen for convenience; for example, we might have $d = 64 = 2^6$ on a binary computer, so that Y_n represents the six most significant bits of the binary representation of U_n. The value of d should be large enough so that the test is meaningful, but not so large that the test becomes impracticably difficult to carry out.

The quantities U_n, Y_n, and d will have the above significance throughout this section, although the value of d will probably be different in different tests.

A. Equidistribution test (Frequency test). The first requirement that sequence (1) must meet is that its numbers are, in fact, uniformly distributed between zero and one. There are two ways to make this test: (a) Use the Kolmogorov–Smirnov test, with $F(x) = x$ for $0 \le x \le 1$. (b) Let d be a convenient number, e.g., 100 on a decimal computer, 64 or 128 on a binary computer, and use the sequence (2) instead of (1). For each integer r, $0 \le r < d$, count the number of times that $Y_j = r$ for $0 \le j < n$, and then apply the chi-square test using $k = d$ and probability $p_s = 1/d$ for each category.

The theory behind this test has been covered in Section 3.3.1.

The equanimity of your average tosser of coins depends upon a law . . . which ensures that he will not upset himself by losing too much nor upset his opponent by winning too often.

—TOM STOPPARD, *Rosencrantz & Guildenstern are Dead* (1966)

B. Serial test. More generally, we want pairs of successive numbers to be uniformly distributed in an independent manner. The sun comes up just about as often as it goes down, in the long run, but this doesn't make its motion random.

To carry out the serial test, we simply count the number of times that the pair $(Y_{2j}, Y_{2j+1}) = (q, r)$ occurs, for $0 \le j < n$; these counts are to be made for each pair of integers (q, r) with $0 \le q, r < d$, and the chi-square test is applied to these $k = d^2$ categories with probability $1/d^2$ in each category. As with the equidistribution test, d may be any convenient number, but it will be somewhat smaller than the values suggested above since a valid chi-square test should have n large compared to k (say $n \ge 5d^2$ at least).

Clearly we can generalize this test to triples, quadruples, etc., instead of pairs (see exercise 2); however, the value of d must then be severely reduced in order to avoid having too many categories. When quadruples and larger numbers of adjacent elements are considered, we therefore make use of less exact tests such as the poker test or the maximum test described below.

Note that $2n$ numbers of the sequence (2) are used in this test in order to make n observations. It would be a mistake to perform the serial test on the pairs (Y_0, Y_1), (Y_1, Y_2), $\ldots$, (Y_{n-1}, Y_n); can the reader see why? We might perform another serial test on the pairs (Y_{2j+1}, Y_{2j+2}), and expect the sequence to pass both tests. Alternatively, I. J. Good has proved that if d is prime, and if the pairs (Y_0, Y_1), (Y_1, Y_2), $\ldots$, (Y_{n-1}, Y_n) are used, and if we use the usual chi-square method to compute both the statistics V_2 for the serial test and V_1 for the frequency test on $Y_0, \ldots, Y_{n-1}$ with the same value of d, then $V_2 - 2V_1$ should have the chi-square distribution with $(d-1)^2$ degrees of freedom when n is large. (See *Proc. Cambridge Phil. Soc.* **49** (1953), 276–284; *Annals Math. Stat.* **28** (1957), 262–264.)

C. Gap test. Another test is used to examine the length of "gaps" between occurrences of U_j in a certain range. If α and β are two real numbers with $0 \le \alpha < \beta \le 1$, we want to consider the lengths of consecutive subsequences $U_j, U_{j+1}, \ldots, U_{j+r}$ in which U_{j+r} lies between α and β but the other U's do not. (This subsequence of $r+1$ numbers represents a gap of length r.)

Algorithm G (*Data for gap test*). The following algorithm, applied to the sequence (1) for any given values of α and β, counts the number of gaps of lengths $0, 1, \ldots, t-1$ and the number of gaps of length $\ge t$, until n gaps have been tabulated.

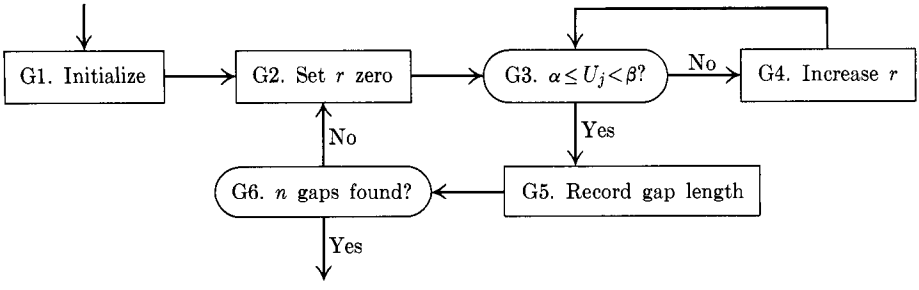


Fig. 6. Gathering data for the gap test. (Algorithms for the "coupon-collector's test" and the "run test" are similar.)

G1. [Initialize.] Set $j \leftarrow -1$, $s \leftarrow 0$, and set COUNT$[r] \leftarrow 0$ for $0 \le r \le t$.

G2. [Set r zero.] Set $r \leftarrow 0$.

G3. [$\alpha \le U_j < \beta$?] Increase j by 1. If $U_j \ge \alpha$ and $U_j < \beta$, go to step G5.

G4. [Increase r.] Increase r by one, and return to step G3.

G5. [Record gap length.] (A gap of length r has now been found.) If $r \ge t$, increase COUNT$[t]$ by one, otherwise increase COUNT$[r]$ by one.

G6. [n gaps found?] Increase s by one. If $s < n$, return to step G2. ∎

After this algorithm has been performed, the chi-square test is applied to the $k = t + 1$ values of COUNT[0], COUNT[1], ..., COUNT$[t]$, using the following probabilities:

$$p_0 = p, \qquad p_1 = p(1-p), \qquad p_2 = p(1-p)^2, \qquad \ldots,$$
$$p_{t-1} = p(1-p)^{t-1}, \qquad p_t = (1-p)^t. \tag{4}$$

Here $p = \beta - \alpha$, the probability that $\alpha \le U_j < \beta$. The values of n and t are to be chosen, as usual, so that each of the values of COUNT$[r]$ is expected to be 5 or more, preferably more.

The gap test is often applied with $\alpha = 0$ or $\beta = 1$ in order to omit one of the comparisons in step G3. The special cases $(\alpha, \beta) = (0, \frac{1}{2})$ or $(\frac{1}{2}, 1)$ give rise to tests that are sometimes called "runs above the mean" and "runs below the mean," respectively.

The probabilities in Eq. (4) are easily deduced, so this derivation is left to the reader. Note that the gap test as described above observes the lengths of n *gaps*; it does not observe the gap lengths among n *numbers*. If the sequence $\langle U_n \rangle$ is sufficiently nonrandom, Algorithm G may not terminate. Other gap tests that examine a fixed number of U's have also been proposed (see exercise 5).

D. Poker test (Partition test). The "classical" poker test considers n groups of five successive integers, $(Y_{5j}, Y_{5j+1}, \ldots, Y_{5j+4})$ for $0 \le j < n$, and observes which of the following seven patterns is matched by each quintuple:

All different:	*abcde*	Full house:	*aaabb*
One pair:	*aabcd*	Four of a kind:	*aaaab*
Two Pairs:	*aabbc*	Five of a kind:	*aaaaa*
Three of a kind:	*aaabc*		

A chi-square test is based on the number of quintuples in each category.

It is reasonable to ask for a somewhat simpler version of this test, to facilitate the programming involved. A good compromise would simply be to count the number of *distinct* values in the set of five. We would then have five categories:

5 different = all different;
4 different = one pair;
3 different = two pairs, or three of a kind;
2 different = full house, or four of a kind;
1 different = five of a kind.

This breakdown is easier to determine systematically, and the test is nearly as good.

In general we can consider n groups of k successive numbers, and we can count the number of k-tuples with r different values. A chi-square test is then made, using the probability

$$p_r = \frac{d(d-1)\ldots(d-r+1)}{d^k} \left\{ {k \atop r} \right\} \tag{5}$$

that there are r different. (The Stirling numbers $\left\{ {k \atop r} \right\}$ are defined in Section 1.2.6, and they can readily be computed using the formulas given there.) Since the probability p_r is very small when $r = 1$ or 2, we generally lump a few categories of low probability together before the chi-square test is applied.

To derive the proper formula for p_r, we must count how many of the d^k k-tuples of numbers between 0 and $d-1$ have exactly r different elements, and divide the total by d^k. Since $d(d-1)\ldots(d-r+1)$ is the number of ordered choices of r things from a set of d objects, we need only show that $\left\{ {k \atop r} \right\}$ is the number of ways to partition a set of k elements into exactly r parts. Therefore exercise 1.2.6–64 completes the derivation of Eq. (5).

E. Coupon collector's test. This test is related to the poker test somewhat as the gap test is related to the frequency test. The sequence Y_0, Y_1, ... is used, and we observe the lengths of segments $Y_{j+1}, Y_{j+2}, \ldots, Y_{j+r}$ required to get a "complete set" of integers from 0 to $d-1$. Algorithm C describes this precisely:

Algorithm C (*Data for coupon collector's test*). Given a sequence of integers Y_0, Y_1, ..., with $0 \le Y_j < d$, this algorithm counts the lengths of n consecutive "coupon collector" segments. At the conclusion of the algorithm, COUNT$[r]$ is the number of segments with length r, for $d \le r < t$, and COUNT$[t]$ is the number of segments with length $\ge t$.

C1. [Initialize.] Set $j \leftarrow -1$, $s \leftarrow 0$, and set COUNT$[r] \leftarrow 0$ for $d \le r \le t$.

C2. [Set q, r zero.] Set $q \leftarrow r \leftarrow 0$, and set OCCURS$[k] \leftarrow 0$ for $0 \le k < d$.

C3. [Next observation.] Increase r and j by 1. If OCCURS$[Y_j] \neq 0$, repeat this step.

C4. [Complete set?] Set OCCURS$[Y_j] \leftarrow 1$ and $q \leftarrow q + 1$. (The subsequence observed so far contains q distinct values; if $q = d$, we therefore have a complete set.) If $q < d$, return to step C3.

C5. [Record the length.] If $r \ge t$, increase COUNT$[t]$ by one, otherwise increase COUNT$[r]$ by one.

C6. [n found?] Increase s by one. If $s < n$, return to step C2. ∎

For an example of this algorithm, see exercise 7. We may think of a boy collecting d types of coupons, which are randomly distributed in his breakfast cereal boxes; he must keep eating more cereal until he has one coupon of each type.

A chi-square test is to be applied to COUNT$[d]$, COUNT$[d+1]$, ..., COUNT$[t]$, with $k = t - d + 1$, after Algorithm C has counted n lengths. The corresponding probabilities are

$$p_r = \frac{d!}{d^r}\left\{{r-1 \atop d-1}\right\}, \quad d \le r < t; \qquad p_t = 1 - \frac{d!}{d^{t-1}}\left\{{t-1 \atop d}\right\}. \tag{6}$$

To derive these probabilities, we simply note that if q_r denotes the probability that a subsequence of length r is *incomplete*, then

$$q_r = 1 - \frac{d!}{d^r}\left\{{r \atop d}\right\}$$

by Eq. (5); for this means we have an r-tuple of elements that do not have all d different values. Then (6) follows from the relations $p_r = q_{r-1} - q_r$ for $d \le r < t$; $p_t = q_{t-1}$.

For formulas that arise in connection with *generalizations* of the coupon collector's test, see exercises 9 and 10 and also the paper by Hermann von Schelling, *AMM* **61** (1954), 306–311.

F. Permutation test. Divide the input sequence into n groups of t elements each, that is, $(U_{jt}, U_{jt+1}, \ldots, U_{jt+t-1})$ for $0 \le j < n$. The elements in each group can have $t!$ possible relative orderings; the number of times each ordering appears is counted, and a chi-square test is applied with $k = t!$ and with probability $1/t!$ for each ordering.

For example, if $t = 3$ we would have six possible categories, according to whether $U_{3j} < U_{3j+1} < U_{3j+2}$ or $U_{3j} < U_{3j+2} < U_{3j+1}$ or $\cdots$ or $U_{3j+2} < U_{3j+1} < U_{3j}$. We assume in this test that equality between U's does not occur; such an assumption is justified, for the probability that two U's are equal is zero.

A convenient way to perform the permutation test on a computer makes use of the following algorithm, which is of interest in itself:

Algorithm P (*Analyze a permutation*). Given a sequence of distinct elements $(U_1, \ldots, U_t)$, we compute an integer $f(U_1, \ldots, U_t)$ such that

$$0 \le f(U_1, \ldots, U_t) < t!,$$

and $f(U_1, \ldots, U_t) = f(V_1, \ldots, V_t)$ if and only if $(U_1, \ldots, U_t)$ and $(V_1, \ldots, V_t)$ have the same relative ordering.

P1. [Initialize.] Set $r \leftarrow t$, $f \leftarrow 0$. (During this algorithm we will have $0 \le f < t!/r!$.)

P2. [Find maximum.] Find the maximum of $\{U_1, \ldots, U_r\}$, and suppose that U_s is the maximum. Set $f \leftarrow r \cdot f + s - 1$.

P3. [Exchange.] Exchange $U_r \leftrightarrow U_s$.

P4. [Decrease r.] Decrease r by one. If $r > 1$, return to step P2. ∎

Note that the sequence $(U_1, \ldots, U_t)$ will have been sorted into ascending order when this algorithm stops. To prove that the result f uniquely characterizes the *initial* order of $(U_1, \ldots, U_t)$, we note that Algorithm P can be run backwards: For $r = 2, 3, \ldots, t$, set $s \leftarrow f \bmod r$, $f \leftarrow \lfloor f/r \rfloor$, and exchange $U_r \, U_s$. It is easy to see that this will undo the effects of steps P2–P4; hence no two permutations can yield the same value of f, and Algorithm P performs as advertised.

The essential idea that underlies Algorithm P is a mixed-radix representation called the "factorial number system": Every integer in the range $0 \le f < t!$ can be uniquely written in the form

$$\begin{aligned} f &= \big(\ldots(c_{t-1} \times (t-1) + c_{t-2}) \times (t-2) + \cdots + c_2\big) \times 2 + c_1 \\ &= (t-1)!\,c_{t-1} + (t-2)!\,c_{t-2} + \cdots + 2!\,c_2 + 1!\,c_1 \end{aligned} \tag{7}$$

where the "digits" c_j are integers satisfying

$$0 \le c_j \le j, \qquad \text{for } 1 \le j < t. \tag{8}$$

In Algorithm P, $c_{r-1} = s - 1$ when step P2 is performed for a given value of r.

G. Run test. A sequence may also be tested for "runs up" and "runs down." This means we examine the length of *monotone* subsequences of the original sequence, i.e., segments that are increasing or decreasing.

As an example of the precise definition of a run, consider the sequence of ten numbers "1298536704"; putting a vertical line at the left and right and between X_j and X_{j+1} whenever $X_j > X_{j+1}$, we obtain

$$|1\ 2\ 9|8|5|3\ 6\ 7|0\ 4|, \tag{9}$$

which displays the "runs up": there is a run of length 3, followed by two runs of length 1, followed by another run of length 3, followed by a run of length 2. The algorithm of exercise 12 shows how to tabulate the length of "runs up."

Unlike the gap test and the coupon collector's test (which are in many other respects similar to this test), *we should not apply a chi-square test to the above data*, since adjacent runs are *not* independent. A long run will tend to be followed by a short run, and conversely. This lack of independence is enough to invalidate a straightforward chi-square test. Instead, the following statistic may be computed, when the run lengths have been determined as in exercise 12:

$$V = \frac{1}{n} \sum_{1 \le i,j \le 6} (\texttt{COUNT}[i] - nb_i)(\texttt{COUNT}[j] - nb_j)a_{ij}, \tag{10}$$

where n is the length of the sequence, and the coefficients a_{ij} and b_i are equal to

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{pmatrix} = \begin{pmatrix} 4529.4 & 9044.9 & 13568 & 18091 & 22615 & 27892 \\ 9044.9 & 18097 & 27139 & 36187 & 45234 & 55789 \\ 13568 & 27139 & 40721 & 54281 & 67852 & 83685 \\ 18091 & 36187 & 54281 & 72414 & 90470 & 111580 \\ 22615 & 45234 & 67852 & 90470 & 113262 & 139476 \\ 27892 & 55789 & 83685 & 111580 & 139476 & 172860 \end{pmatrix} \tag{11}$$

$$(b_1\ \ b_2\ \ b_3\ \ b_4\ \ b_5\ \ b_6) = (\tfrac{1}{6}\ \ \tfrac{5}{24}\ \ \tfrac{11}{120}\ \ \tfrac{19}{720}\ \ \tfrac{29}{5040}\ \ \tfrac{1}{840}).$$

(The values of a_{ij} shown here are approximate only; exact values may be obtained by using formulas derived below.) *The statistic V in* (10) *should have the chi-square distribution with six* (not five) *degrees of freedom*, when n is large. The value of n should be, say, 4000 or more. The same test can be applied to "runs down."

A vastly simpler and more practical run test appears in exercise 14, so a reader who is interested only in testing random number generators should skip the next few pages and go on to the "maximum-of-t test" after looking at exercise 14. On the other hand it is instructive from a mathematical standpoint to see how a complicated run test with interdependent runs can be treated, so we shall now digress for a moment.

Given any permutation on n elements, let $Z_{pi} = 1$ if position i is the beginning of an ascending run of length p or more, and let $Z_{pi} = 0$ otherwise. For example, consider the permutation (9) with $n = 10$; we have

$$Z_{11} = Z_{21} = Z_{31} = Z_{14} = Z_{15} = Z_{16} = Z_{26} = Z_{36} = Z_{19} = Z_{29} = 1,$$

and all other Z's are zero. With this notation,

$$R'_p = Z_{p1} + Z_{p2} + \cdots + Z_{pn} \tag{12}$$

is the number of runs of length $\geq p$, and

$$R_p = R'_p - R'_{p+1} \tag{13}$$

is the number of runs of length p exactly. Our goal is to compute the mean value of R_p, and also the *covariance*

$$\operatorname{covar}(R_p, R_q) = \operatorname{mean}\big((R_p - \operatorname{mean}(R_p))(R_q - \operatorname{mean}(R_q))\big),$$

which measures the interdependence of R_p and R_q. These mean values are to be computed as the average over the set of all $n!$ permutations.

Equations (12) and (13) show that the answers can be expressed in terms of the mean values of Z_{pi} and of $Z_{pi}Z_{qj}$, so as the first step of the derivation we obtain the following results (assuming that $i < j$):

$$\frac{1}{n!}\sum Z_{pi} = \begin{cases} (p + \delta_{i1})/(p+1)!, & \text{if } i \leq n - p + 1; \\ 0, & \text{otherwise.} \end{cases}$$

$$\frac{1}{n!}\sum Z_{pi}Z_{pj} = \begin{cases} (p + \delta_{i1})q/(p+1)!(q+1)!, & \\ & \text{if } i + p < j \leq n - q + 1; \\ (p + \delta_{i1})/(p+1)!q! - (p+q+\delta_{i1})/(p+q+1)!, & \\ & \text{if } i + p = j \leq n - q + 1; \\ 0, & \text{otherwise.} \end{cases} \tag{14}$$

The $\sum$-signs stand for a summation over all possible permutations. To illustrate the calculations involved here, we will work the most difficult case, when $i+p = j \leq n - q + 1$, and when $i > 1$. Note that $Z_{pi}Z_{qj}$ is either zero or one, so the summation consists of counting all permutations $U_1U_2 \ldots U_n$ for which $Z_{pi} = Z_{qj} = 1$, that is, all permutations such that

$$U_{i-1} > U_i < \cdots < U_{i+p-1} > U_{i+p} < \cdots < U_{i+p+q-1}. \tag{15}$$

The number of such permutations may be enumerated as follows: there are $\binom{n}{p+q+1}$ ways to choose the elements for the positions indicated in (15); there

are

$$(p+q+1)\binom{p+q}{p}-\binom{p+q+1}{p+1}-\binom{p+q+1}{1}+1 \tag{16}$$

ways to arrange them in the order (15), as shown in exercise 13; and there are $(n-p-q-1)!$ ways to arrange the remaining elements. Thus there are $\binom{n}{p+q+1}(n-p-q-1)!$ times (16) ways in all, and dividing by $n!$ we get the desired formula.

From relations (14) a rather lengthy calculation leads to

$$\begin{aligned}\operatorname{mean}(R'_p) &= \operatorname{mean}(Z_{p1}+\cdots+Z_{pn})\\ &= (n+1)p/(p+1)!-(p-1)/p!, \qquad 1\le p\le n;\end{aligned} \tag{17}$$

$$\begin{aligned}\operatorname{covar}(R'_p,R'_q) &= \operatorname{mean}(R'_p\,R'_q)-\operatorname{mean}(R'_p)\operatorname{mean}(R'_q)\\ &= \sum_{1\le i,j\le n}\frac{1}{n!}\sum Z_{pi}Z_{pj}-\operatorname{mean}(R'_p)\operatorname{mean}(R'_q)\\ &= \begin{cases}\operatorname{mean}(R'_t)+f(p,q,n), & \text{if } p+q\le n,\\ \operatorname{mean}(R'_t)-\operatorname{mean}(R'_p)\operatorname{mean}(R'_q), & \text{if } p+q>n,\end{cases}\end{aligned} \tag{18}$$

where $t=\max(p,q)$, $s=p+q$, and

$$\begin{aligned}f(p,q,n) = (n+1)\left(\frac{s(1-pq)+pq}{(p+1)!(q+1)!}-\frac{2s}{(s+1)!}\right)+2\left(\frac{s-1}{s!}\right)\\ +\frac{(s^2-s-2)pq-s^2-p^2q^2+1}{(p+1)!(q+1)!}.\end{aligned} \tag{19}$$

This expression for the covariance is unfortunately quite complicated, but it is necessary for a successful run test as described above. From these formulas it is easy to compute

$$\begin{aligned}\operatorname{mean}(R_p) &= \operatorname{mean}(R'_p)-\operatorname{mean}(R'_{p+1}),\\ \operatorname{covar}(R_p,R'_q) &= \operatorname{covar}(R'_p,R'_q)-\operatorname{covar}(R'_{p+1},R'_q),\\ \operatorname{covar}(R_p,R_q) &= \operatorname{covar}(R_p,R'_q)-\operatorname{covar}(R_p,R'_{q+1}).\end{aligned} \tag{20}$$

In *Annals Math. Stat.* **15** (1944), 163–165, J. Wolfowitz proved that the quantities R_1, R_2, ..., R_{t-1}, R'_t become normally distributed as $n\to\infty$, subject to the mean and covariance expressed above; this implies that the following test for runs is valid: Given a sequence of n random numbers, compute the number of runs R_p of length p for $1\le p<t$, and also the number of runs R'_t of length t or more. Let

$$\begin{aligned}Q_1 = R_1-\operatorname{mean}(R_1), \quad \ldots, \quad Q_{t-1} = R_{t-1}-\operatorname{mean}(R_{t-1}),\\ Q_t = R'_t-\operatorname{mean}(R'_t).\end{aligned} \tag{21}$$

Form the matrix C of the covariances of the R's; for example, $C_{13} = \operatorname{covar}(R_1, R_3)$, while $C_{1t} = \operatorname{covar}(R_1, R'_t)$. When $t = 6$, we have

$$C = nC_1 + C_2$$

$$= n\begin{pmatrix} \frac{23}{180} & \frac{-7}{360} & \frac{-5}{336} & \frac{-433}{60480} & \frac{-13}{5670} & \frac{-121}{181440} \\ \frac{-7}{360} & \frac{2843}{20160} & \frac{-989}{20160} & \frac{-7159}{362880} & \frac{-10019}{1814400} & \frac{-1303}{907200} \\ \frac{-5}{336} & \frac{-989}{20160} & \frac{54563}{907200} & \frac{-21311}{1814400} & \frac{-62369}{19958400} & \frac{-7783}{9979200} \\ \frac{-433}{60480} & \frac{-7159}{362880} & \frac{-21311}{1814400} & \frac{886657}{39916800} & \frac{-257699}{239500800} & \frac{-62611}{239500800} \\ \frac{-13}{5670} & \frac{-10019}{1814400} & \frac{-62369}{19958400} & \frac{-257699}{239500800} & \frac{29874811}{5448643200} & \frac{-1407179}{21794572800} \\ \frac{-121}{181440} & \frac{-1303}{907200} & \frac{-7783}{9979200} & \frac{-62611}{239500800} & \frac{-1407179}{21794572800} & \frac{2134697}{1816214400} \end{pmatrix}$$

$$+ \begin{pmatrix} \frac{83}{180} & \frac{-29}{180} & \frac{-11}{210} & \frac{-41}{12096} & \frac{91}{25920} & \frac{41}{18144} \\ \frac{-29}{180} & \frac{-305}{4032} & \frac{319}{20160} & \frac{2557}{72576} & \frac{10177}{604800} & \frac{413}{64800} \\ \frac{-11}{210} & \frac{319}{20160} & \frac{-58747}{907200} & \frac{19703}{604800} & \frac{239471}{19958400} & \frac{39517}{9979200} \\ \frac{-41}{12096} & \frac{2557}{72576} & \frac{19703}{604800} & \frac{-220837}{4435200} & \frac{1196401}{239500800} & \frac{360989}{239500800} \\ \frac{91}{25920} & \frac{10177}{604800} & \frac{239471}{19958400} & \frac{1196401}{239500800} & \frac{-139126639}{7264857600} & \frac{4577641}{10897286400} \\ \frac{41}{18144} & \frac{413}{64800} & \frac{39517}{9979200} & \frac{360989}{239500800} & \frac{4577641}{10897286400} & \frac{-122953057}{21794572800} \end{pmatrix} \tag{22}$$

if $n \geq 12$. Now form $A = (a_{ij})$, the inverse of the matrix C, and compute $\sum_{1 \leq i,j \leq t} Q_i Q_j a_{ij}$. The result for large n should have approximately the chi-square distribution with t degrees of freedom.

The matrix (11) given earlier is the inverse of C_1 to five significant figures. When n is large, A will be approximately $(1/n)C_1^{-1}$; a test with $n = 100$ showed that the entries a_{ij} in (11) were each about 1 percent lower than the true values obtained by inverting (22).

H. Maximum-of-t test. For $0 \leq j < n$, let $V_j = \max(U_{tj}, U_{tj+1}, \ldots, U_{tj+t-1})$. Now apply the Kolmogorov–Smirnov test to the sequence V_0, V_1, $\ldots$, V_{n-1}, with the distribution function $F(x) = x^t$, $0 \leq x \leq 1$. Alternatively, apply the equidistribution test to the sequence V_0^t, V_1^t, $\ldots$, V_{n-1}^t.

To verify this test, we must show that the distribution function for the V_j is $F(x) = x^t$. The probability that $\max(U_1, U_2, \ldots, U_t) \leq x$ is the probability that $U_1 \leq x$ and $U_2 \leq x$ and $\ldots$ and $U_t \leq x$, which is the product of the individual probabilities, namely $xx \ldots x = x^t$.

I. Collision test. Chi-square tests can be made only when there is a nontrivial number of items expected in each category. But there is another kind of test that can be used when the number of categories is much larger than the number of observations; this test is related to "hashing," an important method for information retrieval that we shall study in Chapter 6.

Suppose we have m urns and we throw n balls at random into those urns, where m is much greater than n. Most of the balls will land in urns that were previously empty, but if a ball falls into an urn that already contains at least one ball we say that a "collision" has occurred. The collision test counts the number of collisions, and a generator passes this test if it doesn't induce too many or too few collisions.

To fix the ideas, suppose $m = 2^{20}$ and $n = 2^{14}$. Then each urn will receive only one 64th of a ball, on the average. The probability that a given urn will contain exactly k balls is $p_k = \binom{n}{k} m^{-k}(1 - m^{-1})^{n-k}$, so the expected number of collisions per urn is $\sum_{k\geq 1}(k-1)p_k = \sum_{k\geq 0} kp_k - \sum_{k\geq 1} p_k = n/m - 1 + p_0$. Since $p_0 = (1 - m^{-1})^n = 1 - n/m + \binom{n}{2}m^{-2} +$ smaller terms, we find that the average total number of collisions taken over all m urns is very slightly less than $n^2/2m = 128$.

We can use the collision test to rate a random number generator in a large number of dimensions. For example, when $m = 2^{20}$ and $n = 2^{14}$ we can test the 20-dimensional randomness of a number generator by letting $d = 2$ and forming 20-dimensional vectors $V_j = (Y_{20j}, Y_{20j+1}, \ldots, Y_{20j+19})$ for $0 \leq j < n$. It suffices to keep a table of $m = 2^{20}$ bits to determine collisions, one bit for each possible value of the vector V_j; on a computer with 32 bits per word, this amounts to 2^{15} words. Initially all 2^{20} bits of this table are cleared to zero; then for each V_j, if the corresponding bit is already 1 we record a collision, otherwise we set the bit to 1. This test can also be used in 10 dimensions with $d = 4$, and so on.

To decide if the test is passed, we can use the following table of percentage points when $m = 2^{20}$ and $n = 2^{14}$:

collisions $\leq$	101	108	119	126	134	145	153
with probability	.009	.043	.244	.476	.742	.946	.989

The theory underlying these probabilities is the same we used in the poker test, Eq. (5); the probability that c collisions occur is the probability that $n - c$ urns are occupied, namely

$$\frac{m(m-1)\ldots(m-n+c+1)}{m^n} \left\{ {n \atop n-c} \right\}.$$

Although m and n are very large, it is not difficult to compute these probabilities using the following method:

Algorithm S (*Percentage points for collision test*). Given m and n, this algorithm determines the distribution of the number of collisions that occur when n balls are scattered into m urns. An auxiliary array $A[0]$, $A[1]$, ..., $A[n]$ of floating point numbers is used for the computation; actually $A[j]$ will be nonzero only for $j_0 \leq j \leq j_1$, and $j_1 - j_0$ will be at most of order $\log n$, so it would be possible to get by with considerably less storage.

S1. [Initialize.] Set $A[j] \leftarrow 0$ for $0 \le j \le n$; then set $A[1] \leftarrow 1$ and $j_0 \leftarrow j_1 \leftarrow 1$. Then do step S2 exactly $n - 1$ times and go on to step S3.

S2. [Update probabilities.] (Each time we do this step it corresponds to tossing a ball into an urn; $A[j]$ represents the probability that exactly j of the urns are occupied.) Set $j_1 \leftarrow j_1 + 1$. Then for $j \leftarrow j_1, j_1 - 1, \ldots, j_0$ (in this order), set $A[j] \leftarrow (j/m)A[j] + ((1 + 1/m) - (j/m))A[j-1]$. If $A[j]$ has become very small as a result of this calculation, say $A[j] < 10^{-20}$, set $A[j] \leftarrow 0$; and in such a case, if $j = j_1$ decrease j_1 by 1, or if $j = j_0$ increase j_0 by 1.

S3. [Compute the answers.] In this step we make use of an auxiliary table $(T_1, T_2, \ldots, T_{\text{tmax}}) = (.01, .05, .25, .50, .75, .95, .99, 1.00)$ containing the specified percentage points of interest. Set $p \leftarrow 0$, $t \leftarrow 1$, and $j \leftarrow j_0 - 1$. Do the following iteration until $t = \text{tmax}$: Increase j by 1, and set $p \leftarrow p + A[j]$; then if $p > T_t$, output $n - j - 1$ and $1 - p$ (meaning that with probability $1 - p$ there are at most $n - j - 1$ collisions) and repeatedly increase t by 1 until $p \le T_t$. ∎

J. Serial correlation test. We may also compute the following statistic:

$$C = \frac{n(U_0U_1 + U_1U_2 + \cdots + U_{n-2}U_{n-1} + U_{n-1}U_0) - (U_0 + U_1 + \cdots + U_{n-1})^2}{n(U_0^2 + U_1^2 + \cdots + U_{n-1}^2) - (U_0 + U_1 + \cdots + U_{n-1})^2}. \tag{23}$$

This is the "serial correlation coefficient," which is a measure of the amount U_{j+1} depends on U_j.

Correlation coefficients appear frequently in statistics; if we have n quantities $U_0, U_1, \ldots, U_{n-1}$ and n others $V_0, V_1, \ldots, V_{n-1}$, the correlation coefficient between them is defined to be

$$C = \frac{n\sum(U_jV_j) - (\sum U_j)(\sum V_j)}{\sqrt{(n\sum U_j^2 - (\sum U_j)^2)(n\sum V_j^2 - (\sum V_j)^2)}}. \tag{24}$$

All summations in this formula are to be taken over the range $0 \le j < n$; Eq. (23) is the special case $V_j = U_{(j+1) \bmod n}$. (*Note:* The denominator of (24) is zero when $U_0 = U_1 = \cdots = U_{n-1}$ or $V_0 = V_1 = \cdots = V_{n-1}$; we exclude this case from discussion.)

A correlation coefficient always lies between -1 and $+1$. When it is zero or very small, it indicates that the quantities U_j and V_j are (relatively speaking) independent of each other, but when the correlation coefficient is ± 1 it indicates total linear dependence; in fact $V_j = \alpha \pm \beta U_j$ for all j in such a case, for some constants α and β. (See exercise 17.)

Therefore it is desirable to have C in Eq. (23) close to zero. In actual fact, since U_0U_1 is not completely independent of U_1U_2, the serial correlation

coefficient is not expected to be *exactly* zero. (See exercise 18.) A "good" value of C will be between $\mu_n - 2\sigma_n$ and $\mu_n + 2\sigma_n$, where

$$\mu_n = \frac{-1}{n-1}, \qquad \sigma_n = \frac{1}{n-1}\sqrt{\frac{n(n-3)}{n+1}}, \qquad n > 2. \tag{25}$$

We expect C to be between these limits about 95 percent of the time.

Equations (25) are only conjectured at this time, since the exact distribution of C is not known when the U's are uniformly distributed. For the theory when the U's have the *normal* distribution, see the paper by Wilfrid J. Dixon, *Annals Math. Stat.* **15** (1944), 119–144. Empirical evidence suggests that we may safely use the formulas for the mean and standard deviation that have been derived from the assumption of the normal distribution, without much error; these are the values that have been listed in (25). It is known that $\lim_{n\to\infty} \sqrt{n}\sigma_n = 1$; cf. the article by Anderson and Walker, *Annals Math. Stat.* **35** (1964), 1296–1303, where more general results about serial correlations of *dependent* sequences are derived.

Instead of simply computing the correlation coefficient between the observations $(U_0, U_1, \ldots, U_{n-1})$ and their immediate successors $(U_1, \ldots, U_{n-1}, U_0)$, we can also compute it between $(U_0, U_1, \ldots, U_{n-1})$ and any cyclically shifted sequence $(U_q, \ldots, U_{n-1}, U_0, \ldots, U_{q-1})$; the cyclic correlations should be small for $0 < q < n$. A straightforward computation of Eq. (24) for all q would require about n^2 multiplications, but it is actually possible to compute all the correlations in only $O(n \log n)$ steps by using "fast Fourier transforms." (See Section 4.6.4; cf. also L. P. Schmid, *CACM* **8** (1965), 115.)

K. Tests on subsequences. It frequently happens that the external program using our random sequence will call for numbers in batches. For example, if the program works with three random variables X, Y, and Z, it may consistently invoke the generation of three random numbers at a time. In such applications it is important that the subsequences consisting of every *third* term of the original sequence be random. If the program requires q numbers at a time, the sequences

$$U_0, U_q, U_{2q}, \ldots; \quad U_1, U_{q+1}, U_{2q+1}, \ldots; \quad \ldots; \quad U_{q-1}, U_{2q-1}, U_{3q-1}, \ldots$$

can each be put through the tests described above for the original sequence U_0, U_1, U_2, ...

Experience with linear congruential sequences has shown that these derived sequences rarely if ever behave less randomly than the original sequence, unless q has a large factor in common with the period length. On a binary computer with m equal to the word size, for example, a test of the subsequences for $q = 8$ will tend to give the poorest randomness for all $q < 16$; and on a decimal computer, $q = 10$ yields the subsequences most likely to be unsatisfactory. (This can be explained somewhat on the grounds of potency, since such values of q will tend to lower the potency.)

L. Historical remarks and further discussion. Statistical tests arose naturally in the course of scientists' efforts to "prove" or "disprove" hypotheses about various observed data. The best known early papers dealing with the testing of artificially generated numbers for randomness are two articles by M. G. Kendall and B. Babington-Smith in the *Journal of the Royal Statistical Society* **101** (1938), 147–166, and in the supplement to that journal, **6** (1939), 51–61. These papers were concerned with the testing of random digits between 0 and 9, rather than random real numbers; for this purpose, the authors discussed the frequency test, serial test, gap test, and poker test, although they misapplied the serial test. Kendall and Babington-Smith also used a variant of the coupon collector's test; the method described in this section was introduced by R. E. Greenwood in *Math. Comp.* **9** (1955), 1–4.

The run test has a rather interesting history. Originally, tests were made on runs up and down at once: a run up would be followed by a run down, then another run up, and so on. Note that the run test and the permutation test do not depend on the uniform distribution of the U's, they depend only on the fact that $U_i = U_j$ occurs with probability zero when $i \neq j$; therefore these tests can be applied to many types of random sequences. The run test in primitive form was originated by J. Bienaymé [*Comptes Rendus* **81** (Paris: Acad. Sciences, 1875), 417–423]. Some sixty years later, W. O. Kermack and A. G. McKendrick published two extensive papers on the subject (*Proc. Royal Society Edinburgh* **57** (1937), 228–240, 332–376]; as an example they stated that Edinburgh rainfall between the years 1785 and 1930 was "entirely random in character" with respect to the run test (although they examined only the mean and standard deviation of the run lengths). Several other people began using the test, but it was not until 1944 that the use of the chi-square method in connection with this test was shown to be incorrect. The paper by H. Levene and J. Wolfowitz in *Annals Math. Stat.* **15** (1944), 58–69, introduced the correct run test (for runs up and down, alternately) and discussed the fallacies in earlier misuses of that test. Separate tests for runs up and runs down, as proposed in the text above, are more suited to computer application, so we have not given the more complex formulas for the alternate-up-and-down case. See the survey paper by D. E. Barton and C. L. Mallows, *Annals Math. Stat.* **36** (1965), 236–260.

Of all the tests we have discussed, the frequency test and the serial correlation test seem to be the weakest, in the sense that nearly all random number generators pass these tests. Theoretical grounds for the weakness of these tests are discussed briefly in Section 3.5 (cf. exercise 3.5–26). The run test, on the other hand, is a rather strong test: the results of exercises 3.3.3–23 and 24 suggest that linear congruential generators tend to have runs somewhat longer than normal if the multiplier is not large enough, so the run test of exercise 14 is definitely to be recommended.

The collision test is also highly recommended, since it has been especially designed to detect the deficiencies of many poor generators that have unfortunately become widespread. This test, which is based on ideas of H. Delgas Christiansen [Inst. Math. Stat. and Oper. Res., Tech. Univ. Denmark (Oct. 1975),

unpublished], is unlike the others in that it was not developed before the advent of computers; it is specifically intended for computer use.

The reader probably wonders, *"Why are there so many tests?"* It has been said that more computer time is spent testing random numbers than using them in applications! This is untrue, although it is possible to go overboard in testing.

The need for making several tests has been amply documented. It has been recorded, for example, that some numbers generated by a variant of the middle-square method have passed the frequency test, gap test, and poker test, yet flunked the serial test. Linear congruential sequences with small multipliers have been known to pass many tests, yet fail on the run test because there are too few runs of length one. The maximum-of-t test has also been used to ferret out some bad generators that otherwise seemed to perform respectably.

Perhaps the main reason for doing extensive testing on random number generators is that people misusing Mr. X's random number generator will hardly ever admit that their programs are at fault: they will blame the generator, until Mr. X can *prove* to them that his numbers are sufficiently random. On the other hand, if the source of random numbers is only for Mr. X's personal use, he might decide not to bother to test them, since the techniques recommended in this chapter have a high probability of being satisfactory.

EXERCISES

1. [*10*] Why should the serial test described in part B be applied to (Y_0, Y_1), (Y_2, Y_3), $\ldots$, (Y_{2n-2}, Y_{2n-1}) instead of to (Y_0, Y_1), (Y_1, Y_2), $\ldots$, (Y_{n-1}, Y_n)?

2. [*10*] State an appropriate way to generalize the serial test to triples, quadruples, etc., instead of pairs.

▶ **3.** [*M20*] How many U's need to be examined in the gap test (Algorithm G) before n gaps have been found, on the average, assuming that the sequence is random? What is the standard deviation of this quantity?

4. [*12*] Prove that the probabilities in (4) are correct for the gap test.

5. [*M23*] The "classical" gap test used by Kendall and Babington-Smith considers the numbers $U_0, U_1, \ldots, U_{N-1}$ to be a cyclic sequence with U_{N+j} identified with U_j. Here N is a fixed number of U's that are to be subjected to the test. If n of the numbers $U_0, \ldots, U_{N-1}$ fall into the range $\alpha \le U_j < \beta$, there are n gaps in the cyclic sequence. Let Z_r be the number of gaps of length r, for $0 \le r < t$, and let Z_t be the number of gaps of length $\ge t$; show that the quantity $V = \sum_{0 \le r \le t} (Z_r - np_r)^2 / np_r$ should have the chi-square distribution with t degrees of freedom, in the limit as N goes to infinity, where p_r is given in Eq. (4).

6. [*40*] (H. Geiringer.) A frequency count of the first 2000 decimal digits in the representation of $e = 2.71828\ldots$ gave a χ^2 value of 1.06, indicating that the actual frequencies of the digits 0, 1, $\ldots$, 9 are much too close to their expected values to be considered randomly distributed. (In fact, $\chi^2 \ge 1.15$ with probability 99.9 percent.) The same test applied to the first 10,000 digits of e gives the reasonable value $\chi^2 = 8.61$; but the fact that the first 2000 digits are so evenly distributed is still surprising. Does the same phenomenon occur in the representation of e to other bases? [See *AMM* **72** (1965), 483–500.]

7. [*08*] Apply the coupon collector's test procedure (Algorithm C) with $d = 3$ and $n = 7$, to the following sequence: 11012210221202020012122010102011 21. What lengths do the seven subsequences have?

▶ **8.** [*M22*] How many U's need to be examined, on the average, in the coupon collector's test (Algorithm C) before n complete sets have been found, assuming that the sequence is random? What is the standard deviation? [*Hint:* See Eq. 1.2.9–28.]

9. [*M21*] Generalize the coupon collector's test so that the search stops as soon as w distinct values have been found, where w is a fixed positive integer less than or equal to d. What probabilities should be used in place of (6)?

10. [*M23*] Solve exercise 8 for the more general coupon collector's test described in exercise 9.

11. [*00*] The "runs up" in a particular permutation are displayed in (9); what are the "runs down" in that permutation?

12. [*20*] Let U_0, U_1, ..., U_{n-1} be n distinct numbers. Write an algorithm that determines the lengths of all ascending runs in the sequence. When your algorithm terminates, COUNT[r] should be the number of runs of length r, for $1 \le r \le 5$, and COUNT[6] should be the number of runs of length 6 or more.

13. [*M23*] Show that (16) is the number of permutations of $p+q+1$ distinct elements having the pattern (15).

▶ **14.** [*M15*] If we "throw away" the element that immediately follows a run, so that when X_j is greater than X_{j+1} we start the next run with X_{j+2}, the run lengths are independent, and a simple chi-square test may be used (instead of the horribly complicated method derived in the text). What are the appropriate run-length probabilities for this simple run test?

15. [*M10*] In the maximum-of-t test, why are V_0^t, V_1^t, ..., V_{n-1}^t supposed to be uniformly distributed between zero and one?

▶ **16.** [*15*] (a) Mr. J. H. Quick (a student) wanted to perform the maximum-of-t test for various values of t. Letting $Z_{jt} = \max(U_j, U_{j+1}, \ldots, U_{j+t-1})$, he found a clever way to go from the sequence $Z_{0(t-1)}$, $Z_{1(t-1)}$, ..., to the sequence Z_{0t}, Z_{1t}, ..., using very little time and space. What was his bright idea?

(b) He decided to modify the maximum-of-t method so that the jth observation would be $\max(U_j, \ldots, U_{j+t-1})$; in other words, he took $V_j = Z_{jt}$ instead of $V_j = Z_{(tj)t}$ as the text says. He reasoned that *all* of the Z's should have the same distribution, so the test is even stronger if each Z_{jt}, $0 \le j < n$, is used instead of just every tth one. But when he tried a chi-square equidistribution test on the values of V_j^t, he got extremely high values of the statistic V, which got even higher as t increased. Why did this happen?

17. [*M25*] (a) Given any numbers $U_0, \ldots, U_{n-1}, V_0, \ldots, V_{n-1}$, let

$$\bar{u} = \frac{1}{n} \sum_{0 \le k < n} U_k, \qquad \bar{v} = \frac{1}{n} \sum_{0 \le k < n} V_k.$$

Let $U'_k = U_k - \bar{u}$, $V'_k = V_k - \bar{v}$. Show that the correlation coefficient C given in Eq. (24) is equal to

$$\sum_{0 \le k < n} U'_k V'_k \Bigg/ \sqrt{\sum_{0 \le k < n} {U'_k}^2} \sqrt{\sum_{0 \le k < n} {V'_k}^2}.$$

(b) Let $C = N/D$, where N and D denote the numerator and denominator of the expression in part (a). Show that $N^2 \le D^2$, hence $-1 \le C \le 1$; and obtain a formula for the difference $D^2 - N^2$. [*Hint:* See exercise 1.2.3–30.]

(c) If $C = \pm 1$, show that $\alpha X_k + \beta Y_k = \tau$, $0 \le k < n$, for some constants α, β, and τ, not all zero.

18. [*M20*] (a) Show that if $n = 2$, the serial correlation coefficient (23) is always equal to -1 (unless the denominator is zero). (b) Similarly, show that when $n = 3$, the serial correlation coefficient always equals $-\frac{1}{2}$. (c) Show that the denominator in (23) is zero if and only if $U_0 = U_1 = \cdots = U_{n-1}$.

19. [*M40*] What are the mean and standard deviation of the serial correlation coefficient (23) when $n = 4$ and the U's are independent and uniformly distributed between zero and one?

20. [*M47*] Find the distribution of the serial correlation coefficient (23), for general n, assuming that the U_j are independent random variables uniformly distributed between zero and one.

21. [*19*] What value of f is computed by Algorithm P if it is presented with the permutation $(1, 2, 9, 8, 5, 3, 6, 7, 0, 4)$?

22. [*18*] For what permutation of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ will Algorithm P produce the value $f = 1024$?

*3.3.3. Theoretical Tests

Although it is always possible to test a random number generator using the methods in the previous section, it is far better to have "*a priori* tests," i.e., theoretical results that tell us in advance how well those tests will come out. Such theoretical results give us much more understanding about the generation methods than empirical, "trial-and-error" results do. In this section we shall study the linear congruential sequences in more detail; if we know what the results of certain tests will be before we actually generate the numbers, we have a better chance of choosing a, m, and c properly.

The development of this kind of theory is quite difficult, although some progress has been made. The results obtained so far are generally for *statistical tests made over the entire period.* Not all statistical tests make sense when they are applied over a full period—for example, the equidistribution test will give results that are too perfect—but the serial test, gap test, permutation test, maximum test, etc. can be fruitfully analyzed in this way. Such studies will detect *global* nonrandomness of a sequence, i.e., improper behavior in very large samples.

The theory we shall discuss is quite illuminating, but it does not eliminate the need for testing local nonrandomness by the methods of Section 3.3.2. Indeed, it appears to be extremely hard to prove anything useful about short subsequences. Only a few theoretical results are known about the behavior of linear congruential sequences over less than a full period; these will be discussed at the end of Section 3.3.4. (See also exercise 18.)

Let us begin with a proof of a simple *a priori* law, for the least complicated case of the permutation test. The gist of our first theorem is that we have $X_{n+1} < X_n$ about half the time, provided that the sequence has high potency.

Theorem P. *Let a, c, and m generate a linear congruential sequence with maximum period; let $b = a - 1$ and let d be the greatest common divisor of m and b. The probability that $X_{n+1} < X_n$ is equal to $\frac{1}{2} + r$, where*

$$r = \big(2(c \bmod d) - d\big)/2m; \tag{1}$$

hence $|r| < d/2m$.

Proof. The proof of this theorem involves some techniques that are of interest in themselves. First we define

$$s(x) = (ax + c) \bmod m. \tag{2}$$

Thus, $X_{n+1} = s(X_n)$, and the theorem reduces to counting the number of integers x such that $0 \le x < m$ and $s(x) < x$ (since each such integer occurs somewhere in the period). We want to show that this number is

$$\tfrac{1}{2}\big(m + 2(c \bmod d) - d\big). \tag{3}$$

The function $\lceil (x - s(x))/m \rceil$ is equal to 1 when $x > s(x)$, and it is 0 otherwise; hence the count we wish to obtain can be written simply as

$$\begin{aligned}\sum_{0\le x<m} \left\lceil \frac{x - s(x)}{m} \right\rceil &= \sum_{0\le x<m} \left\lceil \frac{x}{m} - \left(\frac{ax+c}{m} - \left\lfloor \frac{ax+c}{m} \right\rfloor \right) \right\rceil \\ &= \sum_{0\le x<m} \left(\left\lfloor \frac{ax+c}{m} \right\rfloor - \left\lfloor \frac{bx+c}{m} \right\rfloor \right). \end{aligned} \tag{4}$$

(Recall that $\lceil -y \rceil = -\lfloor y \rfloor$ and $b = a - 1$.) Such sums can be evaluated by the method of exercise 1.2.4–37, where we have proved that

$$\sum_{0\le j<k} \left\lfloor \frac{hj + c}{k} \right\rfloor = \frac{(h-1)(k-1)}{2} + \frac{g-1}{2} + g\lfloor c/g \rfloor, \qquad g = \gcd(h, k), \tag{5}$$

whenever h and k are integers and $k > 0$. Since a is relatively prime to m, this formula yields

$$\begin{aligned}\sum_{0\le x<m} \left\lfloor \frac{ax+c}{m} \right\rfloor &= \frac{(a-1)(m-1)}{2} + c, \\ \sum_{0\le x<m} \left\lfloor \frac{bx+c}{m} \right\rfloor &= \frac{(b-1)(m-1)}{2} + \frac{d-1}{2} + c - (c \bmod d), \end{aligned}$$

and (3) follows immediately. ∎

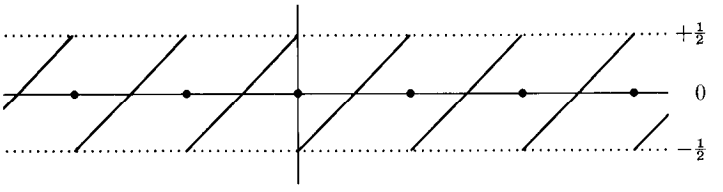


Fig. 7. The sawtooth function $((z))$.

The proof of Theorem P indicates that *a priori* tests can indeed be carried out, provided that we are able to deal satisfactorily with sums involving the $\lfloor\ \rfloor$ and $\lceil\ \rceil$ functions. In many cases the most powerful technique for dealing with floor and ceiling functions is to replace them by two somewhat more symmetrical ones:

$$\delta(z) = \lfloor z \rfloor + 1 - \lceil z \rceil = \begin{cases} 1, & \text{if } z \text{ is an integer;} \\ 0, & \text{if } z \text{ is not an integer;} \end{cases} \tag{6}$$

$$((z)) = z - \lfloor z \rfloor - \tfrac{1}{2} + \tfrac{1}{2}\delta(z) = z - \lceil z \rceil + \tfrac{1}{2} - \tfrac{1}{2}\delta(z). \tag{7}$$

The latter function is a "sawtooth" function familiar in the study of Fourier series; its graph is shown in Fig. 7. The reason for choosing to work with $((z))$ rather than $\lfloor z \rfloor$ or $\lceil z \rceil$ is that $((z))$ possesses several very useful properties:

$$((-z)) = -((z)); \tag{8}$$

$$((z+n)) = ((z)), \quad \text{integer } n; \tag{9}$$

$$((nz)) = ((z)) + \left(\!\!\left(z + \frac{1}{n}\right)\!\!\right) + \cdots + \left(\!\!\left(z + \frac{n-1}{n}\right)\!\!\right), \quad \text{integer } n \geq 1. \tag{10}$$

(See exercise 2.)

In order to get some practice working with these functions, let us prove Theorem P again, this time without relying on exercise 1.2.4–37. With the help of Eqs. (7), (8), (9), we can show that

$$\begin{aligned} \left\lceil \frac{x - s(x)}{m} \right\rceil &= \frac{x - s(x)}{m} - \left(\!\!\left(\frac{x - s(x)}{m}\right)\!\!\right) + \frac{1}{2} - \frac{1}{2}\delta\left(\frac{x - s(x)}{m}\right) \\ &= \frac{x - s(x)}{m} - \left(\!\!\left(\frac{x - (ax + c)}{m}\right)\!\!\right) + \frac{1}{2} \\ &= \frac{x - s(x)}{m} + \left(\!\!\left(\frac{bx + c}{m}\right)\!\!\right) + \frac{1}{2} \end{aligned} \tag{11}$$

since $\big(x - s(x)\big)/m$ is never an integer. Now

$$\sum_{0 \leq x < m} \frac{x - s(x)}{m} = 0$$

since both x and $s(x)$ take on each value of $\{0, 1, \ldots, m-1\}$ exactly once; hence (11) yields

$$\sum_{0\le x<m} \left\lceil \frac{x-s(x)}{m} \right\rceil = \sum_{0\le x<m} \left(\!\!\left(\frac{bx+c}{m} \right)\!\!\right) + \frac{m}{2}. \tag{12}$$

Let $b = b_0 d$, $m = m_0 d$, where b_0 and m_0 are relatively prime. We know that $(b_0 x) \bmod m_0$ takes on the values $\{0, 1, \ldots, m_0 - 1\}$ in some order as x varies from 0 to $m_0 - 1$. By (9) and (10) and the fact that

$$\left(\!\!\left(\frac{b(x+m_0)+c}{m} \right)\!\!\right) = \left(\!\!\left(\frac{bx+c}{m} \right)\!\!\right)$$

we have

$$\begin{aligned} \sum_{0\le x<m} \left(\!\!\left(\frac{bx+c}{m} \right)\!\!\right) &= d \sum_{0\le x\le m_0} \left(\!\!\left(\frac{bx+c}{m} \right)\!\!\right) \\ &= d \sum_{0\le x<m_0} \left(\!\!\left(\frac{c}{m} + \frac{b_0 x}{m} \right)\!\!\right) = d\left(\!\!\left(\frac{c}{d} \right)\!\!\right). \end{aligned} \tag{13}$$

Theorem P follows immediately from (12) and (13).

One consequence of Theorem P is that practically *any* choice of a and c will give a reasonable probability that $X_{n+1} < X_n$, at least over the entire period, except those that have large d. A large value of d corresponds to low potency, and we already know that generators of low potency are undesirable.

The next theorem gives us a more stringent condition for the choice of a and c; we will consider the *serial correlation test* applied over the entire period. The quantity C defined in Section 3.3.2, Eq. (23), is

$$C = \left(m \sum_{0\le x<m} x s(x) - \left(\sum_{0\le x<m} x \right)^2 \right) \Big/ \left(m \sum_{0\le x<m} x^2 - \left(\sum_{0\le x<m} x \right)^2 \right). \tag{14}$$

Let x' be the element such that $s(x') = 0$. We have

$$s(x) = m\left(\!\!\left(\frac{ax+c}{m} \right)\!\!\right) + \frac{m}{2}, \qquad \text{if } x \neq x'. \tag{15}$$

The formulas we are about to derive can be expressed most easily in terms of the function

$$\sigma(h,k,c) = 12 \sum_{0\le j<k} \left(\!\!\left(\frac{j}{k} \right)\!\!\right)\left(\!\!\left(\frac{hj+c}{k} \right)\!\!\right), \tag{16}$$

an important function that arises in several mathematical problems. It is called a *generalized Dedekind sum*, since Richard Dedekind introduced the function $\sigma(h,k,0)$ in 1876 when commenting on one of Riemann's incomplete manuscripts. [See B. Riemann's *Gesammelte Math. Werke*, 2nd ed. (1892), 466–478.]

Using the well-known formulas

$$\sum_{0\le x<m} x = \frac{m(m-1)}{2} \qquad \text{and} \qquad \sum_{0\le x<m} x^2 = \frac{m(m-1)(2m-1)}{6},$$

it is a straightforward matter to transform Eq. (14) into

$$C = \frac{m\sigma(a,m,c) - 3 + 6(m - x' - c)}{m^2 - 1}. \tag{17}$$

(See exercise 5.) Since m is usually very large, we may discard terms of order $1/m$, and we have the approximation

$$C \approx \sigma(a,m,c)/m, \tag{18}$$

with an error of less than $6/m$ in absolute value.

The serial correlation test now reduces to determining the value of the Dedekind sum $\sigma(a,m,c)$. Evaluating $\sigma(a,m,c)$ directly from its definition (16) is hardly any easier than evaluating the correlation coefficient itself directly, but fortunately there are simple methods available for computing Dedekind sums quite rapidly.

Lemma B (*"Reciprocity law" for Dedekind sums*). *Let h, k, c be integers. If $0 \le c < k$, $0 < h \le k$, and if h is relatively prime to k, then*

$$\sigma(h,k,c) + \sigma(k,h,c) = \frac{h}{k} + \frac{k}{h} + \frac{1}{hk} + \frac{6c^2}{hk} - 6\left\lfloor \frac{c}{h} \right\rfloor - 3e(h,c), \tag{19}$$

where

$$e(h,c) = \begin{cases} 1, & \text{if } c = 0 \text{ or } c \bmod h \neq 0; \\ 0, & \text{if } c > 0 \text{ and } c \bmod h = 0. \end{cases} \tag{20}$$

Proof. We leave it to the reader to prove that, under these hypotheses,

$$\sigma(h,k,c) + \sigma(k,h,c) = \sigma(h,k,0) + \sigma(k,h,0) + \frac{6c^2}{hk} - 6\left\lfloor \frac{c}{h} \right\rfloor - 3e(h,c) + 3. \tag{21}$$

(See exercise 6.) The lemma now must be proved only in the case $c = 0$.

The proof we will give, based on complex roots of unity, is essentially due to L. Carlitz. There is actually a simpler proof that uses only elementary manipulations of sums (see exercise 7)—but the following method reveals more of the mathematical tools that are available for problems of this kind and it is therefore much more instructive.

Let $f(x)$ and $g(x)$ be polynomials defined as follows:

$$\begin{aligned} f(x) &= 1 + x + \cdots + x^{k-1} = (x^k - 1)/(x - 1) \\ g(x) &= x + 2x^2 + \cdots + (k-1)x^{k-1} = xf'(x) \\ &= kx^k/(x-1) - x(x^k - 1)/(x-1)^2. \end{aligned} \tag{22}$$

If ω is the complex kth root of unity $e^{2\pi i/k}$, we have by Eq. 1.2.9–13

$$\frac{1}{k}\sum_{0\le j<k}\omega^{-jr}g(\omega^j x)=rx^r,\qquad \text{if } 0\le r<k. \tag{23}$$

Set $x=1$; then $g(\omega^j x)=k/(\omega^j-1)$ if $j\neq 0$, otherwise it equals $k(k-1)/2$, therefore

$$r \bmod k=\sum_{0<j<k}\frac{\omega^{-jr}}{\omega^j-1}+\tfrac{1}{2}(k-1),\qquad \text{if } r \text{ is an integer.}$$

(Eq. (23) shows that the right-hand side equals r when $0\le r<k$, and it is unchanged when multiples of k are added to r.) Hence

$$\Big(\Big(\frac{r}{k}\Big)\Big)=\frac{1}{k}\sum_{0<j<k}\frac{\omega^{-jr}}{\omega^j-1}-\frac{1}{2k}+\frac{1}{2}\delta\Big(\frac{r}{k}\Big). \tag{24}$$

This important formula, which holds whenever r is an integer, allows us to reduce many calculations involving $((r/k))$ to sums involving kth roots of unity, and it brings a whole new range of techniques into the picture. In particular, we get the following formula:

$$\sigma(h,k,0)+\frac{3(k-1)}{k^2}=\frac{12}{k^2}\sum_{0<r<k}\sum_{0<i<k}\sum_{0<j<k}\frac{\omega^{-ir}}{\omega^i-1}\frac{\omega^{-jhr}}{\omega^j-1}. \tag{25}$$

The right-hand side of this formula may be simplified by carrying out the sum on r; we have $\sum_{0\le r<k}\omega^{rs}=f(\omega^s)=0$ if $s \bmod k\neq 0$. Equation (25) now reduces to

$$\sigma(h,k,0)+\frac{3(k-1)}{k}=\frac{12}{k}\sum_{0<j<k}\frac{1}{(\omega^{-jh}-1)(\omega^j-1)}. \tag{26}$$

A similar formula is obtained for $\sigma(k,h,0)$, with $\zeta=e^{2\pi i/h}$ replacing ω.

It is not obvious what we can do with the sum in (26), but there is an elegant way to proceed, based on the fact that each term of the sum is a function of ω^j, where $0<j<k$; hence the sum is essentially taken over the kth roots of unity other than 1. Whenever $x_1, x_2, \ldots, x_n$ are distinct complex numbers, we have the identity

$$\sum_{1\le j\le n}\frac{1}{(x_j-x_1)\ldots(x_j-x_{j-1})(x-x_j)(x_j-x_{j+1})\ldots(x_j-x_n)}=\frac{1}{(x-x_1)\ldots(x-x_n)}, \tag{27}$$

which follows from the usual method of expanding the right-hand side into partial fractions. Moreover, if $q(x) = (x - y_1)(x - y_2)\dots(x - y_m)$, we have

$$q'(y_j) = (y_j - y_1)\dots(y_j - y_{j-1})(y_j - y_{j+1})\dots(y_j - y_m); \tag{28}$$

this identity may often be used to simplify expressions like those in the left-hand side of (27). When h and k are relatively prime, the numbers ω, ω^2, $\dots$, ω^{k-1}, ς, ς^2, $\dots$, ς^{h-1} are all distinct; we can therefore consider formula (27) in the special case of the polynomial $(x - \omega)\dots(x - \omega^{k-1})(x - \varsigma)\dots(x - \varsigma^{h-1}) = (x^k - 1)(x^h - 1)/(x - 1)^2$, obtaining the following identity in x:

$$\frac{1}{h}\sum_{0<j<h}\frac{\varsigma^j(\varsigma^j - 1)^2}{(\varsigma^{jk} - 1)(x - \varsigma^j)} + \frac{1}{k}\sum_{0<j<k}\frac{\omega^j(\omega^j - 1)^2}{(\omega^{jh} - 1)(x - \omega^j)} = \frac{(x - 1)^2}{(x^h - 1)(x^k - 1)}. \tag{29}$$

This identity has many interesting consequences, and it leads to numerous reciprocity formulas for sums of the type given in Eq. (26). For example, if we differentiate (29) twice with respect to x and let $x \to 1$, we find that

$$\frac{2}{h}\sum_{0<j<h}\frac{\varsigma^j(\varsigma^j - 1)^2}{(\varsigma^{jk} - 1)(1 - \varsigma^j)^3} + \frac{2}{k}\sum_{0<j<k}\frac{\omega^j(\omega^j - 1)^2}{(\omega^{jh} - 1)(1 - \omega^j)^3} = \frac{1}{6}\left(\frac{h}{k} + \frac{k}{h} + \frac{1}{hk}\right) + \frac{1}{2} - \frac{1}{2h} - \frac{1}{2k}.$$

Replace j by $h - j$ and by $k - j$ in these sums and use (26) to get

$$\frac{1}{6}\left(\sigma(k, h, 0) + \frac{3(h - 1)}{h}\right) + \frac{1}{6}\left(\sigma(h, k, 0) + \frac{3(k - 1)}{k}\right) = \frac{1}{6}\left(\frac{h}{k} + \frac{k}{h} + \frac{1}{hk}\right) + \frac{1}{2} - \frac{1}{2h} - \frac{1}{2k},$$

which is equivalent to the desired result. ∎

Lemma B gives us an explicit function $f(h, k, c)$ such that

$$\sigma(h, k, c) = f(h, k, c) - \sigma(k, h, c) \tag{30}$$

whenever $0 < h \le k$, $0 \le c < k$, and h is relatively prime to k. From the definition (16) it is clear that

$$\sigma(k, h, c) = \sigma(k \bmod h,\ h,\ c \bmod h). \tag{31}$$

Therefore we can use (30) iteratively to evaluate $\sigma(h, k, c)$, using a process that reduces the parameters as in Euclid's algorithm.

Further simplifications occur when we examine this iterative procedure more closely. Let us set $m_1 = k$, $m_2 = h$, $c_1 = c$, and form the following tableau:

$$\begin{aligned} m_1 &= a_1 m_2 + m_3 & \qquad c_1 &= b_1 m_2 + c_2 \\ m_2 &= a_2 m_3 + m_4 & \qquad c_2 &= b_2 m_3 + c_3 \\ m_3 &= a_3 m_4 + m_5 & \qquad c_3 &= b_3 m_4 + c_4 \\ m_4 &= a_4 m_5 & \qquad c_4 &= b_4 m_5 + c_5 \end{aligned} \tag{32}$$

Here

$$\begin{aligned} a_j &= \lfloor m_j/m_{j+1} \rfloor, & b_j &= \lfloor c_j/m_{j+1} \rfloor, \\ m_{j+2} &= m_j \bmod m_{j+1}, & c_{j+1} &= c_j \bmod m_{j+1}, \end{aligned} \tag{33}$$

and it follows that

$$0 \le m_{j+1} < m_j, \qquad 0 \le c_j < m_j. \tag{34}$$

We have assumed for convenience that Euclid's algorithm terminates in (32) after four iterations; this assumption will reveal the pattern that holds in the general case. Since h and k were relatively prime to start with, we must have $m_5 = 1$ and $c_5 = 0$ in (32).

Let us further assume that $c_3 \neq 0$ but $c_4 = 0$, in order to get a feeling for the effect this has on the recurrence. Equations (30) and (31) yield

$$\begin{aligned} \sigma(h,k,c) &= \sigma(m_2, m_1, c_1) \\ &= f(m_2, m_1, c_1) - \sigma(m_3, m_2, c_2) \\ &= \cdots \\ &= f(m_2, m_1, c_1) - f(m_3, m_2, c_2) + f(m_4, m_3, c_3) - f(m_5, m_4, c_4). \end{aligned}$$

The first part "$h/k + k/h$" of the formula for $f(h,k,c)$ in (19) contributes

$$\frac{m_2}{m_1} + \frac{m_1}{m_2} - \frac{m_3}{m_2} - \frac{m_2}{m_3} + \frac{m_4}{m_3} + \frac{m_3}{m_4} - \frac{m_5}{m_4} - \frac{m_4}{m_5}$$

to the total, and this simplifies to

$$\frac{h}{k} + \left(a_1 + \frac{m_3}{m_2}\right) - \frac{m_3}{m_2} - \left(a_2 + \frac{m_4}{m_3}\right) + \frac{m_4}{m_3} + \left(a_3 + \frac{m_5}{m_4}\right) - \frac{m_5}{m_4} - a_4 = h/k + a_1 - a_2 + a_3 - a_4.$$

The next part "$1/hk$" of (19) also leads to a simple contribution; according to Eq. 4.5.3–9 and other formulas in Section 4.5.3, we have

$$1/m_1m_2 - 1/m_2m_3 + 1/m_3m_4 - 1/m_4m_5 = h'/k - 1, \tag{35}$$

where h' is the unique integer satisfying

$$h'h \equiv 1 \text{ (modulo } k), \quad 0 < h' \le k. \tag{36}$$

Adding up all the contributions, and remembering our assumption that $c_4 = 0$ (so that $e(m_4, c_3) = 0$, cf. (20)), we find that

$$\sigma(h,k,c) = \frac{h+h'}{k} + (a_1 - a_2 + a_3 - a_4) - 6(b_1 - b_2 + b_3 - b_4) + 6\left(\frac{c_1^2}{m_1m_2} - \frac{c_2^2}{m_2m_3} + \frac{c_3^2}{m_3m_4} - \frac{c_4^2}{m_4m_5}\right) + 2,$$

in terms of the assumed tableau (32). Similar results hold in general:

Theorem D. *Let h, k, c be integers with $0 < h \le k$, $0 \le c < k$, and h relatively prime to k. Form the "Euclidean tableau" as defined in (33) above, and assume that the process stops after t steps with $m_{t+1} = 1$. Let s be the smallest subscript such that $c_s = 0$, and let h' be defined by (36). Then*

$$\sigma(h,k,c) = \frac{h+h'}{k} + \sum_{1\le j\le t} (-1)^{j+1}\left(a_j - 6b_j + 6\frac{c_j^2}{m_j m_{j+1}}\right) + 3\big((-1)^s + \delta_{s1}\big) - 2 + (-1)^t. \quad \blacksquare$$

Euclid's algorithm is analyzed carefully in Section 4.5.3; the quantities a_1, a_2, ..., a_t are called the *partial quotients* of h/k. Theorem 4.5.3F tells us that the number of iterations, t, will never exceed $\log_\phi k$; hence Dedekind sums can be evaluated rapidly. The terms $c_j^2/m_j m_{j+1}$ can be simplified further, and an efficient algorithm for evaluating $\sigma(h,k,c)$ appears in exercise 17.

Now that we have analyzed generalized Dedekind sums, let us apply our knowledge to the determination of serial correlation coefficients.

Example 1. *Find the serial correlation when $m = 2^{35}$, $a = 2^{34}+1$, $c = 1$.*

Solution. We have

$$C = \big(2^{35}\sigma(2^{34}+1, 2^{35}, 1) - 3 + 6(2^{35} - (2^{34}-1) - 1)\big)/(2^{70}-1)$$

by Eq. (17). To evaluate $\sigma(2^{34}+1, 2^{35}, 1)$, we can form the tableau

$m_1 = 2^{35}$				$c_1 = 1$			
$m_2 = 2^{34}+1$	$a_1 = 1$			$c_2 = 1$	$b_1 = 0$		
$m_3 = 2^{34}-1$	$a_2 = 1$			$c_3 = 1$	$b_2 = 0$		
$m_4 = 2$	$a_3 = 2^{33}-1$			$c_4 = 1$	$b_3 = 0$		
$m_5 = 1$	$a_4 = 2$			$c_5 = 0$	$b_4 = 1$		

Since $h' = 2^{34}+1$, the value according to Theorem D comes to $2^{33} - 3 + 2^{-32}$. Thus

$$C = (2^{68}+5)/(2^{70}-1) = \tfrac{1}{4} + \epsilon, \qquad |\epsilon| < 2^{-67}. \tag{37}$$

Such a correlation is much, much too high for randomness. Of course, this generator has very low potency, and we have already rejected it as nonrandom.

Example 2. *Find the approximate serial correlation when $m = 10^{10}$, $a = 10001$, $c = 2113248653$.*

Solution. We have $C \approx \sigma(a,m,c)/m$, and the computation proceeds as follows:

$m_1 = 10000000000$				$c_1 = 2113248653$			
$m_2 =$ 10001	$a_1 = 999900$			$c_2 =$ 7350	$b_1 = 211303$		
$m_3 =$ 100	$a_2 =$ 100			$c_3 =$ 50	$b_2 =$ 73		
$m_4 =$ 1	$a_3 =$ 100			$c_4 =$ 0	$b_3 =$ 50		

$$\sigma(m_2, m_1, c_1) = -31.6926653544;$$
$$C \approx -3\cdot 10^{-9}. \tag{38}$$

This is a very respectable value of C indeed. But the generator has a potency of only 3, *so it is not really a very good source of random numbers in spite of the fact that it has low serial correlation.* It is necessary to have a low serial correlation, but not sufficient.

Example 3. *Estimate the serial correlation for general a, m, and c.*

Solution. If we consider just one application of (30), we have

$$\sigma(a,m,c) \approx \frac{m}{a} + 6\frac{c^2}{am} - 6\frac{c}{a} - \sigma(m,a,c).$$

Now $|\sigma(m,a,c)| < a$ by exercise 12, and therefore

$$C \approx \frac{\sigma(a,m,c)}{m} \approx \frac{1}{a}\Big(1 - 6\frac{c}{m} + 6\Big(\frac{c}{m}\Big)^2\Big). \tag{39}$$

The error in this approximation is less than $(a+6)/m$ in absolute value.

The estimate in (39) was the first theoretical result known about the randomness of congruential generators. R. R. Coveyou [*JACM* **7** (1960), 72–74] obtained it by averaging over all *real* numbers x between 0 and m instead of considering only the integer values (cf. exercise 21); then Martin Greenberger [*Math. Comp.* **15** (1961), 383–389] gave a rigorous derivation including an estimate of the error term.

So began one of the saddest chapters in the history of computer science! Although the above approximation is quite correct, it has been grievously misapplied in practice; people abandoned the perfectly good generators they had been using and replaced them by terrible generators that looked good from the standpoint of (39). For more than a decade, the most common random number generators in daily use were seriously deficient, solely because of a theoretical advance. A little knowledge is a dangerous thing.

If we are to learn by past mistakes, we had better look carefully at how (39) has been misused. In the first place people assumed uncritically that a small serial correlation over the whole period would be a pretty good guarantee of randomness; but in fact it doesn't even ensure a small serial correlation for 1000 consecutive elements of the sequence (see exercise 14).

Secondly, (39) and its error term will ensure a relatively small value of C only when $a \approx \sqrt{m}$; therefore people suggested choosing multipliers near $\sqrt{m}$. In fact, we shall see that nearly all multipliers give a value of C that is substantially less than $1/\sqrt{m}$, hence (39) is not a very good approximation to the true behavior. Minimizing a crude upper bound for C does not minimize C.

In the third place, people observed that (39) yields its best estimate when $c/m \approx \frac{1}{2} \pm \frac{1}{6}\sqrt{3}$, since these values are the roots of $1 - 6x + 6x^2 = 0$. "In the absence of any other criterion for choosing c, we might as well use this one." The latter statement is not incorrect, but it is misleading at best, since experience has shown that the value of c has hardly any influence on the true value of the serial

correlation when a is a good multiplier; the choice $c/m \approx \frac{1}{2} \pm \frac{1}{6}\sqrt{3}$ reduces C substantially only in cases like Example 2 above. And we are fooling ourselves in such cases, since the bad multiplier will reveal its deficiencies in other ways.

Clearly we need a better estimate than (39); and such an estimate is now available thanks to Theorem D, which stems principally from the work of U. Dieter [*Math. Comp.* **25** (1971), 855–883]. Theorem D implies that $\sigma(a, m, c)$ will be small *if the partial quotients of* a/m *are small.* Indeed, by analyzing generalized Dedekind sums still more closely, it is possible to obtain quite a sharp estimate:

Theorem K. *Under the assumptions of Theorem D, we always have*

$$-\frac{1}{2}\sum_{\substack{1\le j\le t\\ j\text{ odd}}} a_j - \sum_{\substack{1\le j\le t\\ j\text{ even}}} a_j + \frac{1}{2} \le \sigma(h,k,c) \le \sum_{\substack{1\le j\le t\\ j\text{ odd}}} a_j + \frac{1}{2}\sum_{\substack{1\le j\le t\\ j\text{ even}}} a_j - \frac{1}{2}. \quad (40)$$

Proof. See D. E. Knuth, *Acta Arithmetica* **33** (1978), 297–325, where it is shown further that these bounds are essentially the best possible when there are large partial quotients. ∎

Example 4. *Estimate the serial correlation for* $a = 3141592621$, $m = 2^{35}$, c *odd.*

Solution. The partial quotients of a/m are 10, 1, 14, 1, 7, 1, 1, 1, 3, 3, 3, 5, 2, 1, 8, 7, 1, 4, 1, 2, 4, 2; hence by Theorem K

$$-45 \le \sigma(a, m, c) \le 68,$$

and the serial correlation is guaranteed to be extremely low for all c.

Note that this bound is considerably better than we could obtain from (39), since the error in (39) is of order a/m; our "random" multiplier has turned out to be much better than one specifically chosen to look good on the basis of (39). In fact, it is possible to show that the *average* value of $\sum_{1\le j\le t} a_j$, taken over all multipliers a relatively prime to m, is

$$\frac{6}{\pi^2}(\ln m)^2 + O\big((\log m)(\log\log m)^4\big)$$

(see exercise 4.5.3–35). Therefore the probability that a random multiplier has large $\sum_{1\le j\le t} a_t$, say larger than $(\log m)^{2+\epsilon}$ for some fixed $\epsilon > 0$, approaches zero as $m \to \infty$. This substantiates the empirical evidence that almost all linear congruential sequences have extremely low serial correlation over the entire period.

The exercises below show that other *a priori* tests, such as the serial test over the entire period, can also be expressed in terms of a few generalized Dedekind sums. It follows from Theorem K that linear congruential sequences will pass these tests provided that certain specified fractions (depending on a and m but

not on c) have small partial quotients. In particular, the result of exercise 19 implies that *the serial test on pairs will be satisfactorily passed if and only if a/m has no large partial quotients.*

The book *Dedekind Sums* by Hans Rademacher and Emil Grosswald (Math. Assoc. of America, Carus Monograph No. 16, 1972) discusses the history and properties of Dedekind sums and their generalizations. Further theoretical tests, including the serial test in higher dimensions, are discussed in Section 3.3.4.

EXERCISES—First Set

1. [*M10*] Express "$x \bmod y$" in terms of the sawtooth and δ functions.

2. [*M20*] Prove the "replicative law," Eq. (10).

3. [*HM22*] What is the Fourier series expansion (in terms of sines and cosines) of the function $f(x) = ((x))$?

▶ **4.** [*M19*] If $m = 10^{10}$, what is the highest possible value of d (in the notation of Theorem P), given that the potency of the generator is 10?

5. [*M21*] Carry out the derivation of Eq. (17).

6. [*M27*] Let $hh' + kk' = 1$. (a) Show, without using Lemma B, that

$$\sigma(h,k,c) = \sigma(h,k,0) + 12 \sum_{0<j<c} \left(\!\!\left(\frac{h'j}{k}\right)\!\!\right) + 6\left(\!\!\left(\frac{h'c}{k}\right)\!\!\right)$$

for all integers $c \geq 0$. (b) Show that if $0 < j < k$,

$$\left(\!\!\left(\frac{h'j}{k}\right)\!\!\right) + \left(\!\!\left(\frac{k'j}{h}\right)\!\!\right) = \frac{j}{hk} - \frac{1}{2}\delta\left(\frac{j}{h}\right).$$

(c) Under the assumptions of Lemma B, prove Eq. (21).

▶ **7.** [*M24*] Give a proof of the reciprocity law (19), when $c = 0$, by using the general reciprocity law of exercise 1.2.4–45.

▶ **8.** [*M34*] (L. Carlitz.) Let

$$\rho(p,q,r) = 12 \sum_{0\leq j<r} \left(\!\!\left(\frac{jp}{r}\right)\!\!\right)\left(\!\!\left(\frac{jq}{r}\right)\!\!\right).$$

By generalizing the method of proof used in Lemma B, prove the following beautiful identity due to H. Rademacher: If each of p, q, r is relatively prime to the other two,

$$\rho(p,q,r) + \rho(q,r,p) + \rho(r,p,q) = \frac{p}{qr} + \frac{q}{rp} + \frac{r}{pq} - 3.$$

(The reciprocity law for Dedekind sums, with $c = 0$, is the special case $r = 1$.)

9. [*M40*] Is there a simple proof of Rademacher's identity (exercise 8) along the lines of the proof in exercise 7 of a special case?

10. [*M20*] Show that when $0 < h < k$ it is possible to express $\sigma(k - h, k, c)$ and $\sigma(h, k, -c)$ easily in terms of $\sigma(h, k, c)$.

11. [*M30*] The formulas given in the text show us how to evaluate $\sigma(h, k, c)$ when h and k are relatively prime and c is an integer. For the general case, prove that

a) $\sigma(dh, dk, dc) = \sigma(h, k, c)$, integer $d > 0$;

b) $\sigma(h, k, c + \theta) = \sigma(h, k, c) + 6((h'c/k))$, integer c, real $0 < \theta < 1$, when h and k are relatively prime and $hh' \equiv 1$ (modulo k).

12. [*M24*] Show that if h is relatively prime to k and c is an integer, $|\sigma(h, k, c)| \leq (k - 1)(k - 2)/k$.

13. [*M24*] Generalize Eq. (26) so that it gives an expression for $\sigma(h, k, c)$.

▶ **14.** [*M20*] The linear congruential generator that has $m = 2^{35}$, $a = 2^{18} + 1$, $c = 1$, was given the serial correlation test on three batches of 1000 consecutive numbers, and the result was a very high correlation, between 0.2 and 0.3, in each case. What is the serial correlation of this generator, taken over all 2^{35} numbers of the period?

15. [*M21*] Generalize Lemma B so that it applies to all *real* values of c, $0 \leq c < k$.

16. [*M24*] Given the Euclidean tableau defined in (33), let $p_0 = 1$, $p_1 = a_1$, and $p_j = a_j p_{j-1} + p_{j-2}$ for $1 < j \leq t$. Show that the complicated portion of the sum in Theorem D can be rewritten as follows, making it possible to avoid noninteger computations:

$$\sum_{1 \leq j \leq t} (-1)^{j+1} \frac{c_j^2}{m_j m_{j+1}} = \frac{1}{m_1} \sum_{1 \leq j \leq t} (-1)^{j+1} b_j (c_j + c_{j+1}) p_{j-1}.$$

[*Hint:* Prove that we have $\sum_{1 \leq j \leq r} (-1)^{j+1}/m_j m_{j+1} = (-1)^{r+1} p_{r-1}/m_1 m_{r+1}$ for $1 \leq r \leq t$.]

17. [*M22*] Design an algorithm that evaluates $\sigma(h, k, c)$ for integers h, k, c satisfying the hypotheses of Theorem D. Your algorithm should use only integer arithmetic (of unlimited precision), and it should produce the answer in the form $A + B/k$ where A and B are integers. (Cf. exercise 16.) If possible, use only a finite number of variables for temporary storage, instead of maintaining arrays such as $a_1, a_2, \ldots, a_t$.

▶ **18.** [*M23*] (U. Dieter.) Given positive integers h, k, z, let

$$S(h, k, c, z) = \sum_{0 \leq j < z} \left(\!\!\left(\frac{hj + c}{k} \right)\!\!\right).$$

Show that this sum can be evaluated in "closed form," in terms of generalized Dedekind sums and the sawtooth function. [*Hint:* When $z \leq k$, the quantity $\lfloor j/k \rfloor - \lfloor (j - z)/k \rfloor$ equals 1 for $0 \leq j < z$, and it equals 0 for $z \leq j < k$, so we can introduce this factor and sum over $0 \leq j < k$.]

▶ **19.** [*M23*] Show that the *serial test* can be analyzed over the full period, in terms of generalized Dedekind sums, by finding a formula for the probability that $\alpha \leq X_n < \beta$ and $\alpha' \leq X_{n+1} < \beta'$ when α, β, α', β' are given integers with $0 \leq \alpha < \beta \leq m$, $0 \leq \alpha' < \beta' \leq m$. [*Hint:* Consider the quantity $\lfloor (x - \alpha)/m \rfloor - \lfloor (x - \beta)/m \rfloor$.]

20. [*M29*] (U. Dieter.) Extend Theorem P by obtaining a formula for the probability that $X_n > X_{n+1} > X_{n+2}$, in terms of generalized Dedekind sums.

EXERCISES—Second Set

In many cases, exact computations with integers are quite difficult to carry out, but we can attempt to study the probabilities that arise when we take the average over all real values of x instead of restricting the calculation to integer values. Although these results are only approximate, they shed some light on the subject.

It is convenient to deal with numbers U_n between zero and one; for linear congruential sequences, $U_n = X_n/m$, and we have $U_{n+1} = \{aU_n + \theta\}$, where $\theta = c/m$ and $\{x\}$ denotes $x \bmod 1$. For example, the formula for serial correlation now becomes

$$C = \Bigg(\int_0^1 x\{ax+\theta\}\,dx - \Big(\int_0^1 x\,dx\Big)^2\Bigg) \Big/ \Bigg(\int_0^1 x^2\,dx - \Big(\int_0^1 x\,dx\Big)^2\Bigg).$$

▸ **21.** [*HM23*] (R. R. Coveyou.) What is the value of C in the formula just given?

▸ **22.** [*M22*] Let a be an integer, and let $0 \le \theta < 1$. If x is a real number between 0 and 1, and if $s(x) = \{ax + \theta\}$, what is the probability that $s(x) < x$? (This is the "real number" analog of Theorem P.)

23. [*28*] The previous exercise gives the probability that $U_{n+1} < U_n$. What is the probability that $U_{n+2} < U_{n+1} < U_n$, assuming that U_n is a random real number between zero and one?

24. [*M29*] Under the assumptions of the preceding problem, except with $\theta = 0$, show that $U_n > U_{n+1} > \cdots > U_{n+t-1}$ occurs with probability

$$\frac{1}{t!}\Big(1+\frac{1}{a}\Big)\ldots\Big(1+\frac{t-2}{a}\Big).$$

What is the average length of a descending run starting at U_n, assuming that U_n is selected at random between zero and one?

▸ **25.** [*M25*] Let α, β, α', β' be real numbers with $0 \le \alpha < \beta \le 1$, $0 \le \alpha' < \beta' \le 1$. Under the assumptions of exercise 22, what is the probability that $\alpha \le x < \beta$ and $\alpha' \le s(x) < \beta'$? (This is the "real number" analog of exercise 19.)

26. [*M21*] Consider a "Fibonacci" generator, where $U_{n+1} = \{U_n + U_{n-1}\}$. Assuming that U_1 and U_2 are independently chosen at random between 0 and 1, find the probability that $U_1 < U_2 < U_3$, $U_1 < U_3 < U_2$, $U_2 < U_1 < U_3$, etc. [*Hint:* Divide the "unit square," i.e., the points of the plane $\{(x, y) \mid 0 \le x, y < 1\}$, into six parts, depending on the relative order of x, y, and $\{x + y\}$, and determine the area of each part.]

27. [*M32*] In the Fibonacci generator of the preceding exercise, let U_0 and U_1 be chosen independently in the unit square except that $U_0 > U_1$. Determine the probability that U_1 is the beginning of an upward run of length k, so that $U_0 > U_1 < \cdots < U_k > U_{k+1}$. Compare this with the corresponding probabilities for a random sequence.

28. [*M35*] According to Eq. 3.2.1.3–5, a linear congruential generator with potency 2 satisfies the condition $X_{n-1} - 2X_n + X_{n+1} \equiv (a-1)c$ (modulo m). Consider a generator that abstracts this situation: let $U_{n+1} = \{\alpha + 2U_n - U_{n-1}\}$. As in exercise 26, divide the unit square into parts that show the relative order of U_1, U_2, and U_3 for each pair (U_1, U_2). Are there any values of α for which all six possible orders are achieved with probability $\frac{1}{6}$, assuming that U_1 and U_2 are chosen at random in the unit square?

3.3.4. The Spectral Test

In this section we shall study an especially important way to check the quality of linear congruential random number generators; not only do all good generators pass this test, all generators now known to be bad actually *fail* it. Thus it is by far the most powerful test known, and it deserves particular attention. Our discussion will also bring out some fundamental limitations on the degree of randomness we can expect from linear congruential sequences and their generalizations.

The spectral test embodies aspects of both the empirical and theoretical tests studied in previous sections: it is like the theoretical tests because it deals with properties of the full period of the sequence, and it is like the empirical tests because it requires a computer program to determine the results.

A. Ideas underlying the test. The most important randomness criteria seem to rely on properties of the joint distribution of t consecutive elements of the sequence, and the spectral test deals directly with this distribution. If we have a sequence $\langle U_n \rangle$ of period m, the idea is to analyze the set of all m points

$$\{(U_n, U_{n+1}, \ldots, U_{n+t-1})\} \tag{1}$$

in t-dimensional space.

For simplicity we shall assume that we have a linear congruential sequence (X_0, a, c, m) of maximum period length m (so that $c \neq 0$), or that m is prime and $c = 0$ and the period length is $m - 1$. In the latter case we shall add the point $(0, 0, \ldots, 0)$ to the set (1), so that there are always m points in all; this extra point has a negligible effect when m is large, and it makes the theory much simpler. Under these assumptions, (1) can be rewritten as

$$\left\{ \frac{1}{m}(x, s(x), s(s(x)), \ldots, s^{t-1}(x)) \;\middle|\; 0 \leq x < m \right\}, \tag{2}$$

where

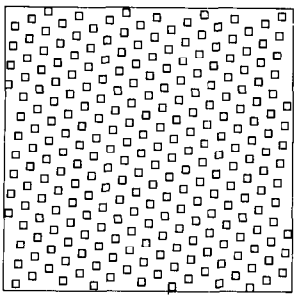
$$s(x) = (ax + c) \bmod m \tag{3}$$

is the "successor" of x. Note that we are considering only the set of all such points in t dimensions, not the order in which those points are actually generated. But the order of generation is reflected in the dependence between components of the vectors; and the spectral test studies such dependence for various dimensions t by dealing with the totality of all points (2).

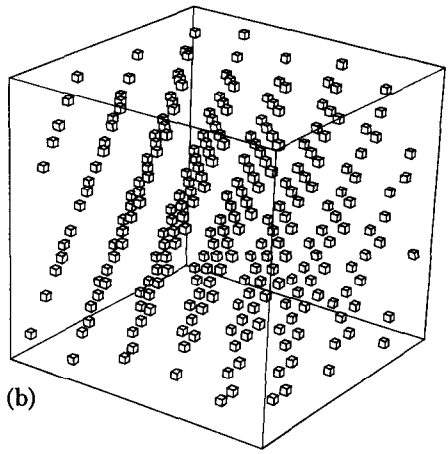
For example, Fig. 8 shows a typical small case in 2 and 3 dimensions, for the generator with

$$s(x) = (137x + 187) \bmod 256. \tag{4}$$

Of course a generator with period length 256 will hardly be random, but 256 is small enough that we can draw the diagram and gain some understanding before we turn to the larger m's that are of practical interest.



(a)



(b)

Fig. 8. (a) The two-dimensional grid formed by all pairs of successive points (X_n, X_{n+1}), when $X_{n+1} = (137X_n + 187) \bmod 256$.
(b) The three-dimensional grid of triplets (X_n, X_{n+1}, X_{n+2}). [Illustrations courtesy of Bruce G. Baumgart.]

Perhaps the most striking thing about the pattern of boxes in Fig. 8 is that we can cover them all by a fairly small number of parallel lines; indeed, there are many different families of parallel lines that will hit all the points. For example, a set of 20 nearly vertical lines will do the job, as will a set of 21 lines that tilt upward at roughly a 30° angle. We commonly observe similar patterns when driving past farmlands that have been planted in a systematic manner.

If the same generator is considered in three dimensions, we obtain 256 points in a cube, obtained by appending a "height" component $s(s(x))$ to each of the 256 points $\big(x, s(x)\big)$ in the plane of Fig. 8(a), as shown in Fig. 8(b). Let's imagine that this 3-D crystal structure has been made into a physical model, a cube that we can turn in our hands; as we rotate it, we will notice various families of parallel planes that encompass all of the points. In the words of Wallace Givens, the random numbers stay "mainly in the planes."

At first glance we might think that such systematic behavior is so nonrandom as to make congruential generators quite worthless; but more careful reflection, remembering that m is quite large in practice, provides a better insight. The regular structure in Fig. 8 is essentially the "grain" we see when examining our random numbers under a high-power microscope. If we take truly random numbers between 0 and 1, and round or truncate them to finite accuracy so that each is an integer multiple of $1/\nu$ for some given number ν, then the t-dimensional points (1) we obtain will have an extremely regular character when viewed through a microscope.

Let $1/\nu_2$ be the maximum distance between lines, taken over all families of parallel straight lines that cover the points $\{(x/m,\ s(x)/m)\}$ in two dimensions. We shall call ν_2 the two-dimensional *accuracy* of the random number generator, since the pairs of successive numbers have a fine structure that is

essentially good to one part in ν_2. Similarly, let $1/\nu_3$ be the maximum distance between planes, taken over all families of parallel planes that cover all points $\{(x/m,\ s(x)/m,\ s(s(x))/m)\}$; we shall call ν_3 the accuracy in three dimensions. The t-dimensional accuracy ν_t is the reciprocal of the maximum distance between hyperplanes, taken over all families of parallel $(t-1)$-dimensional hyperplanes that cover all points $\{(x/m,\ s(x)/m,\ \ldots,\ s^{t-1}(x)/m)\}$.

The essential difference between periodic sequences and truly random sequences that have been truncated to multiples of $1/\nu$ is that the "accuracy" of truly random sequences is the same in all dimensions, while the "accuracy" of periodic sequences decreases as t increases. Indeed, since there are only m points in the t-dimensional cube when m is the period length, we can't achieve a t-dimensional accuracy of more than about $m^{1/t}$.

When the independence of t consecutive values is considered, computer-generated random numbers will behave essentially as if we took truly random numbers and truncated them to $\lg \nu_t$ bits, where ν_t decreases with increasing t. In practice, such varying accuracy is usually all we need. We don't insist that the 10-dimensional accuracy be 2^{35}, in the sense that all $(2^{35})^{10}$ possible 10-tuples $(U_n, U_{n+1}, \ldots, U_{n+9})$ should be equally likely on a 35-bit machine; for such large values of t we want only a few of the leading bits of $(U_n, U_{n+1}, \ldots, U_{n+t-1})$ to behave as if they were independently random.

On the other hand when an application demands high resolution of the random number sequence, simple linear congruential sequences will necessarily be inadequate; a generator with larger period should be used instead, even though only a small fraction of the period will actually be generated. Squaring the period will essentially square the accuracy in higher dimensions, i.e., it will double the effective number of bits of precision.

The spectral test is based on the values of ν_t for small t, say $2 \le t \le 6$. Dimensions 2, 3, and 4 seem to be adequate to detect important deficiencies in a sequence, but since we are considering the entire period it seems best to be somewhat cautious and go up into another dimension or two; on the other hand the values of ν_t for $t \ge 10$ seem to be of no practical significance whatever. (This is fortunate, because it appears to be rather difficult to calculate ν_t when $t \ge 10$.)

Note that there is a vague relation between the spectral test and the serial test; for example, a special case of the serial test, taken over the entire period as in exercise 3.3.3–19, counts the number of boxes in each of 64 subsquares of Fig. 8(a). The main difference is that the spectral test rotates the dots so as to discover the least favorable orientation. We shall return to a consideration of the serial test later in this section.

It may appear at first that we should apply the spectral test only for one suitably high value of t; if a generator passes the test in three dimensions, it seems plausible that it should also pass the 2-D test, hence we might as well omit the latter. The fallacy in this reasoning occurs because we apply more stringent conditions in lower dimensions. A similar situation occurs with the serial test: Consider a generator that (quite properly) has almost the same number of points

in each subcube of the unit cube, when the unit cube has been divided into 64 subcubes of size $\frac{1}{4} \times \frac{1}{4} \times \frac{1}{4}$; this same generator might yield completely *empty* subsquares of the unit square, when the unit square has been divided into 64 subsquares of size $\frac{1}{8} \times \frac{1}{8}$. Since we increase our expectations in lower dimensions, a separate test for each dimension is required.

It is not always true that $\nu_t \le m^{1/t}$, although this upper bound is valid when the points form a rectangular grid. For example, it turns out that $\nu_2 = \sqrt{274} > \sqrt{256}$ in Fig. 8, because a nearly hexagonal structure brings the m points closer together than would be possible in a strictly rectangular arrrangement.

In order to develop an algorithm that computes ν_t efficiently, we must look more deeply at the associated mathematical theory. Therefore a reader who is not mathematically inclined is advised to skip to part D of this section, where the spectral test is presented as a "plug-in" method accompanied by several examples. On the other hand, we shall see that the mathematics behind the spectral test requires only some elementary manipulations of vectors.

Some authors have suggested using the minimum number N_t of parallel covering lines or hyperplanes as the criterion, instead of the maximum distance $1/\nu_t$ between them. However, this number does not appear to be as important as the concept of accuracy defined above, because it is biased by how nearly the slope of the lines or hyperplanes matches the coordinate axes of the cube. For example, the 20 nearly vertical lines that cover all the points of Fig. 8 are actually $1/\sqrt{328}$ units apart, and this might falsely imply an accuracy of one part in $\sqrt{328}$, or perhaps even of one part in 20. The true accuracy of only one part in $\sqrt{274}$ is realized only for the larger family of 21 lines with a slope of $7/15$; another family of 24 lines, with a slope of $-11/13$, also has a greater inter-line distance than the 20-line family, since $1/\sqrt{290} > 1/\sqrt{328}$. The precise way in which families of lines act at the boundaries of the unit hypercube does not seem to be an especially "clean" or significant criterion; however, for those people who prefer to count hyperplanes, it is possible to compute N_t using a method quite similar to the way in which we shall calculate ν_t (see exercise 16).

***B. Theory behind the test.** In order to analyze the basic set (2), we start with the observation that

$$\frac{1}{m} s^j(x) = \left(\frac{a^j x + (1 + a + \cdots + a^{j-1})c}{m} \right) \bmod 1. \tag{5}$$

We can get rid of the "mod 1" operation by extending the set periodically, making infinitely many copies of the original t-dimensional hypercube, proceeding in all directions. This gives us the set

$$\begin{aligned} L &= \left\{ \left(\frac{x}{m} + k_1, \frac{s(x)}{m} + k_2, \ldots, \frac{s^{t-1}(x)}{m} + k_t \right) \,\middle|\, \text{integer } x, k_1, k_2, \ldots, k_t \right\} \\ &= \left\{ V_0 + \left(\frac{x}{m} + k_1, \frac{ax}{m} + k_2, \ldots, \frac{a^{t-1}x}{m} + k_t \right) \,\middle|\, \text{integer } x, k_1, k_2, \ldots, k_t \right\}, \end{aligned}$$

where

$$V_0 = \frac{1}{m}(0, c, (1+a)c, \ldots, (1+a+\cdots+a^{t-2})c) \tag{6}$$

is a constant vector. The variable k_1 is redundant in this representation of L, because we can change $(x, k_1, k_2, \ldots, k_t)$ to $(x + k_1 m, 0, k_2 - ak_1, \ldots, k_t - a^{t-1}k_1)$, reducing k_1 to zero without loss of generality. Therefore we obtain the comparatively simple formula

$$L = \{\, V_0 + y_1 V_1 + y_2 V_2 + \cdots + y_t V_t \mid \text{integer } y_1, y_2, \ldots, y_t \,\}, \tag{7}$$

where

$$V_1 = \frac{1}{m}(1, a, a^2, \ldots, a^{t-1}); \tag{8}$$

$$V_2 = (0, 1, 0, \ldots, 0), \quad V_3 = (0, 0, 1, \ldots, 0), \quad \ldots, \quad V_t = (0, 0, 0, \ldots, 1). \tag{9}$$

The points $(x_1, x_2, \ldots, x_t)$ of L that satisfy $0 \le x_j < 1$ for all j are precisely the m points of our original set (2).

Note that the increment c appears only in V_0, and the effect of V_0 is merely to shift all elements of L without changing their relative distances; hence c *does not affect the spectral test in any way*, and we might as well assume that $V_0 = (0, 0, \ldots, 0)$ when we are calculating ν_t. When V_0 is the zero vector we have a so-called *lattice* of points

$$L_0 = \{\, y_1 V_1 + y_2 V_2 + \cdots + y_t V_t \mid \text{integer } y_1, y_2, \ldots, y_t \,\}, \tag{10}$$

and our goal is to study the distances between adjacent $(t-1)$-dimensional hyperplanes, in families of parallel hyperplanes that cover all the points of L_0.

A family of parallel $(t-1)$-dimensional hyperplanes can be defined by a nonzero vector $U = (u_1, \ldots, u_t)$ that is perpendicular to all of them; and the set of points on a particular hyperplane is then

$$\{\, (x_1, \ldots, x_t) \mid x_1 u_1 + \cdots + x_t u_t = q \,\}, \tag{11}$$

where q is a different constant for each hyperplane in the family. In other words, each hyperplane is the set of all X for which the *dot product* $X \cdot U$ has a given value q. In our case the hyperplanes are all separated by a fixed distance, and one of them contains $(0, 0, \ldots, 0)$; hence we can adjust the magnitude of U so that the set of all *integer* values q gives all the hyperplanes in the family. Then the distance between neighboring hyperplanes is the minimum distance from $(0, 0, \ldots, 0)$ to the hyperplane for $q = 1$, namely

$$\min_{\text{real } x_1, \ldots, x_t} \left\{ \sqrt{x_1^2 + \cdots + x_t^2} \;\middle|\; x_1 u_1 + \cdots + x_t u_t = 1 \right\}. \tag{12}$$

Cauchy's inequality (cf. exercise 1.2.3–30) tells us that

$$(x_1 u_1 + \cdots + x_t u_t)^2 \le (x_1^2 + \cdots + x_t^2)(u_1^2 + \cdots + u_t^2), \tag{13}$$

hence the minimum in (12) occurs when each $x_j = u_j/(u_1^2 + \cdots + u_t^2)$; the distance between neighboring hyperplanes is

$$1\Big/\sqrt{u_1^2 + \cdots + u_t^2} = 1/\text{length}(U). \tag{14}$$

In other words, the quantity ν_t we seek is precisely the length of the shortest vector U that defines a family of hyperplanes $\{ X \cdot U = q \mid \text{integer } q \}$ containing all the elements of L_0.

Such a vector $U = (u_1, \ldots, u_t)$ must be nonzero, and it must satisfy $V \cdot U =$ integer for all V in L_0. In particular, since the points $(1, 0, \ldots, 0)$, $(0, 1, \ldots, 0)$, $\ldots$, $(0, 0, \ldots, 1)$ are all in L_0, all of the u_j must be integers. Furthermore since V_1 is in L_0, we must have $\frac{1}{m}(u_1 + au_2 + \cdots + a^{t-1}u_t) =$ integer, i.e.,

$$u_1 + au_2 + \cdots + a^{t-1}u_t \equiv 0 \quad (\text{modulo } m). \tag{15}$$

Conversely, any nonzero integer vector $U = (u_1, \ldots, u_t)$ satisfying (15) defines a family of hyperplanes with the required properties, since all of L_0 will be covered: $(y_1V_1 + \cdots + y_tV_t) \cdot U$ will be an integer for all integers y_1, $\ldots$, y_t. We have proved that

$$\begin{aligned}\nu_t^2 &= \min_{(u_1,\ldots,u_t)\neq(0,\ldots,0)} \{u_1^2 + \cdots + u_t^2 \mid u_1 + au_2 + \cdots + a^{t-1}u_t \equiv 0 \text{ (modulo } m)\} \\ &= \min_{(x_1,\ldots,x_t)\neq(0,\ldots,0)} \big((mx_1 - ax_2 - a^2x_3 - \cdots - a^{t-1}x_t)^2 + x_2^2 + x_3^2 + \cdots + x_t^2\big).\end{aligned} \tag{16}$$

C. Deriving a computational method. We have now reduced the spectral test to the problem of finding the minimum value (16); but how on earth can we determine that minimum value in a reasonable amount of time? A brute-force search is out of the question, since m is very large in cases of practical interest.

It will be interesting and probably more useful if we develop a computational method for solving an even more general problem: *Find the minimum value of the quantity*

$$f(x_1, \ldots, x_t) = (u_{11}x_1 + \cdots + u_{t1}x_t)^2 + \cdots + (u_{1t}x_1 + \cdots + u_{tt}x_t)^2 \tag{17}$$

over all nonzero integer vectors $(x_1, \ldots, x_t)$, given any nonsingular matrix of coefficients $U = (u_{ij})$. The expression (17) is called a "positive definite quadratic form" in t variables. Since U is nonsingular, (17) cannot be zero unless the x_j are all zero.

Let us write U_1, $\ldots$, U_t for the rows of U. Then (17) may be written

$$f(x_1, \ldots, x_t) = (x_1U_1 + \cdots + x_tU_t) \cdot (x_1U_1 + \cdots + x_tU_t), \tag{18}$$

the square of the length of the vector $x_1U_1 + \cdots + x_tU_t$. The nonsingular matrix U has an inverse, which means that we can find uniquely determined vectors

$V_1, \ldots, V_t$ such that

$$U_i \cdot V_j = \delta_{ij}, \qquad 1 \le i, j \le t. \tag{19}$$

For example, in the special form (16) that arises in the spectral test, we have

$$\begin{aligned}
U_1 &= (\quad m, 0, 0, \ldots, 0), & V_1 &= \tfrac{1}{m}(1, a, a^2, \ldots, a^{t-1}),\\
U_2 &= (\quad -a, 1, 0, \ldots, 0), & V_2 &= \quad (0, 1, \ 0, \ldots, \quad 0),\\
U_3 &= (\quad -a^2, 0, 1, \ldots, 0), & V_3 &= \quad (0, 0, \ 1, \ldots, \quad 0),\\
&\quad \cdots\cdots\cdots\cdots & & \\
U_t &= (-a^{t-1}, 0, 0, \ldots, 1), & V_t &= \quad (0, 0, \ 0, \ldots, \quad 1).
\end{aligned} \tag{20}$$

These V_j are precisely the vectors (8), (9) that we used to define our original lattice L_0. As the reader may well suspect, this is not a coincidence—indeed, if we had begun with an arbitrary lattice L_0, defined by any set of linearly independent vectors $V_1, \ldots, V_t$, the argument we have used above can be generalized to show that the maximum separation between hyperplanes in a covering family is equivalent to minimizing (17), where the coefficients u_{ij} are defined by (19). (See exercise 2.)

Our first step in minimizing (18) is to reduce it to a finite problem, i.e., to show that we won't need to test infinitely many vectors $(x_1, \ldots, x_t)$ to find the minimum. This is where the vectors $V_1, \ldots, V_t$ come in handy; we have

$$x_k = (x_1 U_1 + \cdots + x_t U_t) \cdot V_k,$$

and Cauchy's inequality tells us that

$$\big((x_1 U_1 + \cdots + x_t U_t) \cdot V_k\big)^2 \le f(x_1, \ldots, x_t)(V_k \cdot V_k).$$

Hence we have derived a useful upper bound on each coordinate x_k:

Lemma A. *Let $(x_1, \ldots, x_t)$ be a nonzero vector that minimizes* (18) *and let $(y_1, \ldots, y_t)$ be any nonzero integer vector. Then*

$$x_k^2 \le (V_k \cdot V_k) f(y_1, \ldots, y_t), \qquad \text{for } 1 \le k \le t. \tag{21}$$

In particular, letting $y_i = \delta_{ij}$ for all i,

$$x_k^2 \le (V_k \cdot V_k)(U_j \cdot U_j), \qquad \text{for } 1 \le j, k \le t. \ \blacksquare \tag{22}$$

Lemma A reduces the problem to a finite search, but the right-hand side of (21) is usually much too large to make an exhaustive search feasible; we need at least one more idea. On such occasions, an old maxim provides sound advice: "If you can't solve a problem as it is stated, change it into a simpler problem that

has the same answer." For example, Euclid's algorithm has this form; if we don't know the gcd of the input numbers, we change them into smaller numbers having the same gcd. (In fact, a slightly more general approach probably underlies the discovery of nearly all algorithms: "If you can't solve a problem directly, change it into one or more simpler problems, from whose solution you can solve the original one.")

In our case, a simpler problem is one that requires less searching because the right-hand side of (22) is smaller. The key idea we shall use is that it is possible to change one quadratic form into another one that is equivalent for all practical purposes. Let j be any fixed subscript, $1 \le j \le t$; let $(q_1, \ldots, q_{j-1}, q_{j+1}, \ldots, q_t)$ be any sequence of $t - 1$ integers; and consider the following transformation of the vectors:

$$\begin{aligned} V_i' &= V_i - q_i V_j, \qquad & x_i' &= x_i - q_i x_j, \qquad & U_i' &= U_i, \qquad \text{for } i \neq j; \\ V_j' &= V_j, & x_j' &= x_j, & U_j' &= U_j + \textstyle\sum_{i \neq j} q_i U_i. \end{aligned} \tag{23}$$

It is easy to see that the new vectors $U_1', \ldots, U_t'$ define a quadratic form f' for which $f'(x_1', \ldots, x_t') = f(x_1, \ldots, x_t)$; furthermore the basic orthogonality condition (19) remains valid, because it is easy to check that $U_i' \cdot V_j' = \delta_{ij}$. As $(x_1, \ldots, x_t)$ runs through all nonzero integer vectors, so does $(x_1', \ldots, x_t')$; hence the new form f' has the same minimum as f.

Our goal is to use transformation (23), replacing U_i by U_i' and V_i by V_i' for all i, in order to make the right-hand side of (22) small; and the right-hand side of (22) will be small when both $U_j \cdot U_j$ and $V_k \cdot V_k$ are small. Therefore it is natural to ask the following two questions about the transformation (23):

a) *What choice of q_i makes $V_i' \cdot V_i'$ as small as possible?*

b) *What choice of $q_1, \ldots, q_{j-1}, q_{j+1}, \ldots, q_t$ makes $U_j' \cdot U_j'$ as small as possible?*

It is easiest to solve these questions first for *real* values of the q_i. Question (a) is quite simple, since

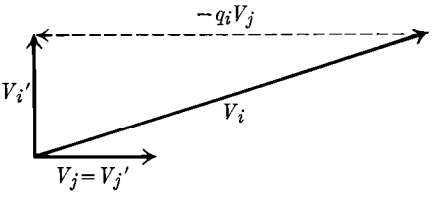
$$\begin{aligned} (V_i - q_i V_j) \cdot (V_i - q_i V_j) &= V_i \cdot V_i - 2q_i\, V_i \cdot V_j + q_i^2\, V_j \cdot V_j \\ &= (V_j \cdot V_j)\big(q_i - (V_i \cdot V_j \,/\, V_j \cdot V_j)\big)^2 \\ &\quad + V_i \cdot V_i - (V_i \cdot V_j)^2 \,/\, V_j \cdot V_j, \end{aligned}$$

and the minimum occurs when

$$q_i = V_i \cdot V_j \,/\, V_j \cdot V_j. \tag{24}$$

Geometrically, we are asking what multiple of V_j should be subtracted from V_i so that the resulting vector V_i' has minimum length, and the answer is to choose q_i so that V_i' is perpendicular to V_j (i.e., so that $V_i' \cdot V_j = 0$); the following

diagram makes this plain.



(25)

Turning to question (b), we want to choose the q_i so that $U_j + \sum_{i \neq j} q_i U_i$ has minimum length; geometrically, we want to start with U_j and add some vector in the $(t-1)$-dimensional hyperplane whose points are the sums of multiples of $\{U_i \mid i \neq j\}$. Again the best solution is to choose things so that U_j' is perpendicular to the hyperplane, i.e., so that $U_j' \cdot U_k = 0$ for all $k \neq j$, i.e.,

$$U_j \cdot U_k + \sum_{i \neq j} q_i (U_i \cdot U_k) = 0, \qquad 1 \leq k \leq t, \qquad k \neq j. \tag{26}$$

(See exercise 12 for a rigorous proof that a solution to question (b) must satisfy these $t-1$ equations.)

Now that we have answered questions (a) and (b), we are in a bit of a quandary; should we choose the q_i according to (24), so that the $V_i' \cdot V_i'$ are minimized, or according to (26), so that $U_j' \cdot U_j'$ is minimized? Either of these alternatives makes an improvement in the right-hand side of (22), so it is not immediately clear which choice should get priority. Fortunately, there is a very simple answer to this dilemma: Conditions (24) and (26) are exactly the same! (See exercise 7.) Therefore questions (a) and (b) have the same answer; we have a happy state of affairs in which we can reduce the length of both the U's and the V's simultaneously. (It may be worthwhile to point out that we have just rediscovered the "Schmidt orthogonalization process.")

Our joy must be tempered with the realization that we have dealt with questions (a) and (b) only for *real* values of the q_i. Our application restricts us to integer values, so we cannot make V_i' exactly perpendicular to V_j. The best we can do for question (a) is to let q_i be the *nearest integer* to $V_i \cdot V_j \,/\, V_j \cdot V_j$ (cf. (25)). It turns out that this is *not* always the best solution to question (b); in fact U_j' may at times be longer than U_j. However, the bound (21) is never increased, since we can remember the smallest value of $f(y_1, \ldots, y_t)$ found so far. Thus a choice of q_i based solely on question (a) is quite satisfactory.

If we apply transformation (23) repeatedly in such a way that none of the vectors V_i gets longer and at least one gets shorter, we can never get into a loop; i.e., we will never be considering the same quadratic form again after a sequence of nontrivial transformations of this kind. But eventually we will get "stuck," in the sense that none of the transformations (23) for $1 \leq j \leq t$ will be able to shorten any of the vectors $V_1, \ldots, V_t$. At that point we can revert to an exhaustive search, using the bounds of Lemma A, which will now

be quite small in most cases. Occasionally these bounds (21) will be poor, and another type of transformation will usually get the algorithm unstuck again and reduce the bounds (see exercise 18). However, transformation (23) by itself has proved to be quite adequate for the spectral test; in fact, it has proved to be amazingly powerful when the computations are arranged as in the algorithm discussed below.

***D. How to perform the spectral test.** Here now is an efficient computational procedure that follows from our considerations. R. W. Gosper and U. Dieter have observed that it is possible to use the results of lower dimensions to make the spectral test significantly faster in higher dimensions. This refinement has been incorporated into the following algorithm, together with a significant simplification of the two-dimensional case.

Algorithm S (*The spectral test*). This algorithm determines the value of

$$\nu_t = \min\left\{\sqrt{x_1^2 + \cdots + x_t^2} \;\middle|\; x_1 + ax_2 + \cdots + a^{t-1}x_t \equiv 0 \text{ (modulo } m)\right\} \quad (27)$$

for $2 \le t \le T$, given a, m, and T, where $0 < a < m$ and a is relatively prime to m. (The number ν_t measures the t-dimensional accuracy of random number generators, as discussed in the text above.) All arithmetic within this algorithm is done on integers whose magnitudes rarely if ever exceed m^2, except in step S8; in fact, nearly all of the integer variables will be less than m in absolute value during the computation.

When ν_t is being calculated for $t \ge 3$, the algorithm works with two $t \times t$ matrices U and V, whose row vectors are denoted by $U_i = (u_{i1}, \ldots, u_{it})$ and $V_i = (v_{i1}, \ldots, v_{it})$ for $1 \le i \le t$. These vectors satisfy the conditions

$$u_{i1} + au_{i2} + \cdots + a^{t-1}u_{it} \equiv 0 \text{ (modulo } m), \qquad 1 \le i \le t; \quad (28)$$

$$U_i \cdot V_j = \delta_{ij} m, \qquad 1 \le i, j \le t. \quad (29)$$

(Thus the V_j of our previous discussion have been multiplied by m, to ensure that their components are integers.) There are three other auxiliary vectors, $X = (x_1, \ldots, x_t)$, $Y = (y_1, \ldots, y_t)$, and $Z = (z_1, \ldots, z_t)$. During the entire algorithm, r will denote $a^{t-1} \bmod m$ and s will denote the smallest upper bound for ν_t^2 that has been discovered so far.

S1. [Initialize.] Set $h \leftarrow a$, $h' \leftarrow m$, $p \leftarrow 1$, $p' \leftarrow 0$, $r \leftarrow a$, $s \leftarrow 1 + a^2$. (The first steps of this algorithm handle the case $t = 2$ by a special method, very much like Euclid's algorithm; we will have

$$h - ap \equiv h' - ap' \equiv 0 \text{ (modulo } m) \qquad \text{and} \qquad hp' - h'p = \pm m \quad (30)$$

during this phase of the calculation.)

S2. [Euclidean step.] Set $q \leftarrow \lfloor h'/h \rfloor$, $u \leftarrow h' - qh$, $v \leftarrow p' - qp$. If $u^2 + v^2 < s$, set $s \leftarrow u^2 + v^2$, $h' \leftarrow h$, $h \leftarrow u$, $p' \leftarrow p$, $p \leftarrow v$, and repeat step S2.

S3. [Compute ν_2.] Set $u \leftarrow u - h$, $v \leftarrow v - p$; and if $u^2 + v^2 < s$, set $s \leftarrow u^2 + v^2$, $h' \leftarrow u$, $p' \leftarrow v$. Then output $\sqrt{s} = \nu_2$. (The validity of this calculation for the two-dimensional case is proved in exercise 5. Now we will set up the U and V matrices satisfying (28) and (29), in preparation for calculations in higher dimensions.) Set

$$U \leftarrow \begin{pmatrix} -h & p \\ -h' & p' \end{pmatrix}, \qquad V \leftarrow \pm \begin{pmatrix} p' & h' \\ -p & -h \end{pmatrix},$$

where the $-$ sign is chosen for V if and only if $p' > 0$.

S4. [Advance t.] If $t = T$, the algorithm terminates. (Otherwise we want to increase t by 1. At this point U and V are $t \times t$ matrices satisfying (28) and (29), and we must enlarge them by adding an appropriate new row and column.) Set $t \leftarrow t + 1$ and $r \leftarrow (ar) \bmod m$. Set U_t to the new row $(-r, 0, 0, \ldots, 0, 1)$ of t elements, and set $u_{it} \leftarrow 0$ for $1 \le i < t$. Set V_t to the new row $(0, 0, 0, \ldots, 0, m)$. Finally, for $1 \le i < t$, set $q \leftarrow \text{round}(v_{i1}\, r/m)$, $v_{it} \leftarrow v_{i1} r - qm$, and $U_t \leftarrow U_t + qU_i$. (Here "round($x$)" denotes the nearest integer to x, e.g., $\lfloor x + 1/2 \rfloor$. We are essentially setting $v_{it} \leftarrow v_{i1} r$ and immediately applying transformation (23) with $j = t$, since the numbers $|v_{i1} r|$ are so large they ought to be reduced at once.) Finally set $s \leftarrow \min(s, U_t \cdot U_t)$, $k \leftarrow t$, and $j \leftarrow 1$. (In the following steps, j denotes the current row index for transformation (23), and k denotes the last such index where the transformation shortened at least one of the V_i.)

S5. [Transform.] For $1 \le i \le t$, do the following operations: If $i \ne j$ and $2|V_i \cdot V_j| > V_j \cdot V_j$, set $q \leftarrow \text{round}(V_i \cdot V_j \;/\; V_j \cdot V_j)$, $V_i \leftarrow V_i - qV_j$, $U_j \leftarrow U_j + qU_i$, and $k \leftarrow j$. (The fact that we omit this transformation, when $2|V_i \cdot V_j|$ exactly equals $V_j \cdot V_j$, prevents the algorithm from looping endlessly; see exercise 19.)

S6. [Examine new bound.] If $k = j$ (i.e., if the transformation in S5 has just done something useful), set $s \leftarrow \min(s, U_j \cdot U_j)$.

S7. [Advance j.] If $j = t$, set $j \leftarrow 1$; otherwise set $j \leftarrow j + 1$. Now if $j \ne k$, return to step S5. (If $j = k$, we have gone through $t - 1$ consecutive cycles of no transformation, so the transformation process is stuck.)

S8. [Prepare for search.] (Now the absolute minimum will be determined, using an exhaustive search over all $(x_1, \ldots, x_t)$ satisfying condition (21) of Lemma A.) Set $X \leftarrow Y \leftarrow (0, \ldots, 0)$, set $k \leftarrow t$, and set

$$z_j \leftarrow \left\lfloor \sqrt{\lfloor (V_j \cdot V_j) s/m^2 \rfloor} \right\rfloor, \qquad \text{for } 1 \le j \le t. \tag{31}$$

(We will examine all $X = (x_1, \ldots, x_t)$ with $|x_j| \le z_j$ for $1 \le j \le t$. In hundreds of applications of this algorithm, no z_j has yet turned out to be greater than 1, nor has the exhaustive search in the following steps ever reduced s; however, such phenomena are probably possible in weird cases,

especially in higher dimensions. During the exhaustive search, the vector Y will always be equal to $x_1U_1 + \cdots + x_tU_t$, so that $f(x_1, \ldots, x_t) = Y \cdot Y$. Since $f(-x_1, \ldots, -x_t) = f(x_1, \ldots, x_t)$, we shall examine only vectors whose first nonzero component is positive. The method is essentially that of counting in steps of one, regarding $(x_1, \ldots, x_t)$ as the digits in a balanced number system with mixed radices $(2z_1 + 1, \ldots, 2z_t + 1)$; cf. Section 4.1.)

S9. [Advance x_k.] If $x_k = z_k$, go to S11. Otherwise increase x_k by 1 and set $Y \leftarrow Y + U_k$.

S10. [Advance k.] Set $k \leftarrow k+1$. Then if $k \le t$, set $x_k \leftarrow -z_k$, $Y \leftarrow Y - 2z_kU_k$, and repeat step S10. But if $k > t$, set $s \leftarrow \min(s, Y \cdot Y)$.

S11. [Decrease k.] Set $k \leftarrow k - 1$. If $k \ge 1$, return to S9. Otherwise output $\nu_t = \sqrt{s}$ (the exhaustive search is completed) and return to S4. ∎

In practice Algorithm S is applied for $T = 5$ or 6, say; it usually works reasonably well when $T = 7$ or 8, but it can be terribly slow when $T \ge 9$ since the exhaustive search tends to make the running time grow as 3^T. (If the minimum value ν_t occurs at many different points, the exhaustive search will hit them all; hence we typically find that all $z_k = 1$ for large t. As remarked above, the values of ν_t are generally irrelevant for practical purposes when t is large.)

An example will help to make Algorithm S clear. Consider the linear congruential sequence defined by

$$m = 10^{10}, \qquad a = 3141592621, \qquad c = 1, \qquad X_0 = 0. \tag{32}$$

Six cycles of the Euclidean algorithm in steps S2 and S3 suffice to prove that the minimum nonzero value of $x_1^2 + x_2^2$ with

$$x_1 + 3141592621x_2 \equiv 0 \text{ (modulo } 10^{10})$$

occurs for $x_1 = 67654$, $x_2 = 226$; hence the two-dimensional accuracy of this generator is

$$\nu_2 = \sqrt{67654^2 + 226^2} \approx 67654.37748.$$

Passing to three dimensions, we seek the minimum nonzero value of $x_1^2 + x_2^2 + x_3^2$ such that

$$x_1 + 3141592621x_2 + 3141592621^2x_3 \equiv 0 \text{ (modulo } 10^{10}). \tag{33}$$

Step S4 sets up the matrices

$$U = \begin{pmatrix} -67654 & -226 & 0 \\ -44190611 & 191 & 0 \\ 5793866 & 33 & 1 \end{pmatrix}, \quad V = \begin{pmatrix} -191 & -44190611 & 2564918569 \\ -226 & 67654 & 1307181134 \\ 0 & 0 & 10000000000 \end{pmatrix}.$$

The first iteration of step S5, with $q = 1$ for $i = 2$ and $q = 4$ for $i = 3$, changes them to

$$U = \begin{pmatrix} -21082801 & 97 & 4 \\ -44190611 & 191 & 0 \\ 5793866 & 33 & 1 \end{pmatrix}, \quad V = \begin{pmatrix} -191 & -44190611 & 2564918569 \\ -35 & 44258265 & -1257737435 \\ 764 & 176762444 & -259674276 \end{pmatrix}.$$

(Note that the first row U_1 has actually gotten longer in this transformation, although eventually the rows of U should get shorter.)

The next fourteen iterations of step S5 have $(j, q_1, q_2, q_3) = (2, -2, *, 0)$, $(3, 0, 3, *)$, $(1, *, -10, -1)$, $(2, -1, *, -6)$, $(3, -1, 0, *)$, $(1, *, 0, 2)$, $(2, 0, *, -1)$, $(3, 3, 4, *)$, $(1, *, 0, 0)$, $(2, -5, *, 0)$, $(3, 1, 0, *)$, $(1, *, -3, -1)$, $(2, 0, *, 0)$, $(3, 0, 0, *)$. Now the transformation process is stuck, but the rows of the matrices have become significantly shorter:

$$U = \begin{pmatrix} -1479 & 616 & -2777 \\ -3022 & 104 & 918 \\ -227 & -983 & -130 \end{pmatrix}, \quad V = \begin{pmatrix} -888874 & 601246 & -2994234 \\ -2809871 & 438109 & 1593689 \\ -854296 & -9749816 & -1707736 \end{pmatrix}. \tag{34}$$

The search limits (z_1, z_2, z_3) in step S8 turn out to be $(0, 0, 1)$, so U_3 is the shortest solution to (33); we have

$$\nu_3 = \sqrt{227^2 + 983^2 + 130^2} \approx 1017.21089.$$

Note that only a few iterations were needed to find this value, although condition (33) looks quite difficult to deal with at first glance. All points (U_n, U_{n+1}, U_{n+2}) produced by this random number generator lie on a family of parallel planes about 0.001 units apart.

E. Ratings for various generators. So far we haven't really given a criterion that tells us whether or not a particular random number generator "passes" or "flunks" the spectral test. In fact, this depends on the application, since some applications demand higher resolution than others. It appears that $\nu_t \geq 2^{30/t}$ for $2 \leq t \leq 6$ will be quite adequate in most applications (although the author must admit choosing this criterion partly because 30 is conveniently divisible by 2, 3, 5, and 6).

For some purposes we would like a criterion that is relatively independent of m, so we can say that a particular multiplier is good or bad with respect to the set of all other multipliers for the given m, without examining any others. A reasonable figure of merit for rating the goodness of a particular multiplier seems to be the volume of the ellipsoid in t-space defined by the relation $(x_1 m - x_2 a - \cdots - x_t a^{t-1})^2 + x_2^2 + \cdots + x_t^2 \leq \nu_t^2$, since this volume tends to indicate how likely it is that nonzero *integer* points $(x_1, \ldots, x_t)$—corresponding to solutions of (15)—are in the ellipsoid. We therefore propose to calculate this volume, namely

$$\mu_t = \frac{\pi^{t/2}\, \nu_t^t}{(t/2)!\, m}, \tag{35}$$

as an indication of the effectiveness of the multiplier a for the given m. In this formula,

$$\left(\frac{t}{2}\right)! = \left(\frac{t}{2}\right)\left(\frac{t}{2} - 1\right) \ldots \left(\frac{1}{2}\right)\sqrt{\pi}, \qquad \text{for } t \text{ odd.} \tag{36}$$

Table 1

SAMPLE RESULTS OF THE SPECTRAL TEST

Line	a	m	ν_2^2	ν_3^2	ν_4^2	ν_5^2	ν_6^2
1	23	10^8+1	530	530	530	530	447
2	2^7+1	2^{35}	16642	16642	16642	15602	252
3	$2^{18}+1$	2^{35}	34359738368	6	4	4	4
4	3141592653	2^{35}	2997222016	1026050	27822	1118	1118
5	137	256	274	30	14	6	4
6	3141592621	10^{10}	4577114792	1034718	62454	1776	542
7	3141592221	10^{10}	4293881050	276266	97450	3366	2382
8	4219755981	10^{10}	10721093248	2595578	49362	5868	820
9	4160984121	10^{10}	9183801602	4615650	16686	6840	1344
10	3141592221	2^{35}	13539813818	5795090	88134	12716	2938
11	2718281829	2^{35}	22939188896	2723830	146116	10782	2914
12	5^{13}	2^{35}	33161885770	2925242	113374	13070	2256
13	5^{15}	2^{35}	22078865098	10274746	167558	5844	2592
14	$2^{23}+2^{12}+5$	2^{35}	167510120	8052254	21476	16802	1630
15	$2^{23}+2^{13}+5$	2^{35}	168231328	5335322	21476	2008	1134
16	$2^{23}+2^{14}+5$	2^{35}	12256151168	5733878	21476	13316	2032
17	$2^{22}+2^{13}+5$	2^{35}	8201443840	1830230	21476	7786	3080
18	$2^{24}+2^{13}+5$	2^{35}	8364058	8364058	21476	16712	1496
19	19935388837	2^{35}	32300850938	705518	22270	9558	2660
20	1175245817	2^{35}	36436418002	7362242	95306	3006	2860
21	17059465	2^{35}	39341117000	9476606	202796	18758	2382
22	$2^{16}+3$	2^{29}	536805386	118	116	116	116
23	1812433253	2^{32}	4326934538	1462856	15082	4866	906
24	1566083941	2^{32}	4659748970	2079590	44902	4652	662
25	69069	2^{32}	4243209856	2072544	52804	6990	242
26	1664525	2^{32}	4938916874	2322494	63712	4092	1038
27	314159269	$2^{31}-1$	1432232969	899290	36985	3427	1144
28	see (39)		$(2^{31}-1)^2$	1.4×10^{12}	643578623	12930027	837632
29	31167285	2^{48}	3.2×10^{14}	4111841446	17341510	306326	59278
30	see the text	2^{64}	8.8×10^{18}	6.4×10^{12}	4.1×10^{9}	45662836	1846368

Thus, in six or fewer dimensions the merit is computed as follows:

$$\mu_2 = \pi\nu_2^2/m, \qquad \mu_3 = \tfrac{4}{3}\pi\nu_3^3/m, \qquad \mu_4 = \tfrac{1}{2}\pi^2\nu_4^4/m,$$
$$\mu_5 = \tfrac{8}{15}\pi^2\nu_5^5/m, \qquad \mu_6 = \tfrac{1}{6}\pi^3\nu_6^6/m. \tag{37}$$

We might say that the multiplier a passes the spectral test if μ_t is 0.1 or more for $2 \le t \le 6$, and it "passes with flying colors" if $\mu_t \ge 1$ for all these t. A low value of μ_t means that we have probably picked a very unfortunate multiplier, since very few lattices will have integer points so close to the origin. Conversely, a high value of μ_t means that we have found an unusually good multiplier for the given m; but it does not mean that the random numbers are necessarily very good, since m might be too small. Only the values ν_t truly indicate the degree of randomness.

($\epsilon = \frac{1}{10}$)

$\lg \nu_2$	$\lg \nu_3$	$\lg \nu_4$	$\lg \nu_5$	$\lg \nu_6$	μ_2	μ_3	μ_4	μ_5	μ_6	Line
4.5	4.5	4.5	4.5	4.4	$2\epsilon^5$	$5\epsilon^4$	0.01	0.34	4.62	1
7.0	7.0	7.0	7.0	4.0	$2\epsilon^6$	$3\epsilon^4$	0.04	4.66	$2\epsilon^3$	2
17.5	1.3	1.0	1.0	1.0	3.14	$2\epsilon^9$	$2\epsilon^9$	$5\epsilon^9$	ϵ^8	3
15.7	10.0	7.4	5.0	5.0	0.27	0.13	0.11	0.01	0.21	4
4.0	2.5	1.9	1.3	1.0	3.36	2.69	3.78	1.81	1.29	5
16.0	10.0	8.0	5.4	4.5	1.44	0.44	1.92	0.07	0.08	6
16.0	9.0	8.3	5.9	5.6	1.35	0.06	4.69	0.35	6.98	7
16.7	10.7	7.8	6.3	4.8	3.39	1.75	1.20	1.39	0.28	8
16.5	11.1	7.0	6.4	5.2	2.89	4.15	0.14	2.04	1.25	9
16.8	11.2	8.2	6.8	5.8	1.24	1.70	1.12	2.79	3.81	10
17.2	10.7	8.6	6.7	5.8	2.10	0.55	3.15	1.85	3.72	11
17.5	10.7	8.4	6.8	5.6	3.03	0.61	1.85	2.99	1.73	12
17.2	11.6	8.7	6.3	5.7	2.02	4.02	4.03	0.40	2.62	13
13.7	11.5	7.2	7.0	5.3	0.02	2.79	0.07	5.61	0.65	14
13.7	11.2	7.2	5.5	5.1	0.02	1.50	0.07	0.03	0.22	15
16.8	11.2	7.2	6.9	5.5	1.12	1.67	0.07	3.13	1.26	16
16.5	10.4	7.2	6.5	5.8	0.75	0.30	0.07	0.82	4.39	17
11.5	11.5	7.2	7.0	5.3	$8\epsilon^4$	2.95	0.07	5.53	0.50	18
17.5	9.7	7.2	6.6	5.7	2.95	0.07	0.07	1.37	2.83	19
17.5	11.4	8.3	5.8	5.7	3.33	2.44	1.30	0.08	3.52	20
17.6	11.6	8.8	7.1	5.6	3.60	3.56	5.91	7.38	2.03	21
14.5	3.4	3.4	3.4	3.4	3.14	ϵ^5	ϵ^4	ϵ^3	0.02	22
16.0	10.2	6.9	6.1	4.9	3.16	1.73	0.26	2.02	0.89	23
16.1	10.5	7.7	6.1	4.7	3.41	2.92	2.32	1.81	0.35	24
16.0	10.5	7.8	6.4	4.0	3.10	2.91	3.20	5.01	0.02	25
16.1	10.6	8.0	6.0	5.0	3.61	3.45	4.66	1.31	1.35	26
15.2	9.9	7.6	5.9	5.1	2.10	1.66	3.14	1.69	3.60	27
31.0	20.2	15.6	11.8	9.8	3.14	1.49	0.44	0.69	0.66	28
24.1	16.0	12.0	9.1	7.9	3.60	3.92	5.27	0.97	3.82	29
31.5	21.3	16.0	12.7	10.4	1.50	3.68	4.52	4.02	1.76	30
upper bounds from (40):					3.63	5.92	9.87	14.89	23.87	

Table 1 shows what sorts of values occur in typical sequences. Each line of the table considers a particular generator, and lists ν_t, μ_t, and the "number of bits of accuracy" $\lg \nu_t$. Lines 1 through 4 show the generators that were the subject of Figs. 2 and 5 in Section 3.3.1. The generators in lines 1 and 2 suffer from too small a multiplier; a diagram like Fig. 8 will have a nearly vertical "stripes" when a is small. The terrible generator in line 3 has a good μ_2 but very poor μ_3 and μ_4; like nearly all generators of potency 2, it has $\nu_3 = \sqrt{6}$ and $\nu_4 = 2$ (see exercise 3). Line 4 shows a "random" multiplier; this generator has satisfactorily passed numerous empirical tests for randomness, but it does not have especially high values of $\mu_2, \ldots, \mu_6$. In fact, the value of μ_5 flunks our criterion.

Line 5 shows the generator of Fig. 8. It passes the spectral test with very high-flying colors, when μ_2 through μ_6 are considered, but of course m is so

small that the numbers can hardly be called random; the ν_t values are terribly low.

Line 6 is the generator discussed above; line 7 is a similar example, having an abnormally low value of μ_3. Line 8 shows a nonrandom multiplier for the same modulus m; all of its partial quotients are 1, 2, or 3. Such multipliers have been suggested by I. Borosh and H. Niederreiter because the Dedekind sums are likely to be especially small and because they produce best results in the two-dimensional serial test (cf. Section 3.3.3 and exercise 30). The particular example in line 8 has only one '3' as a partial quotient; there is no multiplier congruent to 1 modulo 20 whose partial quotients with respect to 10^{10} are only 1's and 2's. The generator in line 9 shows another multiplier chosen with malice aforethought, following a suggestion by A. G. Waterman that guarantees a reasonably high value of μ_2 (see exercise 11).

Lines 10 through 21 of Table 1 show further examples with $m = 2^{35}$, beginning with some random multipliers. The generators in lines 12 and 13 are reminders of the good old days—they were once used extensively since O. Taussky first suggested them in the early 1950s. Lines 14 through 18 show various multipliers of maximum potency having only four 1's in their binary representation. The point of having few 1's is to replace multiplication by a few additions, but only line 16 comes near to being passable. Since these multipliers satisfy $(a - 5)^3 \bmod 2^{35} = 0$, all five of them achieve ν_4 at the same point $(x_1, x_2, x_3, x_4) = (-125, 75, -15, 1)$. Another curiosity is the high value of μ_3 following a very low μ_2 in line 18 (see exercise 8). Lines 19 and 20 are respectively the Borosh–Niederreiter and Waterman multipliers for modulus 2^{35}; and line 21 was found by M. Lavaux and F. Janssens, in a computer search for spectrally good multipliers having a very high μ_2.

Lines 22 through 28 apply to System/370 and other machines with 32-bit words; in this case the comparatively small word size calls for comparatively greater care. Line 22 is, regrettably, the generator that has actually been used on such machines in most of the world's scientific computing centers for about a decade; its very name RANDU is enough to bring dismay into the eyes and stomachs of many computer scientists! The actual generator is defined by

$$X_0 \text{ odd}, \qquad X_{n+1} = (65539X_n) \bmod 2^{31}, \tag{38}$$

and exercise 20 indicates that 2^{29} is the appropriate modulus for the spectral test. Since $9X_n + 6X_{n+2} + X_{n+2} \equiv 0$ (modulo 2^{31}), the generator fails most three-dimensional criteria for randomness, and it should never have been used. Almost any multiplier $\equiv 5$ (modulo 8) would be better. (A curious fact about RANDU, noticed by R. W. Gosper, is that $\nu_4 = \nu_5 = \nu_6 = \nu_7 = \nu_8 = \nu_9 = \sqrt{116}$, hence μ_9 is a spectacular 11.98.) Lines 23 and 24 are the Borosh–Niederreiter and Waterman multipliers for modulus 2^{32}, lines 26 and 29 were found by Lavaux and Janssens, and line 30 (whose excellent multiplier 6364136223846793005 is too big to fit in the column) is due to C. E. Haynes. Line 25 was nominated by George Marsaglia as "a candidate for the best of all multipliers," after a computer search

in dimensions 2 through 5, partly because it is easy to remember. Line 27 uses a random primitive root, modulo the prime $2^{31} - 1$, as multiplier. Line 28 is for the sequence

$$X_n = (271828183X_{n-1} - 314159269X_{n-2}) \bmod (2^{31} - 1), \tag{39}$$

which can be shown to have period length $(2^{31} - 1)^2 - 1$; it has been analyzed with the generalized spectral test of exercise 24.

Theoretical upper bounds on μ_t, which can never be transcended for any m, are shown just below Table 1; it is known that every lattice with m points per unit volume has

$$\nu_t \le \gamma_t m^{1/t}, \tag{40}$$

where γ_t takes the respective values

$$(4/3)^{1/4}, \quad 2^{1/6}, \quad 2^{1/4}, \quad 2^{3/10}, \quad (64/3)^{1/12}, \quad 2^{3/7}, \quad 2^{1/2} \tag{41}$$

for $t = 2, \ldots, 8$. (See exercise 9 and J. W. S. Cassels, *Introduction to the Geometry of Numbers* (Berlin: Springer, 1959), p. 332.) These bounds hold for lattices generated by vectors with arbitrary real coordinates. For example, the optimum lattice for $t = 2$ is hexagonal, and it is generated by vectors of length $2/\sqrt{3m}$ that form two sides of an equilateral triangle. In three dimensions the optimum lattice is generated by vectors V_1, V_2, V_3 that can be rotated into the form $(v, v, -v)$, $(v, -v, v)$, $(-v, v, v)$, where $v = 1/\sqrt[3]{4m}$.

***F. Relation to the serial test.** In a series of important papers published during the 1970s, Harald Niederreiter has shown how to analyze the distribution of the t-dimensional vectors (1) by means of exponential sums. One of the main consequences of his theory is that the serial test in several dimensions will be passed by any generator that passes the spectral test, even when we consider only a sufficiently large part of the period instead of the whole period. We shall now turn briefly to a study of his interesting methods, in the case of linear congruential sequences (X_0, a, c, m) of period length m.

The first idea we need is the notion of *discrepancy* in t dimensions, a quantity that we shall define as the difference between the expected number and the actual number of t-dimensional vectors $(x_n, x_{n+1}, \ldots, x_{n+t-1})$ falling into a hyperrectangular region, maximized over all such regions. To be precise, let $\langle x_n \rangle$ be a sequence of integers in the range $0 \le x_n < m$. We define

$$D_N^{(t)} = \max_R \left| \frac{\text{number of } (x_n, \ldots, x_{n+t-1}) \text{ in } R \text{ for } 0 \le n < N}{N} - \frac{\text{volume of } R}{m^t} \right| \tag{42}$$

where R ranges over all sets of points of the form

$$R = \{ (y_1, \ldots, y_t) \mid \alpha_1 \le y_1 < \beta_1, \ldots, \alpha_t \le y_t < \beta_t \}; \tag{43}$$

here α_j and β_j are integers in the range $0 \le \alpha_j < \beta_j \le m$, for $1 \le j \le t$. The volume of R is clearly $(\beta_1 - \alpha_1) \ldots (\beta_t - \alpha_t)$. To get the discrepancy $D_N^{(t)}$, we imagine looking at all these sets R and finding the one with the greatest excess or deficiency of points $(x_n, \ldots, x_{n+t-1})$.

An upper bound for the discrepancy can be found by using exponential sums. Let $\omega = e^{2\pi i/m}$ be a primitive mth root of unity. If $(x_1, \ldots, x_t)$ and $(y_1, \ldots, y_t)$ are two vectors with all components in the range $0 \le x_j, y_j < m$, we have

$$\sum_{0 \le u_1, \ldots, u_t < m} \omega^{(x_1 - y_1)u_1 + \cdots + (x_t - y_t)u_t} = \begin{cases} m^t, & \text{if } (x_1, \ldots, x_t) = (y_1, \ldots, y_t); \\ 0, & \text{if } (x_1, \ldots, x_t) \neq (y_1, \ldots, y_t). \end{cases}$$

Therefore the number of vectors $(x_n, \ldots, x_{n+t-1})$ in R for $0 \le n < N$, when R is defined by (43), can be expressed as

$$\frac{1}{m^t} \sum_{0 \le n < N} \sum_{0 \le u_1, \ldots, u_t < m} \omega^{x_n u_1 + \cdots + x_{n+t-1} u_t} \times \sum_{\alpha_1 \le y_1 < \beta_1} \cdots \sum_{\alpha_t \le y_t < \beta_t} \omega^{-(y_1 u_1 + \cdots + y_t u_t)}.$$

When $u_1 = \cdots = u_t = 0$ in this sum, we get N/m^t times the volume of R; hence we can express $D_N^{(t)}$ as

$$\max_R \left| \frac{1}{Nm^t} \sum_{0 \le n < N} \sum_{\substack{0 \le u_1, \ldots, u_t < m \\ (u_1, \ldots, u_t) \neq (0, \ldots, 0)}} \omega^{x_n u_1 + \cdots + x_{n+t-1} u_t} \times \sum_{\alpha_1 \le y_1 < \beta_1} \cdots \sum_{\alpha_t \le y_t < \beta_t} \omega^{-(y_1 u_1 + \cdots + y_t u_t)} \right|.$$

Since complex numbers satisfy $|w + z| \le |w| + |z|$ and $|wz| = |w|\,|z|$, it follows that

$$\begin{aligned} D_N^{(t)} &\le \max_R \frac{1}{m^t} \sum_{\substack{0 \le u_1, \ldots, u_t < m \\ (u_1, \ldots, u_t) \neq (0, \ldots, 0)}} \left| \sum_{\alpha_1 \le y_1 < \beta_1} \cdots \sum_{\alpha_t \le y_t < \beta_t} \omega^{-(y_1 u_1 + \cdots + y_t u_t)} \right| g(u_1, \ldots, u_t) \\ &\le \frac{1}{m^t} \sum_{\substack{0 \le u_1, \ldots, u_t < m \\ (u_1, \ldots, u_t) \neq (0, \ldots, 0)}} \max_R \left| \sum_{\alpha_1 \le y_1 < \beta_1} \cdots \sum_{\alpha_t \le y_t < \beta_t} \omega^{-(y_1 u_1 + \cdots + y_t u_t)} \right| g(u_1, \ldots, u_t) \\ &= \sum_{\substack{0 \le u_1, \ldots, u_t < m \\ (u_1, \ldots, u_t) \neq (0, \ldots, 0)}} f(u_1, \ldots, u_t)\, g(u_1, \ldots, u_t), \end{aligned} \tag{44}$$

where

$$g(u_1, \ldots, u_t) = \left| \frac{1}{N} \sum_{0 \le n < N} \omega^{x_n u_1 + \cdots + x_{n+t-1} u_t} \right|;$$

$$\begin{aligned} f(u_1, \ldots, u_t) &= \max_R \frac{1}{m^t} \left| \sum_{\alpha_1 \le y_1 < \beta_1} \cdots \sum_{\alpha_t \le y_t < \beta_t} \omega^{-(y_1 u_1 + \cdots + y_t u_t)} \right| \\ &= \max_R \left| \frac{1}{m} \sum_{\alpha_1 \le y_1 < \beta_1} \omega^{-u_1 y_1} \right| \cdots \left| \frac{1}{m} \sum_{\alpha_t \le y_t < \beta_t} \omega^{-u_t y_t} \right|. \end{aligned}$$

Both f and g can be further simplified in order to get a good upper bound on $D_N^{(t)}$. We have

$$\left|\frac{1}{m}\sum_{\alpha\le y<\beta}\omega^{-uy}\right| = \left|\frac{1}{m}\,\frac{\omega^{-\beta u}-\omega^{-\alpha u}}{\omega^{-u}-1}\right| \le \frac{2}{m|\omega^u-1|} = \frac{1}{m\sin(\pi u/m)}$$

when $u \not\equiv 0$, and the sum is ≤ 1 when $u = 0$; hence

$$f(u_1,\ldots,u_t) \le r(u_1,\ldots,u_t), \tag{45}$$

where

$$r(u_1,\ldots,u_t) = \prod_{\substack{1\le k\le t\\ u_k\not\equiv 0}} \frac{1}{m\sin(\pi u_k/m)}. \tag{46}$$

Furthermore, when $\langle x_n\rangle$ is generated modulo m by a linear conguential sequence, we have

$$\begin{aligned} x_n u_1 + \cdots + x_{n+t-1}u_t &= x_n u_1 + (ax_n + c)u_2 + \cdots \\ &\qquad + (a^{t-1}x_n + c(a^{t-2}+\cdots+1))u_t \\ &= (u_1 + au_2 + \cdots + a^{t-1}u_t)x_n + h(u_1,\ldots,u_t) \end{aligned}$$

where $h(u_1,\ldots,u_t)$ is independent of n; hence

$$g(u_1,\ldots,u_t) = \left|\frac{1}{N}\sum_{0\le n<N}\omega^{q(u_1,\ldots,u_t)x_n}\right|, \tag{47}$$

where

$$q(u_1,\ldots,u_t) = u_1 + au_2 + \cdots + a^{t-1}u_t. \tag{48}$$

Now *here is where the connection to the spectral test comes in*: We will show that the sum $g(u_1,\ldots,u_t)$ is rather small unless $q(u_1,\ldots,u_t) \equiv 0$ (modulo m), i.e., unless $(u_1,\ldots,u_t)$ is a solution to (15). Furthermore exercise 27 shows that $r(u_1,\ldots,u_t)$ is rather small when $(u_1,\ldots,u_t)$ is a "large" solution to (15). Hence the discrepancy $D_N^{(t)}$ will be rather small when (15) has only "large" solutions, i.e., when the spectral test is passed. All that remains is to quantify these qualitative statements by making careful calculations.

In the first place, let's consider the size of $g(u_1,\ldots,u_t)$. When $N = m$, so that the sum (47) is over an entire period, we have $g(u_1,\ldots,u_t) = 0$ except when $(u_1,\ldots,u_t)$ satisfies (15), so the discrepancy is bounded above in this case by the sum of $r(u_1,\ldots,u_t)$ taken over all the nonzero solutions of (15). But let's consider also what happens in a sum like (47) when N is less than m and $q(u_1,\ldots,u_t)$ is not a multiple of m. We have

$$\begin{aligned} \frac{1}{N}\sum_{0\le n<N}\omega^{x_n} &= \frac{1}{N}\sum_{0\le n<N}\frac{1}{m}\sum_{0\le k<m}\omega^{-nk}\sum_{0\le j<m}\omega^{x_j+jk} \\ &= \frac{1}{N}\sum_{0\le k<m}\left(\frac{1}{m}\sum_{0\le n<N}\omega^{-nk}\right)S_{k0}, \end{aligned} \tag{49}$$

where

$$S_{kl} = \sum_{0\le j<m} \omega^{x_{j+l}+jk}. \tag{50}$$

Now $S_{kl} = \omega^{-lk} S_{k0}$, so $|S_{kl}| = |S_{k0}|$ for all l, and we can calculate this common value by further exponential-summery:

$$\begin{aligned}
|S_{k0}|^2 &= \frac{1}{m} \sum_{0\le l<m} |S_{kl}|^2 \\
&= \frac{1}{m} \sum_{0\le l<m} \sum_{0\le j<m} \omega^{x_{j+l}+jk} \sum_{0\le i<m} \omega^{-x_{i+l}-ik} \\
&= \frac{1}{m} \sum_{0\le i,j<m} \omega^{(j-i)k} \sum_{0\le l<m} \omega^{x_{j+l}-x_{i+l}} \\
&= \frac{1}{m} \sum_{0\le i<m} \sum_{i\le j<m+i} \omega^{(j-i)k} \sum_{0\le l<m} \omega^{(a^{j-i}-1)x_{i+l}+(a^{j-i}-1)c/(a-1)}.
\end{aligned}$$

Let s be minimum such that $a^s \equiv 1$ (modulo m), and let

$$s' = (a^s - 1)c/(a-1) \bmod m.$$

Then s is a divisor of m, and $x_{n+js} \equiv x_n + js'$ (modulo m). The sum on l vanishes unless $j - i$ is a multiple of s, so we find that

$$|S_{k0}|^2 = m \sum_{0\le j<m/s} \omega^{jsk+js'}.$$

We have $s' = q's$ where q' is relatively prime to m (cf. exercise 3.2.1–21), so it turns out that

$$|S_{k0}| = \begin{cases} 0, & \text{if } k + q' \not\equiv 0 \text{ (modulo } m/s); \\ m/\sqrt{s}, & \text{if } k + q' \equiv 0 \text{ (modulo } m/s). \end{cases} \tag{51}$$

Putting this information back into (49), and recalling the derivation of (45), shows that

$$\left| \frac{1}{N} \sum_{0\le n<N} \omega^{x_n} \right| \le \frac{m}{N\sqrt{s}} \sum_k r(k), \tag{52}$$

where the sum is over $0 < k < m$ such that $k + q' \equiv 0$ (modulo m/s). Exercise 25 now can be used to estimate the remaining sum, and we find that

$$\left| \frac{1}{N} \sum_{0\le n<N} \omega^{x_n} \right| \le \frac{2}{\pi} \frac{\sqrt{s}}{N} \ln s + O\left(\frac{m}{N\sqrt{s}}\right). \tag{53}$$

The same bound can be used to estimate $|N^{-1} \sum_{0\le n<N} \omega^{qx_n}|$ for any $q \not\equiv 0$ (modulo m), since the effect is to replace m in this derivation by a divisor of m.

In fact, the upper bound gets even smaller when q has a factor in common with m, since s and $m/\sqrt{s}$ generally become smaller. (See exercise 26.)

We have now proved that the $g(u_1,\ldots,u_t)$ part of our upper bound (44) on the discrepancy is small, if N is large enough and if $(u_1,\ldots,u_t)$ does not satisfy the spectral test congruence (15). Exercise 27 proves that the $f(u_1,\ldots,u_t)$ part of our upper bound is small, when summed over all the nonzero vectors $(u_1,\ldots,u_t)$ satisfying (15), provided that all such vectors are far away from $(0,\ldots,0)$. Putting these results together leads to the following theorem of Niederreiter:

Theorem N. *Let $\langle X_n\rangle$ be a linear congruential sequence (X_0,a,c,m) of period length m, and let s be the least positive integer such that $a^s\equiv 1$ (modulo m). Let ν_t be the t-dimensional accuracy of $\langle X_n\rangle$, as determined by the spectral test. Then the t-dimensional discrepancy $D_N^{(t)}$ determined by the first N values of $\langle X_n\rangle$, as defined in (42), satisfies*

$$D_N^{(t)} = O\biggl(\frac{\sqrt{s}\log s\,(\log m)^t}{N}\biggr) + O\biggl(\frac{m(\log m)^t}{N\sqrt{s}}\biggr) + O\bigl((\log m)^t\, r_{\max}\bigr); \tag{54}$$

$$D_m^{(t)} = O\bigl((\log m)^t\, r_{\max}\bigr). \tag{55}$$

Here $r_{\max}$ is the maximum value of the quantity $r(u_1,\ldots,u_t)$ defined in (46), taken over all nonzero integer vectors $(u_1,\ldots,u_t)$ satisfying (15).

Proof. The first two O terms in (54) come from vectors $(u_1,\ldots,u_t)$ in (44) that do not satisfy (15), since exercise 25 proves that $f(u_1,\ldots,u_t)$ summed over *all* $(u_1,\ldots,u_t)$ is $O(((2/\pi)\ln m)^t)$ and exercise 26 bounds each $g(u_1,\ldots,u_t)$. (These terms are missing from (55) since $g(u_1,\ldots,u_t)=0$ in that case.) The remaining O term in (54) and (55) comes from nonzero vectors $(u_1,\ldots,u_t)$ that do satisfy (15), using the bound derived in exercise 27. (By examining this proof carefully, we could replace each O in these formulas by an explicit function of t.) ∎

Eq. (55) relates to the serial test in t dimensions over the entire period, while Eq. (54) gives us useful information about the distribution of the first N generated values when N is less than m, provided that N is not too small. Note that (54) will guarantee low discrepancy only when s is sufficiently large, otherwise the $m/\sqrt{s}$ term will dominate. If $m = p_1^{e_1}\ldots p_r^{e_r}$ and $\gcd(a-1,m) = p_1^{f_1}\ldots p_r^{f_r}$, then s equals $p_1^{e_1-f_1}\ldots p_r^{e_r-f_r}$ (cf. Lemma 3.2.1.2P); thus, the largest values of s correspond to high potency. In the common case $m=2^e$ and $a\equiv 5$ (modulo 8), we have $s=\frac{1}{4}m$, so $D_N^{(t)}$ is $O(\sqrt{m}\,(\log m)^{t+1}/N)+O\bigl((\log m)^t r_{\max}\bigr)$. It is not difficult to prove that $r_{\max}\le\sqrt{2}/\nu_t$ unless ν_t is very small (see exercise 29); therefore Eq. (54) says in particular that the discrepancy will be low in t dimensions if the spectral test is passed and if N is somewhat larger than $\sqrt{m}\,(\log m)^{t+1}$.

In a sense Theorem N is almost too strong, for the result in exercise 30 shows that linear congruential sequences like those in lines 8, 19, and 23 of Table 1 have a discrepancy of order $(\log m)^2/m$ in two dimensions. The discrepancy

in this case is extremely small in spite of the fact that there are parallelogram-shaped regions of area $\approx 1/\sqrt{m}$ containing no points (U_n, U_{n+1}). The fact that discrepancy can change so drastically when the points are rotated warns us that the serial test may not be as meaningful a measure of randomness as the rotation-invariant spectral test.

G. Historical remarks. In 1959, while deriving upper bounds for the error in the evaluation of t-dimensional integrals by the Monte Carlo method, N. M. Korobov devised a way to rate the multiplier of a linear congruential sequence. His formula (which is rather complicated) is related to the spectral test since it is strongly influenced by "small" solutions to (15); but it is not quite the same. Korobov's test has been the subject of an extensive literature, surveyed by Kuipers and Niederreiter in *Uniform Distribution of Sequences* (New York: Wiley, 1974), §2.5.

The spectral test was originally formulated by R. R. Coveyou and R. D. MacPherson [*JACM* **14** (1967), 100–119], who introduced it in an interesting indirect way. Instead of working with the grid structure of successive points, they considered random number generators as sources of t-dimensional "waves." The numbers $\sqrt{x_1^2 + \cdots + x_t^2}$ such that $x_1 + \cdots + a^{t-1}x_t \equiv 0$ (modulo m) in their original treatment were the wave "frequencies," or points in the "spectrum" defined by the random number generator, with low-frequency waves being the most damaging to randomness; hence the name *spectral test*. Coveyou and MacPherson introduced a procedure analogous to Algorithm S for performing their test, based on the principle of Lemma A. However, the original procedure (which used matrices UU^T and VV^T instead of U and V) dealt with extremely large numbers; the idea of working directly with U and V was independently suggested by F. Janssens and by U. Dieter.

Several other authors pointed out that the spectral test could be understood in far more concrete terms; by introducing the study of the grid and lattice structures corresponding to linear congruential sequences, the fundamental limitations on randomness became graphically clear. See G. Marsaglia, *Proc. Nat. Acad. Sci.* **61** (1968), 25–28; W. W. Wood, *J. Chem. Phys.* **48** (1968), 427; R. R. Coveyou, *Studies in Applied Math.* **3** (1969), 70–112; W. A. Beyer, R. B. Roof, and D. Williamson, *Math. Comp.* **25** (1971), 345–360; G. Marsaglia and W. A. Beyer, *Applications of Number Theory to Numerical Analysis*, ed. by S. K. Zaremba (New York: Academic Press, 1972), 249–285, 361–370.

Harald Niederreiter's papers concerning the use of exponential sums to study the distribution of linear congruential sequences have appeared in *Math. Comp.* **26** (1972), 793–795; **28** (1974), 1117–1132; **30** (1976), 571–597; *Advances in Math.* **26** (1977), 99-181 [this is the most important paper of the series]; and *Bull. Amer. Math. Soc.* **84** (1978), 273–274, 957–1041 [this one summarizes the others and contains an extensive bibliography].

EXERCISES

1. [*M10*] To what does the spectral test reduce in *one* dimension? (In other words,

what happens when $t = 1$?)

2. [*HM20*] Let $V_1, \ldots, V_t$ be linearly independent vectors in t-space, let L_0 be the lattice of points defined by (10), and let $U_1, \ldots, U_t$ be defined by (19). Prove that the maximum distance between $(t-1)$-dimensional hyperplanes, over all families of parallel hyperplanes that cover L_0, is $1/\min\{ f(x_1, \ldots, x_t) \mid (x_1, \ldots, x_t) \neq (0, \ldots, 0) \}$, where f is defined in (17).

3. [*M24*] Determine ν_3 and ν_4 for all linear congruential generators of potency 2 and period length m.

▶ **4.** [*M23*] Let u_{11}, u_{12}, u_{21}, u_{22} be elements of a 2×2 integer matrix such that $u_{11} + au_{12} \equiv u_{21} + au_{22} \equiv 0$ (modulo m) and $u_{11}u_{22} - u_{21}u_{12} = m$. (a) Prove that all integer solutions (y_1, y_2) to the congruence $y_1 + ay_2 \equiv 0$ (modulo m) have the form $(y_1, y_2) = (x_1u_{11} + x_2u_{21}, x_1u_{12} + x_2u_{22})$ for integer x_1, x_2. (b) If, in addition, $2|u_{11}u_{21} + u_{12}u_{22}| \leq u_{11}^2 + u_{12}^2 \leq u_{21}^2 + u_{22}^2$, prove that $(y_1, y_2) = (u_{11}, u_{12})$ minimizes $y_1^2 + y_2^2$ over all nonzero solutions to the congruence.

5. [*M30*] Prove that steps S1 through S3 of Algorithm S correctly perform the spectral test in two dimensions. [*Hint:* See exercise 4, and prove that $(h' + h)^2 + (p' + p)^2 \geq h^2 + p^2$ at the beginning of step S2.]

6. [*M30*] Let $a_0, a_1, \ldots, a_{t-1}$ be the partial quotients of a/m as defined in Section 3.3.3, and let $A = \max_{0 \leq j < t} a_j$. Prove that $\mu_2 > 2\pi/(A + 1 + 1/A)$.

7. [*HM22*] Prove that "question (a)" and "question (b)" of the text have the same solution for real values of $q_1, \ldots, q_{j-1}, q_{j+1}, \ldots, q_t$ (cf. (24), (26)).

8. [*M16*] Line 18 of Table 1 has a very low value of μ_2, yet μ_3 is quite satisfactory. What is the highest possible value of μ_3 when $\mu_2 = 10^{-6}$ and $m = 10^{10}$?

9. [*HM32*] (C. Hermite, 1846.) Let $f(x_1, \ldots, x_t)$ be a positive definite quadratic form, defined by the matrix U as in (17), and let θ be the minimum value of f at nonzero integer points. Prove that $\theta \leq (\frac{4}{3})^{(t-1)/2} |\det U|^{2/t}$. [*Hints:* If W is any integer matrix of determinant 1, the matrix WU defines a form equivalent to f; and if S is any orthogonal matrix (i.e., $S^{-1} = S^T$), the matrix US defines a form identically equal to f. Show that there is an equivalent form g whose minimum θ occurs at $(1, 0, \ldots, 0)$. Then prove the general result by induction on t, writing $g(x_1, \ldots, x_t) = \theta(x_1 + \beta_2 x_2 + \cdots + \beta_t x_t)^2 + h(x_2, \ldots, x_t)$ where h is a positive definite quadratic form in $t - 1$ variables.]

10. [*M28*] Let (y_1, y_2) be relatively prime integers such that $y_1 + ay_2 \equiv 0$ (modulo m) and $y_1^2 + y_2^2 < \sqrt{4/3}\, m$. Show that there exist integers (u_1, u_2) such that $u_1 + au_2 \equiv 0$ (modulo m), $u_1y_2 - u_2y_1 = m$, $2|u_1y_1 + u_2y_2| \leq \min(u_1^2 + u_2^2, y_1^2 + y_2^2)$, and $(u_1^2 + u_2^2) \times (y_1^2 + y_2^2) \geq m^2$. (Hence by exercise 4, $\nu_2^2 = \min(u_1^2 + u_2^2, y_1^2 + y_2^2)$.)

▶ **11.** [*HM30*] (Alan G. Waterman, 1974.) Invent a reasonably efficient procedure that computes multipliers $a \equiv 1$ (modulo 4) for which there exists a relatively prime solution to the congruence $y_1 + ay_2 \equiv 0$ (modulo m) with $y_1^2 + y_2^2 = \sqrt{4/3}\, m - \epsilon$, where $\epsilon > 0$ is as small as possible, given $m = 2^e$. (By exercise 10, this choice of a will guarantee that $\nu_2^2 \geq m^2/(y_1^2 + y_2^2) > \sqrt{3/4}\, m$, and there is a chance that ν_2^2 will be near its optimum value $\sqrt{4/3}\, m$. In practice we will compute several such multipliers having small ϵ, choosing the one with best spectral values ν_2, ν_3, ...)

12. [*HM23*] Prove, without geometrical handwaving, that any solution to the text's "question (b)" must also satisfy the set of equations (26).

13. [*HM22*] Lemma A uses the fact that U is nonsingular to prove that a positive definite quadratic form attains a definite, nonzero minimum value at nonzero integer points. Show that this hypothesis is necessary, by exhibiting a quadratic form (19) whose matrix of coefficients is singular, and for which the values of $f(x_1, \ldots, x_t)$ get arbitrarily near zero (but never reach it) at nonzero integer points $(x_1, \ldots, x_t)$.

14. [*24*] Perform Algorithm S by hand, for $m = 100$, $a = 41$, $T = 3$.

▶ **15.** [*M20*] Let U be an integer vector satisfying (15). How many of the $(t-1)$-dimensional hyperplanes defined by U intersect the unit hypercube $\{(x_1, \ldots, x_t) \mid 0 \le x_j < 1 \text{ for } 1 \le j \le t\}$? (This is approximately the number of hyperplanes in the family that will suffice to cover L_0.)

16. [*M30*] (U. Dieter.) Show how to modify Algorithm S in order to calculate the minimum number N_t of parallel hyperplanes intersecting the unit hypercube as in exercise 15, over all U satisfying (15). [*Hint:* What are appropriate analogs to positive definite quadratic forms and to Lemma A?]

17. [*20*] Modify Algorithm S so that, in addition to computing the quantities ν_t, it outputs all integer vectors $(u_1, \ldots, u_t)$ satisfying (15) such that $u_1^2 + \cdots + u_t^2 = \nu_t^2$, for $2 \le t \le T$.

18. [*M30*] (a) Let $m = 2^e$, where e is even. By considering "combinatorial matrices," i.e., matrices whose elements have the form $y + x\delta_{ij}$ (cf. exercise 1.2.3–39), find 3×3 matrices of integers U and V satisfying (29) such that the transformation of step S5 does nothing for any j, but the corresponding values of z_k in (31) are so huge that exhaustive search is out of the question. (The matrix U need not satisfy (28), we are interested here in *arbitrary* positive definite quadratic forms of determinant m.) (b) Although transformation (23) is of no use for the matrices constructed in (a), find another transformation that does produce a substantial reduction.

▶ **19.** [*HM25*] Suppose step S5 were changed slightly, so that a transformation with $q = 1$ would be performed when $2V_i \cdot V_j = V_j \cdot V_j$. (Thus, $q = \lfloor (V_i \cdot V_j \,/\, V_j \cdot V_j) + \frac{1}{2} \rfloor$ in all cases.) Would it be possible for Algorithm S to get into an infinite loop?

20. [*M21*] Discuss how to carry out an appropriate spectral test for linear congruential sequences having $c = 0$, X_0 odd, $m = 2^e$, $a \bmod 8 = 5$.

21. [*M20*] (R. W. Gosper.) A certain application uses random numbers in batches of four, but "throws away" the second of each set. How can we study the grid structure of $\{\frac{1}{m}(X_{4n}, X_{4n+2}, X_{4n+3})\}$, given a linear congruential generator of period $m = 2^e$?

22. [*M46*] What is the best upper bound on μ_3, given that μ_2 is very near its maximum value $\sqrt{4/3}\,\pi$? What is the best upper bound on μ_2, given that μ_3 is very near its maximum value $\frac{4}{3}\pi\sqrt{2}$?

23. [*M46*] Let U_i, V_j be vectors of real numbers with $U_i \cdot V_j = \delta_{ij}$ for $1 \le i, j \le t$, and such that $U_i \cdot U_i = 1$, $2|U_i \cdot U_j| \le 1$, $2|V_i \cdot V_j| \le V_j \cdot V_j$ for $i \ne j$. How large can $V_1 \cdot V_1$ be? (This question relates to the bounds in step S8, if both (23) and the transformation of exercise 18(b) fail to make any reductions. The maximum value known to be achievable is $(n+2)/3$, which occurs when $U_1 = I_1$, $U_j = \frac{1}{2}I_1 + \frac{1}{2}\sqrt{3}\,I_j$, $V_1 = I_1 - (I_2 + \cdots + I_n)/\sqrt{3}$, $V_j = 2I_j/\sqrt{3}$, for $2 \le j \le n$, where $(I_1, \ldots, I_n)$ is the identity matrix; this construction is due to B. V. Alekseev [*Matematicheskie Zametki*, to appear].)

▶ **24.** [*M28*] Generalize the spectral test to second-order sequences of the form $X_n = (aX_{n-1} + bX_{n-2}) \bmod p$, having period length $p^2 - 1$. (Cf. Eq. 3.2.2–8.) How should Algorithm S be modified?

25. [*HM24*] Let d be a divisor of m and let $0 \le q < d$. Prove that $\sum r(k)$, summed over all $0 \le k < m$ such that $k \bmod d = q$, is at most $(2/d\pi)\ln(m/d) + O(1)$. (Here $r(k)$ is defined in Eq. (46) when $t = 1$.)

26. [*M22*] Explain why the derivation of (53) leads to a similar bound on

$$\left| N^{-1} \sum_{0 \le n < N} \omega^{qx_n} \right|$$

for $0 < q < m$. Where does the derivation of (53) break down when $m = 1$?

27. [*HM39*] (E. Hlawka, H. Niederreiter.) Let $r(u_1, \ldots, u_m)$ be the function defined in (46). Prove that $\sum r(u_1, \ldots, u_t)$, summed over all $0 \le u_1, \ldots, u_t < m$ such that $(u_1, \ldots, u_t) \not\equiv (0, \ldots, 0)$ and (15) holds, is $O((2\pi \lg m)^t r_{\max})$, where $r_{\max}$ is the maximum term $r(u_1, \ldots, u_t)$ in the sum.

▶ **28.** [*M28*] (H. Niederreiter.) Find an analog of Theorem N for the case $m =$ prime, $c = 0$, $a =$ primitive root modulo m, $X_0 \not\equiv 0$ (modulo m). [*Hint:* Your exponential sums should involve $\zeta = e^{2\pi i/(m-1)}$ as well as ω.] Prove that in this case the "average" primitive root has discrepancy $D_{m-1}^{(t)} = O\big(t(\log m)^t/\phi(m-1)\big)$, hence good primitive roots exist for all m.

29. [*M21*] Prove that quantity $r_{\max}$ of exercise 27 is never larger than $\sqrt{2}/\nu_t$, unless $\nu_t^2 \le 2(t-1)$.

30. [*M33*] (S. K. Zaremba.) Prove that in two dimensions, $r_{\max} \le m/\max(a_1, \ldots, a_s)$, where $a_1, \ldots, a_s$ are the partial quotients obtained when Euclid's algorithm is applied to m and a. [*Hint:* We have $a/m = /a_1, \ldots, a_s/$ in the notation of Section 4.5.3; apply exercise 4.5.3–42.]

31. [*HM47*] (I. Borosh.) Prove that for all sufficiently large m there exists a number a relatively prime to m such that all partial quotients of a/m are ≤ 3. Furthermore the set of all m satisfying this condition but with all partial quotients ≤ 2 has positive density.

3.4. OTHER TYPES OF RANDOM QUANTITIES

WE HAVE NOW SEEN how to make a computer generate a sequence of numbers $U_0, U_1, U_2, \ldots$ that behaves as if each number were independently selected at random between zero and one with the uniform distribution. Applications of random numbers often call for other kinds of distributions, however; for example, if we want to make a random choice from among k alternatives, we want a random *integer* between 1 and k. If some simulation process calls for a random waiting time between occurrences of independent events, a random number with the "exponential distribution" is desired. Sometimes we don't even want random *numbers*—we want a random permutation (i.e., a random arrangement of n objects) or a random combination (i.e., a random choice of k objects from a collection of n).

In principle, any of these other random quantities may be obtained from the uniform deviates $U_0, U_1, U_2, \ldots$. People have devised a number of important "random tricks" that may be used to perform these manipulations efficiently on a computer, and a study of these techniques also gives some insight into the proper use of random numbers in any Monte Carlo application.

It is conceivable that someday somebody will invent a random number generator that produces one of these other random quantities *directly*, instead of getting it indirectly via the uniform distribution. But except for the "random bit" generator described in Section 3.2.2, no direct methods have so far proved to be practical.

The discussion in the following section assumes the existence of a random sequence of uniformly distributed real numbers between zero and one. A new uniform deviate U is generated whenever we need it. These numbers are usually represented in a computer word with the decimal point assumed at the left.

3.4.1. Numerical Distributions

This section summarizes the best techniques known for producing numbers from various important distributions. Many of the methods were originally suggested by John von Neumann in the early 1950s, and they have gradually been improved upon by other people, notably George Marsaglia, J. H. Ahrens, and U. Dieter.

A. Random choices from a finite set. The simplest and most common type of distribution required in practice is a random *integer*. An integer between 0 and 7 can be extracted from three bits of U on a binary computer; in such a case, these bits should be extracted from the *most significant* (left-hand) part of the computer word, since the least significant bits produced by many random number generators are not sufficiently random. (See the discussion in Section 3.2.1.1.)

In general, to get a random integer X between 0 and $k-1$, we can *multiply* by k, and let $X = \lfloor kU \rfloor$. On MIX, we would write

```
LDA U
MUL K
```
(1)

and after these two instructions have been executed the desired integer will appear in register A. If a random integer between 1 and k is desired, we add one to this result. (The instruction "`INCA 1`" would follow (1).)

This method gives each integer with nearly equal probability. There is a slight error because the computer word size is finite (see exercise 2); but the error is quite negligible if k is small, for example if $k/m < 1/10000$.

In a more general situation we might want to give different weights to different integers. Suppose that the value $X = x_1$ is to be obtained with probability p_1, and $X = x_2$ with probability p_2, ..., $X = x_k$ with probability p_k. We can generate a uniform number U and let

$$X = \begin{cases} x_1, & \text{if } 0 \le U < p_1; \\ x_2, & \text{if } p_1 \le U < p_1 + p_2; \\ \vdots & \\ x_k, & \text{if } p_1 + p_2 + \cdots + p_{k-1} \le U < 1. \end{cases} \tag{2}$$

(Note that $p_1 + p_2 + \cdots + p_k = 1$.)

There is a "best possible" way to do the comparisons of U against various values of $p_1 + p_2 + \cdots + p_s$, as implied in (2); this situation is discussed in Section 2.3.4.5. Special cases can be handled by more efficient methods; for example, to obtain one of the eleven values 2, 3, ..., 12 with the respective "dice" probabilities $\frac{1}{36}, \frac{2}{36}, \ldots, \frac{6}{36}, \ldots, \frac{2}{36}, \frac{1}{36}$, we could compute two independent random integers between 1 and 6 and add them together.

However, none of the above techniques is really the fastest general method for selecting $x_1, \ldots, x_k$ with the correct probabilities. An ingenious way to do the trick has been discovered by A. J. Walker [*Electronics Letters* **10**,8 (1974), 127–128; *ACM Trans. Math. Software* **3** (1976), 253–256]. Suppose we form kU and consider the integer part $K = \lfloor kU \rfloor$ and fraction part $V = (kU) \bmod 1$ separately; for example, after the code (1) we will have K in register A and V in register X. Then we can always obtain the desired distribution by doing the operations

$$\text{if } \quad V < P_K \quad \text{then} \quad X \leftarrow x_{K+1} \quad \text{otherwise} \quad X \leftarrow Y_K, \tag{3}$$

for some appropriate tables $(P_0, \ldots, P_{k-1})$ and $(Y_0, \ldots, Y_{k-1})$. Exercise 7 shows how such tables can be computed in general. Walker's method is sometimes called the method of "aliases."

On a binary computer it is usually helpful to assume that k is a power of 2, so that multiplication can be replaced by shifting; this can be done without loss of generality by introducing additional x's that occur with probability zero. For example, let's consider dice again; suppose we want $X = j$ to occur with the following 16 probabilities:

$j =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$p_j =$	0	0	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$	0	0	0

We can do this using (3), if $k = 16$ and $x_j = j$ for $0 \le j < 16$, and if the P and Y tables are set up as follows:

$$\begin{array}{rcccccccccccccccc} P_j = & 0 & 0 & \frac{4}{9} & \frac{8}{9} & 1 & \frac{7}{9} & 1 & 1 & 1 & \frac{7}{9} & \frac{7}{9} & \frac{8}{9} & \frac{4}{9} & 0 & 0 & 0 \\ Y_j = & 5 & 9 & 7 & 4 & * & 6 & * & * & * & 8 & 4 & 7 & 10 & 6 & 7 & 8 \end{array}$$

(When $P_j = 1$, Y_j is not used.) For example, the value 7 occurs with probability $\frac{1}{16} \cdot \big((1 - P_2) + P_7 + (1 - P_{11}) + (1 - P_{14})\big) = \frac{6}{36}$ as required. It is a peculiar way to throw dice, but the results are indistinguishable from the real thing.

B. General methods for continuous distributions. The most general real-valued distribution may be expressed in terms of its "distribution function" $F(x)$, which specifies the probability that a random quantity X will not exceed x:

$$F(x) = \text{probability that } (X \le x). \tag{4}$$

This function always increases monotonically from zero to one; i.e.,

$$F(x_1) \le F(x_2), \quad \text{if } x_1 \le x_2; \qquad F(-\infty) = 0, \qquad F(+\infty) = 1. \tag{5}$$

Examples of distribution functions are given in Section 3.3.1, Fig. 3. If $F(x)$ is continuous and strictly increasing (so that $F(x_1) < F(x_2)$ when $x_1 < x_2$), it takes on all values between zero and one, and there is an *inverse function* $F^{-1}(y)$ such that, for $0 < y < 1$,

$$y = F(x) \qquad \text{if and only if} \qquad x = F^{-1}(y). \tag{6}$$

In general we can compute a random quantity X with the continuous, strictly increasing distribution $F(x)$ by setting

$$X = F^{-1}(U), \tag{7}$$

where U is uniform; this works because the probability that $X \le x$ is the probability that $F^{-1}(U) \le x$, i.e., the probability that $U \le F(x)$, and this is $F(x)$.

The problem now reduces to one of numerical analysis, namely to find good methods for evaluating $F^{-1}(U)$ to the desired accuracy. Numerical analysis lies outside the scope of this seminumerical book; yet there are a number of important shortcuts available to speed up this general approach, and we will consider them here.

In the first place, if X_1 is a random variable having the distribution $F_1(x)$ and if X_2 is an independent random variable with the distribution $F_2(x)$, then

$$\begin{array}{lll} \max(X_1, X_2) & \text{has the distribution} & F_1(x)F_2(x), \\ \min(X_1, X_2) & \text{has the distribution} & F_1(x) + F_2(x) - F_1(x)F_2(x). \end{array} \tag{8}$$

(See exercise 4.) For example, a uniform deviate U has the distribution $F(x) = x$, for $0 \le x \le 1$; if $U_1, U_2, \ldots, U_t$ are independent uniform deviates, then

$\max(U_1, U_2, \ldots, U_t)$ has the distribution function $F(x) = x^t$, for $0 < x \leq 1$. This is the basis of the "maximum-of-t test" given in Section 3.3.2. Note that the inverse function in this case is $F^{-1}(y) = \sqrt[t]{y}$. In the special case $t = 2$, we see therefore that the two formulas

$$X = \sqrt{U} \qquad \text{and} \qquad X = \max(U_1, U_2) \tag{9}$$

will give equivalent distributions to the random variable X, although this is not obvious at first glance. We need not take the square root of a uniform deviate.

The number of tricks like this is endless: *any* algorithm that employs random numbers as input will give a random quantity with *some* distribution as output. The problem is to find general methods for constructing the algorithm, given the distribution function of the output. Instead of discussing such methods in purely abstract terms, we shall study how they can be applied in important cases.

C. The normal distribution. Perhaps the most important nonuniform, continuous distribution is the so-called *normal distribution with mean zero and standard deviation one:*

$$F(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2}\, dt. \tag{10}$$

The significance of this distribution was indicated in Section 1.2.10. Note that the inverse function F^{-1} is not especially easy to compute; but we shall see that several other techniques are available.

(*1*) *The polar method*, due to G. E. P. Box, M. E. Muller, and G. Marsaglia. (See *Annals Math. Stat.* **28** (1958), 610; and Boeing Scientific Res. Lab. report D1-82-0203 (1962).)

Algorithm P (*Polar method for normal deviates*). This algorithm calculates two independent normally distributed variables, X_1 and X_2.

P1. [Get uniform variables.] Generate two independent random variables, U_1, U_2, uniformly distributed between zero and one. Set $V_1 \leftarrow 2U_1 - 1$, $V_2 \leftarrow 2U_2 - 1$. (Now V_1 and V_2 are uniformly distributed between -1 and $+1$. On most computers it will be preferable to have V_1 and V_2 represented in floating point form at this point.)

P2. [Compute S.] Set $S \leftarrow V_1^2 + V_2^2$.

P3. [Is $S \geq 1$?] If $S \geq 1$, return to step P1. (Steps P1 through P3 are executed 1.27 times on the average, with a standard deviation of 0.587; see exercise 6.)

P4. [Compute X_1, X_2.] Set X_1, X_2 according to the following two equations:

$$X_1 = V_1 \sqrt{\frac{-2 \ln S}{S}}, \qquad X_2 = V_2 \sqrt{\frac{-2 \ln S}{S}}. \tag{11}$$

These are the normally distributed variables desired. ∎

To prove the validity of this method, we use elementary analytic geometry and calculus: If $S < 1$ in step P3, the point in the plane with Cartesian coordinates (V_1, V_2) is a *random point uniformly distributed inside the unit circle.* Transforming to polar coordinates $V_1 = R\cos\Theta$, $V_2 = R\sin\Theta$, we find $S = R^2$, $X_1 = \sqrt{-2\ln S}\cos\Theta$, $X_2 = \sqrt{-2\ln S}\sin\Theta$. Using also the polar coordinates $X_1 = R'\cos\Theta'$, $X_2 = R'\sin\Theta'$, we find that $\Theta' = \Theta$ and $R' = \sqrt{-2\ln S}$. It is clear that R' and Θ' are independent, since R and Θ are independent inside the unit circle. Also, Θ' is uniformly distributed between 0 and 2π; and the probability that $R' \le r$ is the probability that $-2\ln S \le r^2$, i.e., the probability that $S \ge e^{-r^2/2}$. This equals $1 - e^{-r^2/2}$, since $S = R^2$ is uniformly distributed between zero and one. The probability that R' lies between r and $r + dr$ is therefore the derivative of $1 - e^{-r^2/2}$, namely, $re^{-r^2/2}\,dr$. Similarly, the probability that Θ' lies between θ and $\theta + d\theta$ is $(1/2\pi)\,d\theta$. The joint probability that $X_1 \le x_1$ and that $X_2 \le x_2$ now can be computed, it is

$$\begin{aligned}\int_{\{(r,\theta)\,|\,r\cos\theta\le x_1,\, r\sin\theta\le x_2\}} &\frac{1}{2\pi}e^{-r^2/2}\,r\,dr\,d\theta \\ &= \frac{1}{2\pi}\int_{\{(x,y)\,|\,x\le x_1,\, y\le x_2\}} e^{-(x^2+y^2)/2}\,dx\,dy \\ &= \left(\sqrt{\frac{1}{2\pi}}\int_{-\infty}^{x_1} e^{-x^2/2}\,dx\right)\left(\sqrt{\frac{1}{2\pi}}\int_{-\infty}^{x_2} e^{-y^2/2}\,dy\right).\end{aligned}$$

This proves that X_1 and X_2 are independent and normally distributed, as desired.

(*2*) *The rectangle-wedge-tail method,* introduced by G. Marsaglia. In this method we use the distribution

$$F(x) = \sqrt{\frac{2}{\pi}}\int_0^x e^{-t^2/2}\,dt, \qquad x \ge 0, \tag{12}$$

so that $F(x)$ gives the distribution of the *absolute value* of a normal deviate. After X has been computed according to distribution (12), we will attach a random sign to its value, and this will make it a true normal deviate.

The rectangle-wedge-tail approach is based on several important general techniques that we shall explore as we develop the algorithm. The first key idea is to regard $F(x)$ as a *mixture* of several other functions, namely to write

$$F(x) = p_1F_1(x) + p_2F_2(x) + \cdots + p_nF_n(x), \tag{13}$$

where F_1, F_2, ..., F_n are appropriate distributions and p_1, p_2, ..., p_n are nonnegative probabilities that sum to 1. If we generate a random variable X by choosing distribution F_j with probability p_j, it is easy to see that X will have distribution F overall. Some of the distributions $F_j(x)$ may be rather difficult to handle, even harder than F itself, but we can usually arrange things so that the

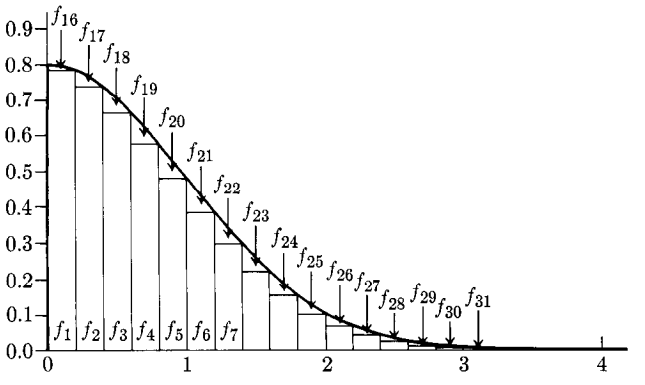


Fig. 9. The density function divided into 31 parts. The area of each part represents the average number of times a random number with that density is to be computed.

probability p_j is very small in this case. Most of the distributions $F_j(x)$ will be quite easy to accommodate, since they will be trivial modifications of the uniform distribution. The resulting method yields an extremely efficient program, since its *average* running time is very small.

It is easier to understand the method if we work with the *derivatives* of the distributions instead of the distributions themselves. Let

$$f(x) = F'(x), \qquad f_j(x) = F_j'(x);$$

these are called the *density* functions of the probability distributions. Equation (13) becomes

$$f(x) = p_1 f_1(x) + p_2 f_2(x) + \cdots + p_n f_n(x). \tag{14}$$

Each $f_j(x)$ is ≥ 0, and the total area under the graph of $f_j(x)$ is 1; so there is a convenient graphical way to display the relation (14): The area under $f(x)$ is divided into n parts, with the part corresponding to $f_j(x)$ having area p_j. See Fig. 9, which illustrates the situation in the case of interest to us here, with $f(x) = F'(x) = \sqrt{2/\pi}\, e^{-x^2/2}$; the area under this curve has been divided into $n = 31$ parts. There are 15 rectangles, which represent $p_1 f_1(x), \ldots, p_{15} f_{15}(x)$; there are 15 wedge-shaped pieces, which represent $p_{16} f_{16}(x), \ldots, p_{30} f_{30}(x)$; and the remaining part $p_{31} f_{31}(x)$ is the "tail," namely the entire graph of $f(x)$ for $x \geq 3$.

The rectangular parts $f_1(x), \ldots, f_{15}(x)$ represent *uniform distributions*. For example, $f_3(x)$ represents a random variable uniformly distributed between $\frac{2}{5}$ and $\frac{3}{5}$. The altitude of $p_j f_j(x)$ is $f(j/5)$, hence the area of the jth rectangle is

$$p_j = \tfrac{1}{5} f(j/5) = \sqrt{\frac{2}{25\pi}}\, e^{-j^2/50}, \qquad \text{for } 1 \leq j \leq 15. \tag{15}$$

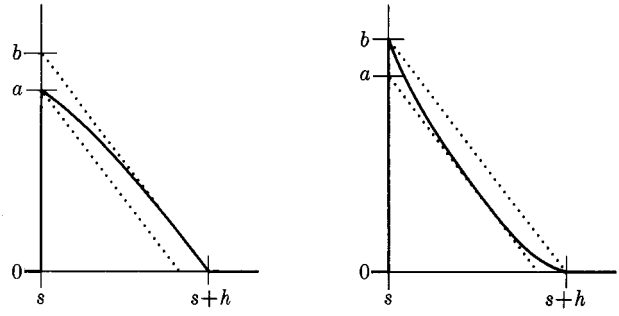


Fig. 10. Density functions for which Algorithm L may be used to generate random numbers.

In order to generate such rectangular portions of the distribution, we simply compute

$$X = \tfrac{1}{5}U + S, \tag{16}$$

where U is uniform and S takes the value $(j-1)/5$ with probability p_j. Since $p_1 + \cdots + p_{15} = .9183$, we can use simple uniform deviates like this about 92 percent of the time.

In the remaining 8 percent, we will usually have to generate one of the wedge-shaped distributions $F_{16}, \ldots, F_{30}$. Typical examples of what we need to do are shown in Fig. 10. When $x < 1$, the curved part is concave downward, and when $x > 1$ it is concave upward, but in each case the curved part is reasonably close to a straight line, and it can be enclosed in two parallel lines as shown.

To handle these wedge-shaped distributions, we will rely on yet another general technique, von Neumann's *rejection method* for obtaining a complicated density from another one that "encloses" it. The polar method described above is a simple example of such an approach: Steps P1–P3 obtain a random point inside the unit circle by first generating a random point in a larger square, rejecting it and starting over again if the point was outside the circle.

The general rejection method is even more powerful than this. To generate a random variable X with density f, let g be another probability density function such that

$$f(t) \le cg(t) \tag{17}$$

for all t, where c is a constant. Now generate X according to density g, and also generate an independent uniform deviate U. If $U \ge f(X)/cg(X)$, reject X and start again with another X and U. When the condition $U < f(X)/cg(X)$ finally occurs, the resulting X will have density f as desired. [*Proof:* $X \le x$ will occur with probability $p(x) = \int_{-\infty}^{x} \big(g(t)\,dt \cdot f(t)/cg(t)\big) + qp(x)$, where the quantity $q = \int_{-\infty}^{\infty} \big(g(t)\,dt \cdot (1 - f(t)/cg(t))\big) = 1 - 1/c$ is the probability of rejection; hence $p(x) = \int_{-\infty}^{x} f(t)\,dt$.]

The rejection technique is most efficient when c is small, since there will be c iterations on the average before a value is accepted. (See exercise 6.) In some

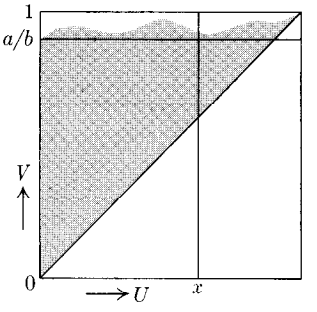


Fig. 11. Region of "acceptance" in Algorithm L.

cases $f(x)/cg(x)$ is always 0 or 1; then U need not be generated. In other cases if $f(x)/cg(x)$ is hard to compute, we may be able to "squeeze" it between two bounding functions

$$r(x) \le f(x)/cg(x) \le s(x) \tag{18}$$

that are much simpler, and the exact value of $f(x)/cg(x)$ need not be calculated unless $r(x) \le U < s(x)$. The following algorithm solves the wedge problem by developing the rejection method still further.

Algorithm L (*Nearly linear densities*). This algorithm may be used to generate a random variable X for any distribution whose density $f(x)$ satisfies the following conditions (cf. Fig. 10):

$$\begin{aligned} f(x) &= 0, && \text{for } x < s \text{ and for } x > s+h; \\ a - b(x-s)/h \le f(x) &\le b - b(x-s)/h, && \text{for } s \le x \le s+h. \end{aligned} \tag{19}$$

L1. [Get $U \le V$.] Generate two independent random variables U, V, uniformly distributed between zero and one. If $U > V$, exchange $U \leftrightarrow V$.

L2. [Easy case?] If $V \le a/b$, go to L4.

L3. [Try again?] If $V > U + (1/b)f(s+hU)$, go back to step L1. (If a/b is close to 1, this step of the algorithm will not be necessary very often.)

L4. [Compute X.] Set $X \leftarrow s + hU$. ∎

When step L4 is reached, the point (U, V) is a random point in the area shaded in Fig. 11, namely, $0 \le U \le V \le U + (1/b)f(s+hU)$. Conditions (19) ensure that

$$\frac{a}{b} \le U + \frac{1}{b} f(s+hU) \le 1.$$

Now the probability that $X \le s + hx$, for $0 \le x \le 1$, is the ratio of area to the left of the vertical line $U = x$ in Fig. 11 to the total area, namely,

$$\int_0^x \frac{1}{b} f(s+hu)\,du \Big/ \int_0^1 \frac{1}{b} f(s+hu)\,du = \int_s^{s+hx} f(v)\,dv;$$

therefore X has the correct distribution.

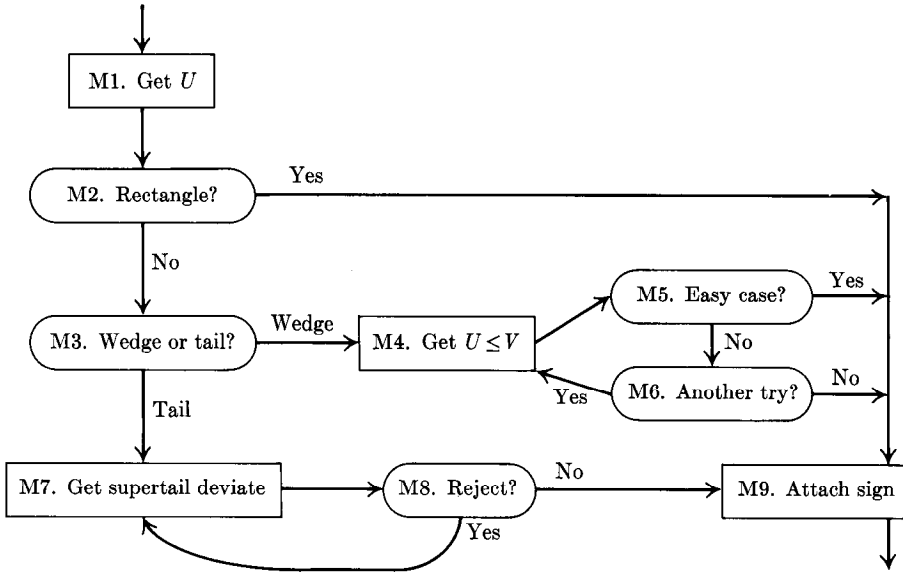


Fig. 12. The "rectangle-wedge-tail" algorithm for generating normal deviates.

With appropriate constants a_j, b_j, s_j, Algorithm L will take care of the wedge-shaped densities f_{j+15} of Fig. 9, for $1 \le j \le 15$. The final distribution, F_{31}, needs to be treated only about one time in 370; it is used whenever a result $X \ge 3$ is to be computed. Exercise 11 shows that a standard rejection scheme can be used for this "tail"; hence we are ready to consider the procedure in its entirety:

Algorithm M (*Rectangle-wedge-tail method for normal deviates*). This algorithm uses a number of auxiliary tables $(P_0, \ldots, P_{31})$, $(Q_1, \ldots, Q_{15})$, $(Y_0, \ldots, Y_{31})$, $(Z_0, \ldots, Z_{31})$, $(S_1, \ldots, S_{16})$, $(D_{16}, \ldots, D_{30})$, $(E_{16}, \ldots, E_{30})$, constructed as explained in exercise 10; examples appear in Table 1. We assume that a binary computer is being used; a similar procedure could be worked out for decimal machines.

M1. [Get U.] Generate a uniform random number $U = (.b_0 b_1 b_2 \ldots b_t)_2$. (Here the b's are the bits in the binary representation of U. For reasonable accuracy, t should be at least 24.) Set $\psi \leftarrow b_0$. (Later, ψ will be used to determine the sign of the result.)

M2. [Rectangle?] Set $j \leftarrow (b_1 b_2 b_3 b_4 b_5)_2$, a binary number determined by the leading bits of U, and set $f \leftarrow (.b_6 b_7 \ldots b_t)_2$, the fraction determined by the remaining bits. If $f \ge P_j$, set $X \leftarrow Y_j + fZ_j$ and go to M9. Otherwise if $j \le 15$ (i.e., $b_1 = 0$), set $X \leftarrow S_j + fQ_j$ and go to M9. (This is an adaptation of Walker's alias method (3).)

Table 1

EXAMPLE OF TABLES USED WITH ALGORITHM M*

j	P_j	P_{j+16}	Q_j	Y_j	Y_{j+16}	Z_j	Z_{j+16}	S_{j+1}	D_{j+15}	E_{j+15}
0	.000	.067		0.00	0.59	0.20	0.21	0.0		
1	.849	.161	.236	—0.92	0.96	1.32	0.24	0.2	.505	25.00
2	.970	.236	.206	—5.86	—0.06	6.66	0.26	0.4	.773	12.50
3	.855	.285	.234	—0.58	0.12	1.38	0.28	0.6	.876	8.33
4	.994	.308	.201	—33.13	1.31	34.93	0.29	0.8	.939	6.25
5	.995	.304	.201	—39.55	0.31	41.35	0.29	1.0	.986	5.00
6	.933	.280	.214	—2.57	1.12	2.97	0.28	1.2	.995	4.06
7	.923	.241	.217	—1.61	0.54	2.61	0.26	1.4	.987	3.37
8	.727	.197	.275	0.67	0.75	0.73	0.25	1.6	.979	2.86
9	1.000	.152	.200	0.00	0.56	0.00	0.24	1.8	.972	2.47
10	.691	.112	.289	0.35	0.17	0.65	0.23	2.0	.966	2.16
11	.454	.079	.440	—0.17	0.38	0.37	0.22	2.2	.960	1.92
12	.287	.052	.698	0.92	—0.01	0.28	0.21	2.4	.954	1.71
13	.174	.033	1.150	0.36	0.39	0.24	0.21	2.6	.948	1.54
14	.101	.020	1.974	—0.02	0.20	0.22	0.20	2.8	.942	1.40
15	.057	.086	3.526	0.19	0.78	0.21	0.22	3.0	.936	1.27

*In practice, this data would be given with much greater precision; the table shows only enough figures so that interested readers may test their own algorithms for computing the values more accurately.

M3. [Wedge or tail?] (Now $15 \le j \le 31$, and each particular value j occurs with probability p_j.) If $j = 31$, go to M7.

M4. [Get $U \le V$.] Generate two new uniform deviates, U, V; if $U > V$, exchange $U \leftrightarrow V$. (We are now performing Algorithm L.) Set $X \leftarrow S_{j-15} + \frac{1}{5}U$.

M5. [Easy case?] If $V \le D_j$, go to M9.

M6. [Another try?] If $V > U + E_j(e^{(S_{j-14}^2 - X^2)/2} - 1)$, go back to step M4; otherwise go to M9. (This step is executed with low probability.)

M7. [Get supertail deviate.] Generate two new independent uniform deviates, U, V, and set $X \leftarrow \sqrt{9 - 2\ln V}$.

M8. [Reject?] If $UX \ge 3$, go back to step M7. (This will occur only about one-twelfth as often as we reach step M8.)

M9. [Attach sign.] If $\psi = 1$, set $X \leftarrow -X$. ∎

This algorithm is a very pretty example of mathematical theory intimately interwoven with programming ingenuity—a fine illustration of the art of computer programming! Only steps M1, M2, and M9 need to be performed most of the time, and the other steps aren't terribly slow either. The first publications of the rectangle-wedge-tail method were by G. Marsaglia, *Annals Math. Stat.* **32** (1961), 894–899; G. Marsaglia, M. D. MacLaren, and T. A. Bray, *CACM* **7** (1964), 4–10. Further refinements of Algorithm M have been developed by G. Marsaglia, K. Ananthanarayanan, and N. J. Paul, *Inf. Proc. Letters* **5** (1976), 27–30.

(*3*) *The odd-even method,* due to G. E. Forsythe. An amazingly simple technique for generating random deviates with a density of the general exponential form

$$f(x) = Ce^{-h(x)}, \quad \text{for } a \le x < b, \qquad f(x) = 0 \quad \text{otherwise}, \tag{20}$$

when

$$0 \le h(x) \le 1 \qquad \text{for } a \le x < b, \tag{21}$$

was discovered by John von Neumann and G. E. Forsythe about 1950. The idea is based on the rejection method described earlier, letting $g(x)$ be the uniform distribution on $[a, b)$: We set $X \leftarrow a+(b-a)U$, where U is a uniform deviate, and then we want to accept X with probability $e^{-h(X)}$. The latter operation could be done by testing $e^{-h(X)}$ vs. V, or $h(X)$ vs. $-\ln V$, when V is another uniform deviate, but the job can be done without applying any transcendental functions in the following interesting way. Set $V_0 \leftarrow h(X)$, then generate uniform deviates V_1, V_2, ... until finding some $K \ge 1$ with $V_{K-1} < V_K$. For fixed X and k, the probability that $h(X) \ge V_1 \ge \cdots \ge V_k$ is $1/k!$ times the probability that $\max(V_1, \ldots, V_k) \le h(X)$, namely $h(X)^k/k!$; hence the probability that $K = k$ is $h(X)^{k-1}/(k-1)! - h(X)^k/k!$, and the probability that K is odd is

$$\sum_{k \text{ odd},\, k \ge 1} \left(\frac{h(X)^{k-1}}{(k-1)!} - \frac{h(X)^k}{k!} \right) = e^{-h(X)}. \tag{22}$$

Therefore we reject X and try again if K is even; we accept X as a random variable with density (20) if K is odd. Note that we usually won't have to generate many V's in order to determine K, since the average value of K (given X) is $\sum_{k \ge 0}$ probability that$(K > k) = \sum_{k \ge 0} h(X)^k/k! = e^{h(X)} \le e$.

Forsythe realized some years later that this approach leads to an efficient method for calculating normal deviates, without the need for any auxiliary routines to calculate square roots or logarithms as in Algorithms P and M. His procedure, with an improved choice of intervals $[a, b)$ due to J. H. Ahrens and U. Dieter, can be summarized as follows.

Algorithm F (*Odd-even method for normal deviates*). This algorithm generates normal deviates on a binary computer, assuming approximately $t + 1$ bits of accuracy. The algorithm requires a table of values $d_j = a_j - a_{j-1}$, for $1 \le j \le t + 1$, where a_j is defined by the relation

$$\sqrt{\frac{2}{\pi}} \int_{a_j}^{\infty} e^{-x^2/2}\, dx = \frac{1}{2^j}. \tag{23}$$

F1. [Get U.] Generate a uniform random number $U = (.b_0 b_1 \ldots b_t)_2$, where b_0, ..., b_t denote the bits in binary notation. Set $\psi \leftarrow b_0$, $j \leftarrow 1$, and $a \leftarrow 0$.

F2. [Find first zero b_j.] If $b_j = 1$, set $a \leftarrow a + d_j$, $j \leftarrow j + 1$, and *repeat* this step. (If $j = t + 1$, treat b_j as zero.)

F3. [Generate candidate.] (Now $a = a_{j-1}$, and the current value of j occurs with probability $\approx 2^{-j}$. We will generate X in the range $[a_{j-1}, a_j)$, using

the rejection method described above, with $h(x) = x^2/2 - a^2/2 = y^2/2 + ay$ where $y = x - a$. Exercise 12 proves that $h(x) \le 1$ as required in (21).) Set $Y \leftarrow d_j$ times $(.b_{j+1} \ldots b_t)_2$ and $V \leftarrow (\frac{1}{2}Y + a)Y$. (Since the average value of j is 2, there will usually be enough significant bits in $(.b_{j+1} \ldots b_t)_2$ to provide decent accuracy. The calculations are readily done in fixed point arithmetic.)

F4. [Reject?] Generate a uniform deviate U. If $V < U$, go on to step F5. Otherwise set V to a new uniform deviate; and if now $U < V$ (i.e., if K is even, in the discussion above), go back to F3, otherwise repeat step F4.

F5. [Return X.] Set $X \leftarrow a + Y$. If $\psi = 1$, set $X \leftarrow -X$. ∎

Values of d_j for $1 \le j \le 47$ appear in a paper by Ahrens and Dieter, *Math. Comp.* **27** (1973), 927–937; their paper discusses refinements of the algorithm that improve its speed at the expense of more tables. Algorithm F is attractive since it is almost as fast as Algorithm M and it is easier to implement. The average number of uniform deviates per normal deviate is 2.53947; R. P. Brent [*CACM* **17** (1974), 704–705] has shown how to reduce this number to 1.37446 at the expense of two subtractions and one division per uniform deviate saved.

(*4*) *Ratios of uniform deviates.* There is yet another good way to generate normal deviates, discovered by A. J. Kinderman and J. F. Monahan in 1976. Their idea is to generate a random point (U, V) in the region defined by

$$0 < u \le 1, \qquad -2u\sqrt{\ln(1/u)} \le v \le 2u\sqrt{\ln(1/u)}, \tag{24}$$

and then to output the ratio $X \leftarrow V/U$. The shaded area of Fig. 13 is the magic region (24) that makes this all work. Before we study the associated theory, let us first state the algorithm so that its efficiency and simplicity are manifest:

Algorithm R (*Ratio method for normal deviates*). This algorithm generates normal deviates X.

R1. [Get U, V.] Generate two independent uniform deviates U and V, where U is nonzero, and set $X \leftarrow \sqrt{8/e}(V - \frac{1}{2})/U$. (Now X is the ratio of the coordinates $(U, \sqrt{8/e}(V - \frac{1}{2}))$ of a random point in the rectangle that encloses the shaded region in Fig. 13. We will accept X if the corresponding point actually lies "in the shade," otherwise we will try again.)

R2. [Optional upper bound test.] If $X^2 \le 5 - 4e^{1/4}U$, output X and terminate the algorithm. (This step can be omitted if desired; it tests whether or not the selected point is in the interior region of Fig. 13, making it unnecessary to calculate a logarithm.)

R3. [Optional lower bound test.] If $X^2 \ge 4e^{-1.35}/U + 1.4$, go back to R1. (This step could also be omitted; it tests whether or not the selected point is outside the exterior region of Fig. 13, making it unnecessary to calculate a logarithm.)

R4. [Final test.] If $X^2 \le -4/\ln U$, output X and terminate the algorithm. Otherwise go back to R1. ∎

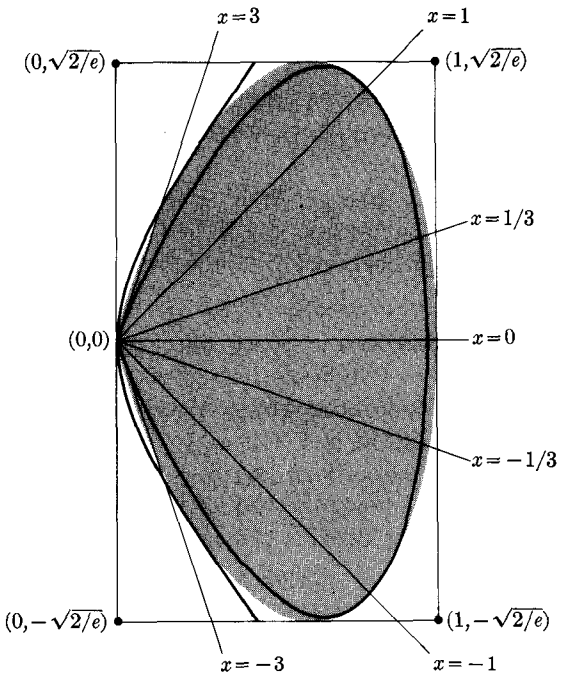


Fig. 13. Region of "acceptance" in the ratio - of - uniforms method for normal deviates. Lengths of lines with coordinate ratio x have the normal distribution.

Exercises 20 and 21 work out the timing analysis; four different algorithms are analyzed, since steps R2 and R3 can be included or omitted depending on one's preference. The following table shows how many times each step will be performed, on the average, depending on which of the optional tests is applied:

Step	Neither	R2 only	R3 only	Both
R1	1.369	1.369	1.369	1.369
R2	0	1.369	0	1.369
R3	0	0	1.369	0.467
R4	1.369	0.467	1.134	0.232

(25)

Thus it pays to omit the optional tests if there is a very fast logarithm operation, but if the log routine is rather slow it pays to include them.

But why does it work? One reason is that we can calculate the probability that $X \leq x$, and it turns out to be the correct value (10). But such a calculation isn't very easy unless one happens to hit on the right "trick," and anyway it is better to understand how the algorithm might have been discovered in the first place. Kinderman and Monahan derived it by working out the following theory that can be used with any well-behaved density function $f(x)$ [cf. *ACM Trans. Math. Software* **3** (1977), 257–260].

In general, suppose that a point (U, V) has been generated uniformly over the region of the (u, v)-plane defined by

$$u > 0, \qquad u^2 \leq g(v/u) \tag{26}$$

for some nonnegative integrable function g. If we set $X \leftarrow V/U$, the probability that $X \le x$ can be calculated by integrating $du\,dv$ over the region defined by the two relations in (26) plus the auxiliary condition $v/u \le x$, then dividing by the same integral without this extra condition. Letting $v = tu$ so that $dv = u\,dt$, the integral becomes

$$\int_{-\infty}^{x} dt \int_{0}^{\sqrt{g(t)}} u\,du = \frac{1}{2}\int_{-\infty}^{x} g(t)\,dt.$$

Hence the probability that $X \le x$ is

$$\int_{-\infty}^{x} g(t)\,dt \Big/ \int_{-\infty}^{+\infty} g(t)\,dt. \tag{27}$$

The normal distribution comes out when $g(t) = e^{-t^2/2}$; and the condition $u^2 \le g(v/u)$ simplifies in this case to $(v/u)^2 \le -4\ln u$. It is easy to see that the set of all (u, v) satisfying this relation is entirely contained in the rectangle of Fig. 13.

The bounds in steps R2 and R3 define interior and exterior regions with simpler boundary equations. The well-known inequality

$$e^x \ge 1 + x,$$

which holds for all real numbers x, can be used to show that

$$1 + \ln c - cu \;\le\; -\ln u \;\le\; 1/cu - 1 + \ln c \tag{28}$$

for any constant $c > 0$. Exercise 21 proves that $c = e^{1/4}$ is the best possible constant to use in step R2. The situation is more complicated in step R3, and there doesn't seem to be a simple expression for the optimum c in that case, but computational experiments show that the best value for R3 is approximately $e^{1.35}$. The approximating curves (28) are tangent to the true boundary when $u = 1/c$.

It is possible to obtain a faster method by partitioning the region into subregions, most of which can be handled more quickly. Of course, this means that auxiliary tables will be needed, as in Algorithms M and F.

(*5*) *Variations of the normal distribution.* So far we have considered the normal distribution with mean zero and standard deviation one. If X has this distribution, then

$$Y = \mu + \sigma X \tag{29}$$

has the normal distribution with mean μ and standard deviation σ. Furthermore, if X_1 and X_2 are independent normal deviates with mean zero and standard deviation one, and if

$$Y_1 = \mu_1 + \sigma_1 X_1, \qquad Y_2 = \mu_2 + \sigma_2(\rho X_1 + \sqrt{1-\rho^2}\,X_2), \tag{30}$$

then Y_1 and Y_2 are *dependent* random variables, normally distributed with means μ_1, μ_2 and standard deviations σ_1, σ_2, and with correlation coefficient ρ. (For a generalization to n variables, see exercise 13.)

D. The exponential distribution. After uniform deviates and normal deviates, the next most important random quantity is an *exponential deviate*. Such numbers occur in "arrival time" situations; for example, if a radioactive substance emits alpha particles at a rate so that one particle is emitted every μ seconds on the average, then the time between two successive emissions has the exponential distribution with mean μ. This distribution is defined by the formula

$$F(x) = 1 - e^{-x/\mu}, \qquad x \geq 0. \tag{31}$$

(*1*) *Logarithm method.* Clearly, if $y = F(x) = 1 - e^{-x/\mu}$, then $x = F^{-1}(y) = -\mu \ln(1-y)$. Therefore by Eq. (7), $-\mu \ln(1-U)$ has the exponential distribution. Since $1 - U$ is uniformly distributed when U is, we conclude that

$$X = -\mu \ln U \tag{32}$$

is exponentially distributed with mean μ. (The case $U = 0$ must be avoided.)

(*2*) *Random minimization method.* We saw in Algorithm F that there are simple and fast alternatives to calculating the logarithm of a uniform deviate. The following especially efficient approach has been developed by G. Marsaglia, M. Sibuya, and J. H. Ahrens.

Algorithm S (*Exponential distribution with mean μ*). This algorithm produces exponential deviates on a binary computer, using uniform deviates with t-bit accuracy. The constants

$$Q[k] = \frac{\ln 2}{1!} + \frac{(\ln 2)^2}{2!} + \cdots + \frac{(\ln 2)^k}{k!}, \qquad k \geq 1, \tag{33}$$

should be precomputed, extending until $Q[k] > 1 - 2^{1-t}$.

S1. [Get U and shift.] Generate a t-bit uniform random binary fraction $U = (.b_1 b_2 \ldots b_t)_2$; locate the first zero bit b_j, and shift off the leading j bits, setting $U \leftarrow (.b_{j+1} \ldots b_t)_2$. (As in Algorithm F, the average value of j is 2.)

S2. [Immediate acceptance?] If $U < \ln 2$, set $X \leftarrow \mu(j \ln 2 + U)$ and terminate the algorithm. (Note that $Q[1] = \ln 2$.)

S3. [Minimize.] Find the least $k \geq 2$ such that $U < Q[k]$. Generate k new uniform deviates $U_1, \ldots, U_k$ and set $V \leftarrow \min(U_1, \ldots, U_k)$.

S4. [Deliver the answer.] Set $X \leftarrow \mu(j + V) \ln 2$. ∎

Alternative ways to generate exponential deviates (e.g., a ratio of uniforms as in Algorithm R) might also be used.

E. Other continuous distributions. Let us now consider briefly how to handle some other distributions that arise reasonably often in practice.

(*1*) *The gamma distribution* of order $a > 0$ is defined by

$$F(x) = \frac{1}{\Gamma(a)} \int_0^x t^{a-1} e^{-t}\, dt, \qquad x \geq 0. \tag{34}$$

When $a = 1$, this is the exponential distribution with mean 1; when $a = \frac{1}{2}$, it is the distribution of $\frac{1}{2}Z^2$, where Z has the normal distribution (mean 0, variance 1). If X and Y are independent gamma-distributed random variables, of order a and b, respectively, then $X + Y$ has the gamma distribution of order $a + b$. Thus, for example, the sum of k independent exponential deviates with mean 1 has the gamma distribution of order k. If the logarithm method (32) is being used to generate these exponential deviates, we need compute only one logarithm: $X \leftarrow -\ln(U_1 \ldots U_k)$, where $U_1, \ldots, U_k$ are nonzero uniform deviates. This technique handles all integer orders a; to complete the picture, a suitable method for $0 < a < 1$ appears in exercise 16.

The simple logarithm method is much too slow when a is large, since it requires $\lfloor a \rfloor$ uniform deviates. For large a, the following algorithm due to J. H. Ahrens is reasonably efficient, and it is easy to write in terms of standard subroutines.

Algorithm A (*Gamma distribution of order* $a > 1$).

A1. [Generate candidate.] Set $Y \leftarrow \tan(\pi U)$, where U is a uniform deviate, and set $X \leftarrow \sqrt{2a-1}\,Y + a - 1$. (In place of $\tan(\pi U)$ we could use a polar method, e.g., V_2/V_1 in step P4 of Algorithm P.)

A2. [Accept?] If $X \leq 0$, return to A1. Otherwise generate a uniform deviate V, and return to A1 if $V > (1 + Y^2)\exp\big((a-1)\ln(X/(a-1)) - \sqrt{2a-1}\,Y\big)$. Otherwise accept X. ∎

The average number of times step A1 is performed is < 1.902 when $a \geq 3$. For further discussion, proof, and a slightly more complex method that is two to three times faster, see J. H. Ahrens and U. Dieter, *Computing* **12** (1974), 223–246.

There is also an attractive approach for large a based on the remarkable fact that gamma deviates are approximately equal to aX^3, where X is normally distributed with mean $1 - 1/(9a)$ and standard deviation $1/\sqrt{9a}$; see G. Marsaglia, *Computers and Math.* **3** (1977), 321–325.*

(*2*) *The beta distribution* with positive parameters a and b is defined by

$$F(x) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int_0^x t^{a-1}(1-t)^{b-1}\, dt, \qquad 0 \leq x \leq 1. \tag{35}$$

Let X_1 and X_2 be independent gamma deviates of order a and b, respectively, and set $X \leftarrow X_1/(X_1 + X_2)$. Another method, useful for small a and b, is to set

*Change "$+(3a-1)$" to "$-(3a-1)$" in Step 3 of the algorithm on page 323.

$Y_1 \leftarrow U_1^{1/a}$ and $Y_2 \leftarrow U_2^{1/b}$ repeatedly until $Y_1+Y_2 \le 1$; then $X \leftarrow Y_1/(Y_1+Y_2)$. [See M. D. Jöhnk, *Metrika* **8** (1964), 5–15.] Still another approach, if a and b are integers (not too large), is to set X to the bth largest of $a+b-1$ independent uniform deviates (cf. exercise 7 at the beginning of Chapter 5). See also the direct method described by R. C. H. Cheng, *CACM* **21** (1978), 317–322.

(*3*) *The chi-square distribution* with ν degrees of freedom (Eq. 3.3.1–22) is obtained by setting $X \leftarrow 2Y$, where Y is a random variable having the gamma distribution of order $\nu/2$.

(*4*) *The F-distribution* (variance-ratio distribution) with ν_1 and ν_2 degrees of freedom is defined by

$$F(x) = \frac{\nu_1^{\nu_1/2}\,\nu_2^{\nu_2/2}\,\Gamma\big((\nu_1+\nu_2)/2\big)}{\Gamma(\nu_1/2)\Gamma(\nu_2/2)} \int_0^x t^{\nu_1/2-1}(\nu_2+\nu_1 t)^{-\nu_1/2-\nu_2/2}\,dt, \tag{36}$$

where $x \ge 0$. Let Y_1 and Y_2 be independent, having the chi-square distribution with ν_1 and ν_2 degrees of freedom, respectively; set $X \leftarrow Y_1\nu_2/Y_2\nu_1$. Or set $X \leftarrow \nu_2 Y/\nu_1(1-Y)$, where Y is a beta variate with parameters $\nu_1/2$, $\nu_2/2$.

(*5*) *The t-distribution* with ν degrees of freedom is defined by

$$F(x) = \frac{\Gamma\big((\nu+1)/2\big)}{\sqrt{\pi\nu}\,\Gamma(\nu/2)} \int_{-\infty}^x (1+t^2/\nu)^{-(\nu+1)/2}\,dt. \tag{37}$$

Let Y_1 be a normal deviate (mean 0, variance 1) and let Y_2 be independent of Y_1, having the chi-square distribution with ν degrees of freedom; set $X \leftarrow Y_1/\sqrt{Y_2/\nu}$. Alternatively, when $\nu > 2$, let Y_1 be a normal deviate and let Y_2 independently have the exponential distribution with mean $2/(\nu-2)$; set $Z \leftarrow Y_1^2/(\nu-2)$ and reject (Y_1, Y_2) if $e^{-Y_2-Z} \ge 1-Z$, otherwise set $X \leftarrow Y_1/\sqrt{(1-2\nu)(1-z)}$. The latter method is due to George Marsaglia, *Math. Comp.* **34** (1980), 235–236. [See also A. J. Kinderman, J. F. Monahan, and J. G. Ramage, *Math. Comp.* **31** (1977), 1009–1018.]

(*6*) *Random point on n-dimensional sphere with radius one.* Let $X_1, X_2, \ldots, X_n$ be independent normal deviates (mean 0, variance 1); the desired point on the unit sphere is

$$(X_1/r, X_2/r, \ldots, X_n/r), \qquad \text{where } r = \sqrt{X_1^2+X_2^2+\cdots+X_n^2}. \tag{38}$$

Note that if the X's are calculated using the polar method, Algorithm P, we compute two independent X's each time, and $X_1^2+X_2^2 = -2\ln S$ (in the notation of that algorithm); this saves a little of the time needed to evaluate r. The validity of this method comes from the fact that the distribution function for the point $(X_1, \ldots, X_n)$ has a density that depends only on its distance from the origin, so when it is projected onto the unit sphere it has the uniform distribution. This method was first suggested by G. W. Brown, in *Modern Mathematics for the Engineer*, First series, ed. by E. F. Beckenbach (New York: McGraw-Hill,

1956), p. 302. To get a random point *inside* the n-sphere, R. P. Brent suggests taking a point on the surface and multiplying it by $U^{1/n}$.

In three dimensions a significantly simpler method can be used, since each individual coordinate is uniformly distributed between -1 and 1: Find V_1, V_2, and S by steps P1–P3 of Algorithm P; then the desired random point on the surface of a globe is $(\alpha V_1, \alpha V_2, 2S - 1)$, where $\alpha = 2\sqrt{1 - S}$. [Robert E. Knop, *CACM* **13** (1970), 326.]

F. Important integer-valued distributions. A probability distribution that consists only of integer values can essentially be handled by the techniques described at the beginning of this section; but some of these distributions are so important in practice, they deserve special mention here.

(*1*) *The geometric distribution.* If some event occurs with probability p, the number N of independent trials needed until the event first occurs (or between occurrences of the event) has the geometric distribution. We have $N = 1$ with probability p, $N = 2$ with probability $(1 - p)p$, ..., $N = n$ with probability $(1 - p)^{n-1}p$. This is essentially the situation we have already considered in the gap test of Section 3.3.2; it is also directly related to the number of times certain loops in the algorithms of this section are executed, e.g., steps P1–P3 of the polar method.

A convenient way to generate a variable with this distribution is to set

$$N \leftarrow \lceil \ln U \,/\, \ln(1 - p) \rceil. \tag{39}$$

To check this formula, we observe that $\lceil \ln U \,/\, \ln(1 - p) \rceil = n$ if and only if $n - 1 < \ln U \,/\, \ln(1 - p) \leq n$, that is, $(1 - p)^{n-1} > U \geq (1 - p)^n$, and this happens with probability $p(1 - p)^{n-1}$ as required. Note that $\ln U$ can be replaced by $-Y$, where Y has the exponential distribution with mean 1.

The special case $p = \frac{1}{2}$ can be handled more easily on a binary computer, since formula (39) becomes $N \leftarrow \lceil -\log_2 U \rceil$; that is, N is one more than the number of leading zero bits in the binary representation of U.

(*2*) *The binomial distribution* (t, p). If some event occurs with probability p, and if we carry out t independent trials, the total number N of occurrences equals n with probability $\binom{t}{n} p^n (1 - p)^{t-n}$. (See Section 1.2.10.) In other words if we generate $U_1, \ldots, U_t$, we want to count how many of these are $< p$. For small t we can obtain N in exactly this way.

For large t, we can generate a beta variate X with integer parameters a and b where $a + b - 1 = t$; this effectively gives us the bth largest of t elements, without bothering to generate the other elements. Now if $X \geq p$, we set $N \leftarrow N_1$ where N_1 has the binomial distribution $(a - 1, p/X)$, since this tells us how many of $a - 1$ random numbers in the range $[0, X)$ are $< p$; and if $X < p$, we set $N \leftarrow a + N_1$ where N_1 has the binomial distribution $(b - 1, (p - X)/(1 - X))$, since N_1 tells us how many of $b - 1$ random numbers in the range $[X, 1)$ are $< p$. By choosing $a = 1 + \lfloor t/2 \rfloor$, the parameter t will be reduced to a reasonable size after about $\lg t$ reductions of this kind. (This approach is due to J. H. Ahrens, who has also suggested an alternative for medium-sized t; see exercise 27.)

(*3*) *The Poisson distribution* with mean μ. This distribution is related to the exponential distribution as the binomial distribution is related to the geometric: It represents the number of occurrences, per unit time, of an event that can occur at any instant of time. For example, the number of alpha particles emitted by a radioactive substance in a single second has a Poisson distribution.

According to this principle, we can produce a Poisson deviate N by generating independent exponential deviates X_1, X_2, ... with mean $1/\mu$, stopping as soon as $X_1 + \cdots + X_m \geq 1$; then $N \leftarrow m - 1$. The probability that $X_1 + \cdots + X_m \geq 1$ is the probability that a gamma deviate of order m is $\geq \mu$, and this comes to $\int_\mu^\infty t^{m-1}e^{-t}\,dt/(m-1)!$; hence the probability that $N = n$ is

$$\frac{1}{n!}\int_\mu^\infty t^n e^{-t}\,dt \;-\; \frac{1}{(n-1)!}\int_\mu^\infty t^{n-1}e^{-t}\,dt = e^{-\mu}\frac{\mu^n}{n!}, \qquad n \geq 0. \tag{40}$$

If we generate exponential deviates by the logarithm method, the above recipe tells us to stop when $-(\ln U_1 + \cdots + \ln U_m)/\mu \geq 1$. Simplifying this expression, we see that the desired Poisson deviate can be obtained by calculating $e^{-\mu}$, converting it to a fixed point representation, then generating one or more uniform deviates U_1, U_2, ... until the product satisfies $U_1 \ldots U_m \leq e^{-\mu}$, finally setting $N \leftarrow m - 1$. On the average this requires the generation of $\mu + 1$ uniform deviates, so it is a very useful approach when μ is not too large.

When μ is large, we can obtain a method of order $\log \mu$ by using the fact that we know how to handle the gamma and binomial distributions for large orders: First generate X with the gamma distribution of order $m = \lfloor \alpha\mu \rfloor$, where α is a suitable constant. (Since X is equivalent to $-\ln(U_1 \ldots U_m)$, we are essentially bypassing m steps of the previous method.) If $X < \mu$, set $N \leftarrow m + N_1$, where N_1 is a Poisson deviate with mean $\mu - X$; and if $X \geq \mu$, set $N \leftarrow N_1$, where N_1 has the binomial distribution $(m-1, \mu/X)$. This method is due to J. H. Ahrens and U. Dieter, whose experiments suggest that $\frac{7}{8}$ is a good choice for α.

The validity of the above reduction when $X \geq \mu$ is a consequence of the following important principle: "Let X_1, ..., X_m be independent exponential deviates with the same mean; let $S_j = X_1 + \cdots + X_j$ and let $V_j = S_j/S_m$ for $1 \leq j \leq m$. Then the distribution of V_1, V_2, ..., V_{m-1} is the same as the distribution of $m - 1$ independent uniform deviates sorted into increasing order." To establish this principle formally, we compute the probability that $V_1 \leq v_1$, ..., $V_{m-1} \leq v_{m-1}$, given the value of $S_m = s$, for arbitrary values $0 \leq v_1 \leq \cdots \leq v_{m-1} \leq 1$: Let $f(v_1, v_2, \ldots, v_{m-1})$ be the $(m-1)$-fold integral

$$\int_0^{v_1 s} \mu e^{-t_1/\mu}\,dt_1 \int_0^{v_2 s - t_1} \mu e^{-t_2/\mu}\,dt_2 \ldots \times \int_0^{v_{m-1}s - t_1 - \cdots - t_{m-2}} \mu e^{-t_{m-1}/\mu}\,dt_{m-1} \cdot \mu e^{-(s - t_1 - \cdots - t_{m-1})/\mu};$$

then

$$\frac{f(v_1, v_2, \ldots, v_{m-1})}{f(1, 1, \ldots, 1)} = \frac{\int_0^{v_1} du_1 \int_{u_1}^{v_2} du_2 \ldots \int_{u_{m-2}}^{v_{m-1}} du_{m-1}}{\int_0^1 du_1 \int_{u_1}^1 du_2 \ldots \int_{u_{m-2}}^1 du_{m-1}},$$

by making the substitution $t_1 = su_1$, $t_1 + t_2 = su_2$, ..., $t_1 + \cdots + t_{m-1} = su_{m-1}$. The latter ratio is the corresponding probability that uniform deviates $U_1, \ldots, U_{m-1}$ satisfy $U_1 \le v_1, \ldots, U_{m-1} \le v_{m-1}$, given that they also satisfy $U_1 \le \cdots \le U_{m-1}$.

A more efficient but somewhat more complicated technique for binomial and Poisson deviates is sketched in exercise 22.

G. For further reading. The forthcoming book *Non-Uniform Random Numbers* by J. H. Ahrens and U. Dieter discusses many more algorithms for the generation of random variables with nonuniform distributions, together with a careful consideration of the efficiency of each technique on typical computers.

From a theoretical point of view it is interesting to consider *optimal* methods for generating random variables with a given distribution, in the sense that the method produces the desired result from the minimum possible number of random bits. For the beginnings of a theory dealing with such questions, see D. E. Knuth and A. C. Yao, *Algorithms and Complexity*, ed. by J. F. Traub (New York: Academic Press, 1976), 357–428.

Exercise 16 is recommended as a review of many of the techniques in this section.

EXERCISES

1. [*10*] If α and β are real numbers with $\alpha < \beta$, how would you generate a random real number uniformly distributed between α and β?

2. [*M16*] Assuming that mU is a random integer between 0 and $m - 1$, what is the *exact* probability that $\lfloor kU \rfloor = r$, if $0 \le r < k$? Compare this with the desired probability $1/k$.

▶ **3.** [*14*] Discuss treating U as an integer and *dividing* by k to get a random integer between 0 and $k - 1$, instead of multiplying as suggested in the text. Thus (1) would be changed to

```
ENTA 0
LDX  U
DIV  K
```

with the result appearing in register X. Is this a good method?

4. [*M20*] Prove the two relations in (8).

▶ **5.** [*21*] Suggest an efficient method to compute a random variable with the distribution $px + qx^2 + rx^3$, where $p \ge 0$, $q \ge 0$, $r \ge 0$, and $p + q + r = 1$.

6. [*HM21*] A quantity X is computed by the following method:

"*Step* 1. Generate two independent uniform deviates U, V.

Step 2. If $U^2 + V^2 \ge 1$, return to step 1; otherwise set $X \leftarrow U$."

What is the distribution function of X? How many times will step 1 be performed? (Give the mean and standard deviation.)

▶ **7.** [*20*] (A. J. Walker.) Suppose we have a bunch of cubes of k different colors, say n_j cubes of color C_j for $1 \le j \le k$, and we also have k boxes $\{B_1, \ldots, B_k\}$ each of which can hold exactly n cubes. Furthermore $n_1 + \cdots + n_k = kn$, so the cubes will just fit in the boxes. Prove (constructively) that there is always a way to put the cubes into the boxes so that each box contains at most two different colors of cubes; in fact, there is a way to do it so that, whenever box B_j contains two colors, one of those colors is C_j. Show how to use this principle to compute the P and Y tables required in (3), given a probability distribution $(p_1, \ldots, p_k)$.

8. [*M15*] Show that operation (3) could be changed to

$$\text{if} \quad U < P_K \quad \text{then} \quad X \leftarrow x_{K+1} \quad \text{otherwise} \quad X \leftarrow Y_K$$

(i.e., using the original value of U instead of V) if this were more convenient, by suitably modifying P_0, P_1, ..., P_{k-1}.

9. [*HM10*] Why is the curve $f(x)$ of Fig. 9 concave downward for $x < 1$, concave upward for $x > 1$?

▶ **10.** [*HM24*] Explain how to calculate auxiliary constants P_j, Q_j, Y_j, Z_j, S_j, D_j, E_j so that Algorithm M delivers answers with the correct distribution.

▶ **11.** [*HM27*] Prove that steps M7–M8 of Algorithm M generate a random variable with the appropriate tail of the normal distribution; i.e., the probability that $X \le x$ should be

$$\int_3^x e^{-t^2/2}\,dt \Big/ \int_3^\infty e^{-t^2/2}\,dt, \qquad x \ge 3.$$

[*Hint:* Show that it is a special case of the rejection method, with $g(t) = Cte^{-t^2/2}$ for some C.]

12. [*HM23*] (R. P. Brent.) Prove that the numbers a_j defined in (23) satisfy the relation $a_j^2/2 - a_{j-1}^2/2 < \ln 2$ for all $j \ge 1$. [*Hint:* If $f(x) = e^{x^2/2} \int_x^\infty e^{-t^2/2}\,dt$, show that $f(x) < f(y)$ for $0 \le x < y$.]

13. [*HM25*] Given a set of n independent normal deviates, X_1, X_2, ..., X_n, with mean 0 and variance 1, show how to find constants b_j and a_{ij}, $1 \le j \le i \le n$, so that if

$$Y_1 = b_1 + a_{11}X_1, \qquad Y_2 = b_2 + a_{21}X_1 + a_{22}X_2, \qquad \ldots,$$
$$Y_n = b_n + a_{n1}X_1 + a_{n2}X_2 + \cdots + a_{nn}X_n,$$

then Y_1, Y_2, ..., Y_n are dependent normally distributed variables, Y_j has mean μ_j, and the Y's have a given covariance matrix (c_{ij}). (The covariance, c_{ij}, of Y_i and Y_j is defined to be the average value of $(Y_i - \mu_i)(Y_j - \mu_j)$. In particular, c_{jj} is the variance of Y_j, the square of its standard deviation. Not all matrices (c_{ij}) can be covariance matrices, and your construction is, of course, only supposed to work whenever a solution to the given conditions is possible.)

14. [*M21*] If X is a random variable with continuous distribution $F(x)$, and if c is a constant, what is the distribution of cX?

15. [*HM21*] If X_1 and X_2 are independent random variables with the respective distributions $F_1(x)$ and $F_2(x)$, and with densities $f_1(x) = F_1'(x)$, $f_2(x) = F_2'(x)$, what are the distribution and density functions of the quantity $X_1 + X_2$?

▶ **16.** [*HM22*] (J. H. Ahrens.) Develop an algorithm for gamma deviates of order a when $0 < a \leq 1$, using the rejection method with $c\,g(t) = t^{a-1}/\Gamma(a)$ for $0 < t < 1$, $e^{-t}/\Gamma(a)$ for $t \geq 1$.

▶ **17.** [*M24*] What is the *distribution function* $F(x)$ for the geometric distribution with probability p? What is the *generating function* $G(z)$? What are the mean and standard deviation of this distribution?

18. [*M24*] Suggest a method to compute a random integer N for which N takes the value n with probability $np^2(1-p)^{n-1}$, $n \geq 0$. (The case of particular interest is when p is rather small.)

19. [*22*] The *negative binomial distribution* (t, p) has integer values $N = n$ with probability $\binom{t-1+n}{n}p^t(1-p)^n$. (Unlike the ordinary binomial distribution, t need not be an integer, since this quantity is nonnegative for all n whenever $t > 0$.) Generalizing exercise 18, explain how to generate integers N with this distribution when t is a small positive integer. What method would you suggest if $t = p = \frac{1}{2}$?

20. [*M20*] Let A be the area of the shaded region in Fig. 13, and let R be the area of the enclosing rectangle. Let I be the area of the interior region recognized by step R2, and let E be the area between the exterior region rejected in step R3 and the outer rectangle. Determine the number of times each step of Algorithm R is performed, for each of its four variants as in (25), in terms of A, R, I, and E.

21. [*HM29*] Derive formulas for the quantities A, R, I, and E defined in exercise 20. (For I and especially E you may wish to use an interactive computer algebra system.) Show that $c = e^{1/4}$ is the best possible constant in step R2 for tests of the form "$X^2 \leq 4(1 + \ln c) - 4cU$."

22. [*HM40*] Can the exact Poisson distribution for large μ be obtained by generating an appropriate normal deviate, converting it to an integer in some convenient way, and applying a (possibly complicated) correction a small percent of the time?

23. [*HM23*] (J. von Neumann.) Are the following two ways to generate a random quantity X equivalent (i.e., does the quantity X have the same distribution)?

Method 1: Set $X \leftarrow \sin((\pi/2)U)$, where U is uniform.

Method 2: Generate two uniform deviates, U, V, and if $U^2 + V^2 \geq 1$, repeat until $U^2 + V^2 < 1$. Then set $X \leftarrow |U^2 - V^2|/(U^2 + V^2)$.

24. [*HM40*] (S. Ulam, J. von Neumann.) Let V_0 be a randomly selected real number between 0 and 1, and define the sequence $\langle V_n \rangle$ by the rule $V_{n+1} = 4V_n(1 - V_n)$. If this computation is done with perfect accuracy, the result should be a sequence with the distribution $\sin^2 \pi U$, where U is uniform, i.e., with distribution function $F(x) = \int_0^x dx/\sqrt{2\pi x(1-x)}$. For if we write $V_n = \sin^2 \pi U_n$, we find that $U_{n+1} = (2U_n) \bmod 1$; and by the fact that almost all real numbers have a random binary expansion (see Section 3.5), this sequence U_n is equidistributed. But if the computation of V_n is done with only finite accuracy, the above argument breaks down because we soon are dealing with noise from the roundoff error. [*Reference:* von Neumann's *Collected Works* **5**, 768–770.]

Analyze the sequence $\langle V_n \rangle$ defined above when only finite accuracy is present, both empirically (for various different choices of V_0) and theoretically. Does the sequence have a distribution resembling the expected distribution?

25. [*M25*] Let X_1, X_2, ..., X_5 be binary words each of whose bits is independently 0 or 1 with probability $\frac{1}{2}$. What is the probability that a given bit position of $X_1 \vee (X_2 \wedge (X_3 \vee (X_4 \wedge X_5)))$ contains a 1? Generalize.

26. [*M18*] Let N_1 and N_2 be independent Poisson deviates with respective means μ_1 and μ_2, where $\mu_1 > \mu_2 \geq 0$. Prove or disprove: (a) $N_1 + N_2$ has the Poisson distribution with mean $\mu_1 + \mu_2$. (b) $N_1 - N_2$ has the Poisson distribution with mean $\mu_1 - \mu_2$.

27. [*22*] (J. H. Ahrens.) On most binary computers there is an efficient way to count the number of 1's in a binary word (cf. Section 7.1). Hence there is a nice way to obtain the binomial distribution (t, p) when $p = \frac{1}{2}$, simply by generating t random bits and counting the number of 1's.

Design an algorithm that produces the binomial distribution (t, p) for *arbitrary* p, using only a subroutine for the special case $p = \frac{1}{2}$ as a source of random data. [*Hint:* Simulate a process that first looks at the most significant bits of t uniform deviates, then at the second bit of those deviates whose leading bit is not sufficient to determine whether or not their value is $< p$, etc.]

28. [*HM35*] (R. P. Brent.) Develop a method to generate a random point on the surface of the ellipsoid defined by $\sum a_k x_k^2 = 1$, where $a_1 \geq \cdots \geq a_n > 0$.

29. [*M20*] (J. L. Bentley and J. D. Saxe.) Fint a simple way to generate n numbers $X_1, \ldots, X_n$ that are uniform between 0 and 1 except for the fact that they are sorted: $X_1 \leq \cdots \leq X_n$. Your algorithm should take only $O(n)$ steps.

3.4.2. Random Sampling and Shuffling

Many data processing applications call for an unbiased choice of n records at random from a file containing N records. This problem arises, for example, in quality control or other statistical calculations where sampling is needed. Usually N is very large, so that it is impossible to contain all the data in memory at once; and the individual records themselves are often very large, so that we can't even hold n records in memory. Therefore we seek an efficient procedure for selecting n records by deciding either to accept or to reject each record as it comes along, writing the accepted records onto an output file.

Several methods have been devised for this problem. The most obvious approach is to select each record with probability n/N; this may sometimes be appropriate, but it gives only an *average* of n records in the sample. The standard deviation is $\sqrt{n(1-n/N)}$, and it is possible that the sample will be either too large for the desired application, or too small to give the necessary results.

A simple modification of the "obvious" procedure gives what we want: The $(t+1)$st record should be selected with probability $(n-m)/(N-t)$, if m items have already been selected. This is the appropriate probability, since of all the possible ways to choose n things from N such that m values occur in the first t, exactly

$$\binom{N-t-1}{n-m-1} \Big/ \binom{N-t}{n-m} = \frac{n-m}{N-t} \tag{1}$$

of these select the $(t+1)$st element.

The idea developed in the preceding paragraph leads immediately to the following algorithm:

Algorithm S (*Selection sampling technique*). To select n records at random from a set of N, where $0 < n \leq N$.

S1. [Initialize.] Set $t \leftarrow 0$, $m \leftarrow 0$. (During this algorithm, m represents the number of records selected so far, and t is the total number of input records we have dealt with.)

S2. [Generate U.] Generate a random number U, uniformly distributed between zero and one.

S3. [Test.] If $(N - t)U \geq n - m$, go to step S5.

S4. [Select.] Select the next record for the sample, and increase m and t by 1. If $m < n$, go to step S2; otherwise the sample is complete and the algorithm terminates.

S5. [Skip.] Skip the next record (do not include it in the sample), increase t by 1, and go to step S2. ∎

This algorithm may appear to be unreliable at first glance and, in fact, to be incorrect; but a careful analysis (see the exercises below) shows that it is completely trustworthy. It is not difficult to verify that

a) At most N records are input (we never run off the end of the file before choosing n items).

b) The sample is completely unbiased; in particular, the probability that any given element is selected, e.g., the last element of the file, is n/N.

Statement (b) is true in spite of the fact that we are *not* selecting the $(t+1)$st item with probability n/N, we select it with the probability in Eq. (1)! This has caused some confusion in the published literature. Can the reader explain this seeming contradiction?

(*Note:* When using Algorithm S, one should be careful to use a different source of random numbers U each time the program is run, to avoid connections between the samples obtained on different days. This can be done, for example, by choosing a different value of X_0 for the linear congruential method each time; X_0 could be set to the current date, or to the last X value generated on the previous run of the program.)

We will usually not have to pass over all N records; in fact, since (b) above says that the last record is selected with probability n/N, we will terminate the algorithm *before* considering the last record exactly $(1 - n/N)$ of the time. The average number of records considered when $n = 2$ is about $\frac{2}{3}N$, and the general formulas are given in exercises 5 and 6.

Algorithm S and a number of other sampling techniques are discussed in a paper by C. T. Fan, Mervin E. Muller, and Ivan Rezucha, *J. Amer. Stat. Assoc.* **57** (1962), 387–402. The method was independently discovered by T. G. Jones, *CACM* **5** (1962), 343.

A problem arises if we don't know the value of N in advance, since the precise value of N is crucial in Algorithm S. Suppose we want to select n items at random from a file, without knowing exactly how many are present in that file. We could first go through and count the records, then take a second pass to select them; but it is generally better to sample $m \geq n$ of the original items on the first pass, where m is much less than N, so that only m items must be considered on the second pass. The trick, of course, is to do this in such a way that the final result is a truly random sample of the original file.

Since we don't know when the input is going to end, we must keep track of a random sample of the input records seen so far, thus always being prepared for the end. As we read the input we will construct a "reservoir" that contains only those m records that have appeared among the previous samples. The first n records always go into the reservoir. When the $(t+1)$st record is being input, for $t \geq n$, we will have in memory a table of n indices pointing to those records in the reservoir that belong to the random sample we have chosen from the first t records. The problem is to maintain this situation with t increased by one, namely to find a new random sample from among the $t+1$ records now known to be present. It is not hard to see that we should include the new record in the new sample with probability $n/(t+1)$, and in such a case it should replace a random element of the previous sample.

Thus, the following procedure does the job:

Algorithm R (*Reservoir sampling*). To select n records at random from a file of unknown size $\geq n$, given $n > 0$. An auxiliary file called the "reservoir" contains all records that are candidates for the final sample. The algorithm uses a table of distinct indices $I[j]$ for $1 \leq j \leq n$, each of which points to one of the records in the reservoir.

R1. [Initialize.] Input the first n records and copy them to the reservoir. Set $I[j] \leftarrow j$ for $1 \leq j \leq n$, and set $t \leftarrow m \leftarrow n$. (If the file being sampled has fewer than n records, it will of course be necessary to abort the algorithm and report failure. During this algorithm, $\{I[1], \ldots, I[n]\}$ point to the records in the current sample, m is the size of the reservoir, and t is the number of input records dealt with so far.)

R2. [End of file?] If there are no more records to be input, go to step R6.

R3. [Generate and test.] Increase t by 1, then generate a random integer M between 1 and t (inclusive). If $M > n$, go to R5.

R4. [Add to reservoir.] Copy the next record of the input file to the reservoir, increase m by 1, and set $I[M] \leftarrow m$. (The record previously pointed to by $I[M]$ is being replaced in the sample by the new record.) Go back to R2.

R5. [Skip.] Skip over the next record of the input file (do not include it in the reservoir), and return to step R2.

R6. [Second pass.] Sort the I table entries so that $I[1] < \cdots < I[n]$; then go through the reservoir, copying the records with these indices into the output file that is to hold the final sample. ∎

Algorithm R is due to Alan G. Waterman. The reader may wish to work out the example of its operation that appears in exercise 9.

If the records are sufficiently short, it is of course unnecessary to have a reservoir at all; we can keep the n records of the current sample in memory at all times (see exercise 10).

The natural question to ask about Algorithm R is, "What is the expected size of the reservoir?" Exercise 11 shows that the average value of m is exactly $n(1 + H_N - H_n)$; this is approximately $n(1 + \ln(N/n))$. So if $N/n = 1000$, the reservoir will contain only about $\frac{1}{125}$ as many items as the original file.

Note that Algorithms S and R can be used to obtain samples for several independent categories simultaneously. For example, if we have a large file of names and addresses of U.S. residents, we could pick random samples of exactly 10 people from each of the 50 states without making 50 passes through the file, and without first sorting the file by state.

The sampling problem can be regarded as the computation of a random *combination*, according to the conventional definition of combinations of N things taken n at a time (see Section 1.2.6). Now let us consider the problem of computing a random *permutation* of t objects; we will call this the *shuffling problem*, since shuffling a deck of cards is nothing more than subjecting it to a random permutation.

A moment's reflection is enough to convince oneself that the approaches people traditionally use to shuffle cards are miserably inadequate; there is no hope of obtaining each of the $t!$ permutations with anywhere near equal probability by such methods. It has been said that expert bridge players make use of this fact when deciding whether or not to "finesse."

If t is small, we can obtain a random permutation very quickly by generating a random integer between 1 and $t!$. For example, when $t = 4$, a random number between 1 and 24 suffices to select a random permutation from a table of all possibilities. But for large t, it is necessary to be more careful if we want to claim that each permutation is equally likely, since $t!$ is much larger than the accuracy of individual random numbers.

A suitable shuffling procedure can be obtained by recalling Algorithm 3.3.2P, which gives a simple correspondence between each of the $t!$ possible permutations and a sequence of numbers $(c_1, c_2, \ldots, c_{t-1})$, with $0 \le c_j \le j$. It is easy to compute such a set of numbers at random, and we can use the correspondence to produce a random permutation.

Algorithm P (*Shuffling*). Let $X_1, X_2, \ldots, X_t$ be a set of t numbers to be shuffled.

P1. [Initialize.] Set $j \leftarrow t$.

P2. [Generate U.] Generate a random number U, uniformly distributed between zero and one.

P3. [Exchange.] Set $k \leftarrow \lfloor jU \rfloor + 1$. (Now k is a random integer, between 1 and j.) Exchange $X_k \leftrightarrow X_j$.

P4. [Decrease j.] Decrease j by 1. If $j > 1$, return to step P2. ∎

This algorithm was first published by L. E. Moses and R. V. Oakford, in *Tables of Random Permutations* (Stanford University Press, 1963); and by R. Durstenfeld, *CACM* **7** (1964), 420. It can also be modified to obtain a random permutation of a random combination (see exercise 14).

For a discussion of random combinatorial objects of other kinds (e.g., partitions), see Section 7.2 and/or the book *Combinatorial Algorithms* by Nijenhuis and Wilf (New York: Academic Press, 1975).

EXERCISES

1. [*M12*] Explain Eq. (1).

2. [*20*] Prove that Algorithm S never tries to read more than N records of its input file.

▶ **3.** [*22*] The $(t+1)$st item in Algorithm S is selected with probability $(n-m)/(N-t)$, not n/N, yet the text claims that the sample is unbiased—so each item should be selected with the *same* probability. How can both of these statements be true?

4. [*M23*] Let $p(m,t)$ be the probability that exactly m items are selected from among the first t in the selection sampling technique. Show directly from Algorithm S that

$$p(m,t) = \binom{t}{m}\binom{N-t}{n-m}\Big/\binom{N}{n}, \qquad \text{for } 0 \le t \le N.$$

5. [*M24*] What is the average value of t when Algorithm S terminates? (In other words, how many of the N records have been passed, on the average, before the sample is complete?)

6. [*M24*] What is the standard deviation of the value computed in the previous exercise?

▶ **7.** [*M25*] Prove that any *given* choice of n records from the set of N is obtained by Algorithm S with probability $1/\binom{N}{n}$. Therefore the sample is completely unbiased.

8. [*M46*] Algorithm S computes one uniform deviate for each input record it handles. Find a more efficient way to determine the number of input records to skip before the first is selected, assuming that N/n is rather large. (We could iterate this process to select the remaining $n-1$ records, thus reducing the number of necessary random deviates from order N to order n.)

9. [*12*] Let $n = 3$. If Algorithm R is applied to a file containing 20 records numbered 1 thru 20, and if the random numbers generated in step R3 are respectively

$$4, 1, 6, 7, 5, 3, 5, 11, 11, 3, 7, 9, 3, 11, 4, 5, 4,$$

which records go into the reservoir? Which are in the final sample?

10. [*15*] Modify Algorithm R so that the reservoir is eliminated, assuming that the n records of the current sample can be held in memory.

▶ **11.** [*M25*] Let p_m be the probability that exactly m elements are put into the reservoir during the first pass of Algorithm R. Determine the generating function $G(z) = \sum_m p_m z^m$, and find the mean and standard deviation. (Use the ideas of Section 1.2.10.)

12. [*M26*] The gist of Algorithm P is that any permutation π can be uniquely written as a product of transpositions in the form $\pi = (a_t t)\dots(a_3 3)(a_2 2)$, where $1 \le a_j \le j$ for $t \ge j > 1$. Prove that there is also a unique representation of the form $\pi = (b_2 2)(b_3 3)\dots(b_t t)$, where $1 \le b_j \le j$ for $1 < j \le t$, and design an algorithm that computes the b's from the a's in $O(t)$ steps.

13. [*M23*] (S. W. Golomb.) One of the most common ways to shuffle cards is to divide the deck into two parts as equal as possible, and to "riffle" them together. (See the discussion of card-playing etiquette in Hoyle's rules of card games; we read, "A shuffle of this sort should be made about three times to mix the cards thoroughly.") Consider a deck of $2n - 1$ cards $X_1, X_2, \dots, X_{2n-1}$; a "perfect shuffle" s divides this deck into $X_1, X_2, \dots, X_n$ and $X_{n+1}, \dots, X_{2n-1}$, then perfectly interleaves them to obtain $X_1, X_{n+1}, X_2, X_{n+2}, \dots, X_{2n-1}, X_n$. The "cut" operation c^j changes $X_1, X_2, \dots, X_{2n-1}$ into $X_{j+1}, \dots, X_{2n-1}, X_1, \dots, X_j$. Show that by combining perfect shuffles and cuts, at most $(2n - 1)(2n - 2)$ different arrangements of the deck are possible, if $n > 1$.

▶ **14.** [*30*] (Ole-Johan Dahl.) If $X_k = k$ for $1 \le k \le t$ at the start of Algorithm P, and if we terminate the algorithm when j reaches the value $t - n$, the sequence $X_{t-n+1}, \dots, X_t$ is a random permutation of a random combination of n elements. Show how to simulate the effect of this procedure using only $O(n)$ cells of memory.

▶ **15.** [*M25*] Devise a way to compute a random sample of n records from N, given N and n, based on the idea of hashing (Section 6.4). Your method should use $O(n)$ storage locations and an average of $O(n)$ units of time, and it should present the sample as a sorted set of integers $1 \le X_1 < X_2 < \cdots < X_n \le N$.

16. [*25*] Discuss the problem of *weighted* sampling, where each subset of n elements is obtained with probability proportional to the product of the weights of the elements.

3.5. WHAT IS A RANDOM SEQUENCE?

A. Introductory remarks. We have seen in this chapter how to generate sequences

$$\langle U_n \rangle = U_0, U_1, U_2, \ldots \tag{1}$$

of real numbers in the range $0 \le U_n < 1$, and we have called them "random" sequences even though they are completely deterministic in character. To justify this terminology, we claimed that the numbers "behave as if they are truly random." Such a statement may be satisfactory for practical purposes (at the present time), but it sidesteps a very important philosophical and theoretical question: Precisely what do we mean by "random behavior"? A quantitative definition is needed. It is undesirable to talk about concepts that we do not really understand, especially since many apparently paradoxical statements can be made about random numbers.

The mathematical theory of probability and statistics carefully sidesteps the question; it refrains from making absolute statements, and instead expresses everything in terms of how much *probability* is to be attached to statements involving random sequences of events. The axioms of probability theory are set up so that abstract probabilities can be computed readily, but nothing is said about what probability really signifies, or how this concept can be applied meaningfully to the actual world. In the book *Probability, Statistics, and Truth* (New York: Macmillan, 1957), R. von Mises discusses this situation in detail, and presents the view that a proper definition of probability depends on obtaining a proper definition of a random sequence.

Let us paraphrase here some statements made by two of the many authors who have commented on the subject.

> *D. H. Lehmer* (1951): "A random sequence is a vague notion embodying the idea of a sequence in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests, traditional with statisticians and depending somewhat on the uses to which the sequence is to be put."
>
> *J. N. Franklin* (1962): "The sequence (1) is random if it has every property that is shared by all infinite sequences of independent samples of random variables from the uniform distribution."

Franklin's statement essentially generalizes Lehmer's to say that the sequence must satisfy *all* statistical tests. His definition is not completely precise, and we will see later that a reasonable interpretation of his statement leads us to conclude that there is no such thing as a random sequence! So let us begin with Lehmer's less restrictive statement and attempt to make *it* precise. What we really want is a relatively short list of mathematical properties, each of which is satisfied by our intuitive notion of a random sequence; furthermore, the list is to be complete enough so that we are willing to agree that any sequence satisfying these properties is "random." In this section, we will develop what seems to be

an adequate definition of randomness according to these criteria, although many interesting questions remain to be answered.

Let u and v be real numbers, $0 \le u < v \le 1$. If U is a random variable that is uniformly distributed between 0 and 1, the probability that $u \le U < v$ is equal to $v - u$. For example, the probability that $\frac{1}{3} \le U < \frac{2}{3}$ is $\frac{1}{3}$. How can we translate this property of the single number U into a property of the infinite sequence $U_0, U_1, U_2, \ldots$? The obvious answer is to count how many times U_n lies between u and v, and the average number of times should equal $v - u$. Our intuitive idea of probability is based in this way on the frequency of occurrence.

More precisely, let $\nu(n)$ be the number of values of j, $0 \le j < n$, such that $u \le U_j < v$; we want the ratio $\nu(n)/n$ to approach the value $v - u$ as n approaches infinity:

$$\lim_{n\to\infty} \nu(n)/n = v - u. \tag{2}$$

If this condition holds for all choices of u and v, the sequence is said to be *equidistributed.*

Let $S(n)$ be a statement about the integer n and the sequence $U_1, U_2, \ldots$; for example, $S(n)$ might be the statement considered above, "$u \le U_n < v$." We can generalize the idea used in the preceding paragraph to define "the probability that $S(n)$ is true" with respect to a particular infinite sequence: Let $\nu(n)$ be the number of values of j, $0 \le j < n$, such that $S(j)$ is true.

Definition A. *We say* $\Pr(S(n)) = \lambda$, *if* $\lim_{n\to\infty} \nu(n)/n = \lambda$. (Read, "The probability that $S(n)$ is true equals λ, if the limit as n tends to infinity of $\nu(n)/n$ equals λ.")

In terms of this notation, the sequence $U_0, U_1, \ldots$ is equidistributed if and only if $\Pr(u \le U_n < v) = v - u$, for all real numbers u, v with $0 \le u < v \le 1$.

A sequence may be equidistributed without being random. For example, if $U_0, U_1, \ldots$ and $V_0, V_1, \ldots$ are equidistributed sequences, it is not hard to show that the sequence

$$W_0, W_1, W_2, W_3, \ldots = \tfrac{1}{2}U_0,\ \tfrac{1}{2}+\tfrac{1}{2}V_0,\ \tfrac{1}{2}U_1,\ \tfrac{1}{2}+\tfrac{1}{2}V_1,\ \ldots \tag{3}$$

is also equidistributed, since the subsequence $\frac{1}{2}U_0, \frac{1}{2}U_1, \ldots$ is equidistributed between 0 and $\frac{1}{2}$, while the alternate terms $\frac{1}{2} + \frac{1}{2}V_0, \frac{1}{2} + \frac{1}{2}V_1, \ldots$, are equidistributed between $\frac{1}{2}$ and 1. In the sequence of W's, a value less than $\frac{1}{2}$ is always followed by a value greater than or equal to $\frac{1}{2}$, and conversely; hence the sequence is not random by any reasonable definition. A stronger property than equidistribution is needed.

A natural generalization of the equidistribution property, which removes the objection stated in the preceding paragraph, is to consider adjacent pairs of numbers of our sequence. We can require the sequence to satisfy the condition

$$\Pr(u_1 \le U_n < v_1 \quad \text{and} \quad u_2 \le U_{n+1} < v_2) = (v_1 - u_1)(v_2 - u_2) \tag{4}$$

for any four numbers u_1, v_1, u_2, v_2 with $0 \le u_1 < v_1 \le 1$, $0 \le u_2 < v_2 \le 1$. In

general, for any positive integer k we can require our sequence to be *k-distributed* in the following sense:

Definition B. *The sequence* (1) *is said to be k-distributed if*

$$\Pr(u_1 \le U_n < v_1, \ldots, u_k \le U_{n+k-1} < v_k) = (v_1 - u_1)\ldots(v_k - u_k) \tag{5}$$

for all choices of real numbers u_j, v_j, *with* $0 \le u_j < v_j \le 1$, *for* $1 \le j \le k$.

An equidistributed sequence is a 1-distributed sequence. Note that if $k > 1$, a k-distributed sequence is always $(k-1)$-distributed, since we may set $u_k = 0$ and $v_k = 1$ in Eq. (5). Thus, in particular, any sequence that is known to be 4-distributed must also be 3-distributed, 2-distributed, and equidistributed. We can investigate the largest k for which a given sequence is k-distributed; and this leads us to formulate

Definition C. *A sequence is said to be* ∞*-distributed if it is k-distributed for all positive integers k.*

So far we have considered "$[0,1)$ sequences," i.e., sequences of real numbers lying between zero and one. The same ideas apply to integer-valued sequences; let us say a sequence $\langle X_n \rangle = X_0, X_1, X_2, \ldots$ is a "b-ary sequence" if each X_n is one of the integers $0, 1, \ldots, b-1$. Thus, a 2-ary (binary) sequence is a sequence of zeros and ones.

We also say that a k-digit "b-ary number" is a string of k integers $x_1 x_2 \ldots x_k$, where $0 \le x_j < b$ for $1 \le j \le k$.

Definition D. *A b-ary sequence is said to be k-distributed if*

$$\Pr(X_n X_{n+1} \ldots X_{n+k-1} = x_1 x_2 \ldots x_k) = 1/b^k \tag{6}$$

for all b-ary numbers $x_1 x_2 \ldots x_k$.

It is clear from this definition that if $U_0, U_1, \ldots$ is a k-distributed $[0,1)$ sequence, then the sequence $\lfloor bU_0 \rfloor, \lfloor bU_1 \rfloor, \ldots$ is a k-distributed b-ary sequence. (If we set $u_j = x_j/b$, $v_j = (x_j + 1)/b$, $X_n = \lfloor bU_n \rfloor$, Eq. (5) becomes Eq. (6).) Furthermore, every k-distributed b-ary sequence is also $(k-1)$-distributed, if $k > 1$: we add together the probabilities for the b-ary numbers $x_1 \ldots x_{k-1}\,0$, $x_1 \ldots x_{k-1}\,1, \ldots, x_1 \ldots x_{k-1}\,(b-1)$ to obtain

$$\Pr(X_n \ldots X_{n+k-2} = x_1 \ldots x_{k-1}) = 1/b^{k-1}.$$

(Probabilities for disjoint events are additive; see exercise 5.) It therefore is natural to speak of an ∞-distributed b-ary sequence, as in Definition C above.

The representation of a positive real number in the radix-b number system may be regarded as a b-ary sequence; for example, π corresponds to the 10-ary

sequence 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, ... It has been conjectured that this sequence is ∞-distributed, but nobody has yet been able to prove that it is even 1-distributed.

Let us analyze these concepts a little more closely in the case when k equals a million. A binary sequence that is 1000000-distributed is going to have runs of a million zeros in a row! Similarly, a $[0,1)$ sequence that is 1000000-distributed is going to have runs of a million consecutive values each of which is less than $\frac{1}{2}$. It is true that this will happen only $(\frac{1}{2})^{1000000}$ of the time, on the average, but the fact is that it *does* happen. Indeed, this phenomenon will occur in any truly random sequence, using our intuitive notion of "truly random." One can easily imagine that such a situation will have a drastic effect if this set of a million "truly random" numbers is being used in a computer-simulation experiment; there would be good reason to complain about the random number generator. However, if we have a sequence of numbers that never has runs of a million consecutive U's less than $\frac{1}{2}$, the sequence is not random, and it will not be a suitable source of numbers for other conceivable applications that use extremely long blocks of U's as input. In summary, *a truly random sequence will exhibit local nonrandomness.* Local nonrandomness is necessary in some applications, but it is disastrous in others. We are forced to conclude that no sequence of "random" numbers can be adequate for every application.

In a similar vein, one may argue that there is no way to judge whether a *finite* sequence is random or not; any particular sequence is just as likely as any other one. These facts are definitely stumbling blocks if we are ever to have a useful definition of randomness, but they are not really cause for alarm. It is still possible to give a definition for the randomness of infinite sequences of real numbers in such a way that the corresponding theory (viewed properly) will give us a great deal of insight concerning the ordinary finite sequences of rational numbers that are actually generated on a computer. Furthermore, we shall see later in this section that there are several plausible definitions of randomness for finite sequences.

B. ∞-distributed sequences. Let us now undertake a brief study of the theory of sequences that are ∞-distributed. To describe the theory adequately, we will need to use a bit of higher mathematics, so we assume in the remainder of this subsection that the reader knows the material ordinarily taught in an "advanced calculus" course.

First it is convenient to generalize Definition A, since the limit appearing there does not exist for all sequences. Let us define

$$\overline{\Pr}(S(n)) = \limsup_{n\to\infty} (\nu(n)/n), \qquad \underline{\Pr}(S(n)) = \liminf_{n\to\infty} (\nu(n)/n). \tag{7}$$

Then $\Pr(S(n))$, if it exists, is the common value of $\underline{\Pr}(S(n))$ and $\overline{\Pr}(S(n))$.

We have seen that a k-distributed $[0,1)$ sequence leads to a k-distributed b-ary sequence, if U is replaced by $\lfloor bU \rfloor$. Our first theorem shows that a converse result is also true.

Theorem A. *Let $\langle U_n\rangle = U_0, U_1, U_2, \ldots$ be a $[0,1)$ sequence. If the sequence*

$$\langle \lfloor b_j U_n \rfloor \rangle = \lfloor b_j U_0 \rfloor, \lfloor b_j U_1 \rfloor, \lfloor b_j U_2 \rfloor, \ldots$$

is a k-distributed b_j-ary sequence for all b_j in an infinite sequence of integers $1 < b_1 < b_2 < b_3 < \cdots$, then the original sequence $\langle U_n\rangle$ is k-distributed.

As an example of this theorem, suppose that $b_j = 2^j$. The sequence $\lfloor 2^j U_0 \rfloor, \lfloor 2^j U_1 \rfloor, \ldots$ is essentially the sequence of the first j bits of the binary representations of $U_0, U_1, \ldots$. If all these integer sequences are k-distributed, in the sense of Definition D, then the real-valued sequence $U_0, U_1, \ldots$ must also be k-distributed in the sense of Definition B.

Proof of Theorem A. If the sequence $\lfloor bU_0 \rfloor, \lfloor bU_1 \rfloor, \ldots$ is k-distributed, it follows by the addition of probabilities that Eq. (5) holds whenever each u_j and v_j is a rational number with denominator b. Now let u_j, v_j be any real numbers, and let u'_j, v'_j be rational numbers with denominator b such that

$$u'_j \le u_j < u'_j + 1/b, \qquad v'_j \le v_j < v'_j + 1/b.$$

Let $S(n)$ be the statement that $u_1 \le U_n < v_1, \ldots, u_k \le U_{n+k-1} < v_k$. We have

$$\begin{aligned}\overline{\Pr}(S(n)) &\le \Pr\left(u'_1 \le U_n < v'_1 + \frac{1}{b}, \ldots, u'_k \le U_{n+k-1} < v'_k + \frac{1}{b}\right)\\ &= \left(v'_1 - u'_1 + \frac{1}{b}\right)\ldots\left(v'_k - u'_k + \frac{1}{b}\right);\\ \underline{\Pr}(S(n)) &\ge \Pr\left(u'_1 + \frac{1}{b} \le U_n < v'_1, \ldots, u'_k + \frac{1}{b} \le U_{n+k-1} < v'_k\right)\\ &= \left(v'_1 - u'_1 - \frac{1}{b}\right)\ldots\left(v'_k - u'_k - \frac{1}{b}\right).\end{aligned}$$

Now $|(v'_j - u'_j \pm 1/b) - (v_j - u_j)| \le 2/b$; since our inequalities hold for all $b = b_j$, and since $b_j \to \infty$ as $j \to \infty$, we have

$$(v_1 - u_1)\ldots(v_k - u_k) \le \underline{\Pr}(S(n)) \le \overline{\Pr}(S(n)) \le (v_1 - u_1)\ldots(v_k - u_k). \quad \blacksquare$$

The next theorem is our main tool for proving things about k-distributed sequences.

Theorem B. *Let $\langle U_n\rangle$ be a k-distributed $[0,1)$ sequence, and let $f(x_1, x_2, \ldots, x_k)$ be a Riemann-integrable function of k variables; then*

$$\begin{aligned}\lim_{n\to\infty} \frac{1}{n} \sum_{0\le j<n} & f(U_j, U_{j+1}, \ldots, U_{j+k-1})\\ &= \int_0^1 \cdots \int_0^1 f(x_1, x_2, \ldots, x_k)\, dx_1 \ldots dx_k.\end{aligned} \tag{8}$$

Proof. The definition of a k-distributed sequence states that this result is true in the special case that

$$f(x_1,\ldots,x_k)=\begin{cases}1, & \text{if } u_1\le x_1<v_1,\ldots,u_k\le x_k<v_k;\\ 0, & \text{otherwise.}\end{cases} \tag{9}$$

Therefore Eq. (8) is true whenever $f=a_1f_1+a_2f_2+\cdots+a_mf_m$ and when each f_j is a function of type (9); in other words, Eq. (8) holds whenever f is a "step-function" obtained by (i) partitioning the unit k-dimensional cube into subcells whose faces are parallel to the coordinate axes, and (ii) assigning a constant value to f on each subcell.

Now let f be any Riemann-integrable function. If ϵ is any positive number, we know (by the definition of Riemann-integrability) that there exist step functions $\underline{f}$ and $\overline{f}$ such that $\underline{f}(x_1,\ldots,x_k)\le f(x_1,\ldots,x_k)\le\overline{f}(x_1,\ldots,x_k)$, and such that the difference of the integrals of $\underline{f}$ and $\overline{f}$ is less than ϵ. Since Eq. (8) holds for $\underline{f}$ and $\overline{f}$, and since

$$\begin{aligned}\frac{1}{n}\sum_{0\le j<n}\underline{f}(U_j,\ldots,U_{j+k-1})&\le\frac{1}{n}\sum_{0\le j<n}f(U_j,\ldots,U_{j+k-1})\\&\le\frac{1}{n}\sum_{0\le j<n}\overline{f}(U_j,\ldots,U_{j+k-1}),\end{aligned}$$

we conclude that Eq. (8) is true also for f. ∎

Theorem B can be applied, for example, to the *permutation test* of Section 3.3.2. Let $(p_1,p_2,\ldots,p_k)$ be any permutation of the numbers $\{1,2,\ldots,k\}$; we want to show that

$$\Pr(U_{n+p_1-1}<U_{n+p_2-1}<\cdots<U_{n+p_k-1})=1/k!. \tag{10}$$

To prove this, assume that the sequence $\langle U_n\rangle$ is k-distributed, and let

$$f(x_1,\ldots,x_k)=\begin{cases}1, & \text{if } x_{p_1}<x_{p_2}<\cdots<x_{p_k};\\ 0, & \text{otherwise.}\end{cases}$$

We have

$$\begin{aligned}&\Pr(U_{n+p_1-1}<U_{n+p_2-1}<\cdots<U_{n+p_k-1})\\&\quad=\int_0^1\cdots\int_0^1 f(x_1,\ldots,x_k)\,dx_1\ldots dx_k\\&\quad=\int_0^1 dx_{p_k}\int_0^{x_{p_k}}\cdots\int_0^{x_{p_3}}dx_{p_2}\int_0^{x_{p_2}}dx_{p_1}=\frac{1}{k!}.\end{aligned}$$

Corollary P. *If a $[0,1)$ sequence is k-distributed, it satisfies the permutation test of order k, in the sense of Eq.* (10). ∎

We can also show that the *serial correlation test* is satisfied:

Corollary S. *If a* $[0,1)$ *sequence is* $(k+1)$*-distributed, the serial correlation coefficient between* U_n *and* U_{n+k} *tends to zero:*

$$\lim_{n\to\infty} \frac{\frac{1}{n}\sum U_jU_{j+k} - (\frac{1}{n}\sum U_j)(\frac{1}{n}\sum U_{j+k})}{\sqrt{(\frac{1}{n}\sum U_j^2 - (\frac{1}{n}\sum U_j)^2)(\frac{1}{n}\sum U_{j+k}^2 - (\frac{1}{n}\sum U_{j+k})^2)}} = 0.$$

(All summations here are for $0 \le j < n$.)

Proof. By Theorem B, the quantities

$$\tfrac{1}{n}\sum U_jU_{j+k}, \qquad \tfrac{1}{n}\sum U_j^2, \qquad \tfrac{1}{n}\sum U_{j+k}^2, \qquad \tfrac{1}{n}\sum U_j, \qquad \tfrac{1}{n}\sum U_{j+k}$$

tend to the respective limits $\frac{1}{4}, \frac{1}{3}, \frac{1}{3}, \frac{1}{2}, \frac{1}{2}$ as $n \to \infty$. ∎

Let us now consider some slightly more general distribution properties of sequences. We have defined the notion of k-distribution by considering all of the adjacent k-tuples; for example, a sequence is 2-distributed if and only if the points

$$(U_0,U_1),\ (U_1,U_2),\ (U_2,U_3),\ (U_3,U_4),\ (U_4,U_5),\ \ldots$$

are equidistributed in the unit square. It is quite possible, however, that this can happen while alternate pairs of points (U_1,U_2), (U_3,U_4), (U_5,U_6), ... are *not* equidistributed; if the density of points (U_{2n-1},U_{2n}) is deficient in some area, the other points (U_{2n},U_{2n+1}) might compensate. For example, the periodic binary sequence

$$\langle X_n\rangle = 0,0,0,1,\ 0,0,0,1,\ 1,1,0,1,\ 1,1,0,1,\ 0,0,0,1,\ \ldots, \tag{11}$$

with a period of length 16, is seen to be 3-distributed; yet the sequence of even-numbered elements $\langle X_{2n}\rangle = 0, 0, 0, 0, 1, 0, 1, 0, \ldots$ has three times as many zeros as ones, while the subsequence of odd-numbered elements $\langle X_{2n+1}\rangle = 0, 1, 0, 1, 1, 1, 1, 1, \ldots$ has three times as many ones as zeros.

If a sequence $\langle U_n\rangle$ is ∞-distributed, example (11) shows that it is not at all obvious that the subsequence of alternate terms $\langle U_{2n}\rangle = U_0, U_2, U_4, U_6, \ldots$ will be ∞-distributed or even 1-distributed. But we shall see that $\langle U_{2n}\rangle$ is, in fact, ∞-distributed, and much more is true.

Definition E. *A* $[0,1)$ *sequence* $\langle U_n\rangle$ *is said to be* (m,k)*-distributed if*

$$\Pr(u_1 \le U_{mn+j} < v_1,\ u_2 \le U_{mn+j+1} < v_2,\ \ldots,\ u_k \le U_{mn+j+k-1} < v_k) = (v_1-u_1)\ldots(v_k-u_k)$$

for all choices of real numbers u_r, v_r *with* $0 \le u_r < v_r \le 1$ *for* $1 \le r \le k$, *and for all integers* j *with* $0 \le j < m$.

Thus a k-distributed sequence is the special case $m = 1$ in Definition E; the case $m = 2$ means that the k-tuples starting in even positions must have the same density as the k-tuples starting in odd positions, etc.

Several properties of Definition E are obvious:

An (m, k)-distributed sequence is (m, κ)-distributed for $1 \le \kappa \le k$. (12)

An (m, k)-distributed sequence is (d, k)-distributed for all divisors d of m. (13)

We can also define the concept of an (m, k)-distributed b-ary sequence, as in Definition D; and the proof of Theorem A remains valid for (m, k)-distributed sequences.

The next theorem, which is in many ways rather surprising, shows that the property of being ∞-distributed is very strong indeed, much stronger than we imagined it to be when we first considered the definition of the concept.

Theorem C (Ivan Niven and H. S. Zuckerman). *An ∞-distributed sequence is (m, k)-distributed for all positive integers m and k.*

Proof. It suffices to prove the theorem for b-ary sequences, by using the generalization of Theorem A just mentioned. Furthermore, we may assume that $m = k$, because (12) and (13) tell us that the sequence will be (m, k)-distributed if it is (mk, mk)-distributed.

So we will prove that *any ∞-distributed b-ary sequence $X_0, X_1, \ldots$ is (m, m)-distributed for all positive integers m.* Our proof is a simplified version of the original one given by Niven and Zuckerman in *Pacific J. Math.* **1** (1951), 103–109.

The key idea we shall use is an important technique that applies to many situations in mathematics: "If the sum of m quantities and the sum of their squares are both consistent with the hypothesis that the m quantities are equal, then that hypothesis is true." In a strong form, this principle may be stated as follows:

Lemma E. *Given m sequences of numbers $\langle y_{jn} \rangle = y_{j0}, y_{j1}, \ldots$ for $1 \le j \le m$, suppose that*

$$\begin{aligned} \lim_{n\to\infty} (y_{1n} + y_{2n} + \cdots + y_{mn}) &= m\alpha, \\ \limsup_{n\to\infty} (y_{1n}^2 + y_{2n}^2 + \cdots + y_{mn}^2) &\le m\alpha^2. \end{aligned} \tag{14}$$

Then for each j, $\lim_{n\to\infty} y_{jn}$ exists and equals α.

An incredibly simple proof of this lemma is given in exercise 9. ∎

Resuming our proof of Theorem C, let $x = x_1 x_2 \ldots x_m$ be a b-ary number, and say that x *occurs* at position p if $X_{p-m+1} X_{p-m+2} \ldots X_p = x$. Let $\nu_j(n)$ be the number of occurrences of x at position p when $p < n$ and $p \bmod m = j$. Let $y_{jn} = \nu_j(n)/n$; we wish to prove that

$$\lim_{n\to\infty} y_{jn} = 1/mb^m. \tag{15}$$

First we know that

$$\lim_{n\to\infty}(y_{0n}+y_{1n}+\cdots+y_{(m-1)n})=1/b^m, \tag{16}$$

since the sequence is m-distributed. By Lemma E and Eq. (16), the theorem will be proved if we can show that

$$\limsup_{n\to\infty}(y_{0n}^2+y_{1n}^2+\cdots+y_{(m-1)n}^2)\le 1/mb^{2m}. \tag{17}$$

This inequality is not obvious yet; some rather delicate maneuvering is necessary before we can prove it. Let q be a multiple of m, and consider

$$C(n)=\sum_{0\le j<m}\binom{\nu_j(n)-\nu_j(n-q)}{2}. \tag{18}$$

This is the number of pairs of occurrences of x in positions p_1,p_2 with $n-q\le p_1<p_2<n$ and with p_2-p_1 a multiple of m. Consider now the sum

$$S_N=\sum_{1\le n\le N+q}C(n). \tag{19}$$

Each pair of occurrences of x in positions p_1,p_2 with $p_1<p_2<p_1+q$, where p_2-p_1 is a multiple of m and $p_1\le N$, is counted p_1+q-p_2 times in the total S_N (namely, when $p_2<n\le p_1+q$); and the pairs of such occurrences with $N<p_1<p_2<N+q$ are counted $N+q-p_2$ times.

Let $d_t(n)$ be the number of pairs of occurrences of x in positions p_1,p_2 with $p_1+t=p_2<n$. The analysis above shows that

$$\sum_{0<t<q/m}(q-mt)d_{mt}(N+q)\ge S_N\ge\sum_{0<t<q/m}(q-mt)d_{mt}(N). \tag{20}$$

Since the original sequence is q-distributed,

$$\lim_{N\to\infty}\frac{1}{N}d_{mt}(N)=1/b^{2m} \tag{21}$$

for all t, $0<t<q/m$, and therefore by (20) we have

$$\lim_{N\to\infty}\frac{S_N}{N}=\sum_{0<t<q/m}(q-mt)/b^{2m}=q(q-m)/2mb^{2m}. \tag{22}$$

This fact will prove the theorem, after some manipulation.

By definition,

$$2S_N = \sum_{1\le n\le N+q} \sum_{0\le j<m} \big((\nu_j(n) - \nu_j(n-q))^2 - (\nu_j(n) - \nu_j(n-q))\big),$$

and we can remove the unsquared terms by applying (16) to get

$$\lim_{N\to\infty} \frac{T_N}{N} = q(q-m)/mb^{2m} + q/b^m, \tag{23}$$

where

$$T_N = \sum_{1\le n\le N+q} \sum_{0\le j<m} \big(\nu_j(n) - \nu_j(n-q)\big)^2.$$

Using the inequality

$$\frac{1}{r}\Big(\sum_{1\le j\le r} a_j\Big)^2 \le \sum_{1\le j\le r} a_j^2$$

(cf. exercise 1.2.3–30), we find that

$$\limsup_{N\to\infty} \sum_{0\le j<m} \frac{1}{N(N+q)}\Big(\sum_{1\le n\le N+q} \big(\nu_j(n) - \nu_j(n-q)\big)\Big)^2 \le q(q-m)/mb^{2m} + q/b^m. \tag{24}$$

We also have

$$q\,\nu_j(N) \le \sum_{N<n\le N+q} \nu_j(n) = \sum_{1\le n\le N+q} \big(\nu_j(n) - \nu_j(n-q)\big) \le q\,\nu_j(N+q),$$

and putting this into (24) gives

$$\limsup_{N\to\infty} \sum_{0\le j<m} \big(\nu_j(N)/N\big)^2 \le (q-m)/qmb^{2m} + 1/qb^m. \tag{25}$$

This formula has been established whenever q is a multiple of m; and if we let $q \to \infty$ we obtain (17), completing the proof.

For a possibly simpler proof, see J. W. S. Cassels, *Pacific J. Math.* **2** (1952), 555–557. ∎

Exercises 29 and 30 illustrate the nontriviality of this theorem, and they also demonstrate the fact that a q-distributed sequence will have probabilities deviating from the true (m, m)-distribution probabilities by essentially $1/\sqrt{q}$ at most. (Cf. (25).) The full hypothesis of ∞-distribution is necessary for the proof of the theorem.

As a result of Theorem C, we can prove that an ∞-distributed sequence passes the serial test, the maximum-of-t test, the collision test, and the tests on subsequences mentioned in Section 3.3.2; it is not hard to show that the gap test, the poker test, and the run test are also satisfied (see exercises 12 through 14). The coupon collector's test is considerably more difficult to deal with, but it too is satisfied (see exercises 15 and 16).

The existence of ∞-distributed sequences of a rather simple type is guaranteed by the next theorem.

Theorem F (J. Franklin). *The* $[0,1)$ *sequence* $U_0, U_1, \ldots,$ *with*

$$U_n = \theta^n \bmod 1 \tag{26}$$

is ∞*-distributed for almost all real numbers* $\theta > 1$. *That is, the set*

$$\{\theta \mid \theta > 1 \text{ and (26) is not } \infty\text{-distributed}\}$$

is of measure zero.

The proofs of this theorem and some generalizations are given in Franklin's paper cited below. ∎

Franklin has shown that θ must be a transcendental number for (26) to be ∞-distributed. The powers $(\pi^n \bmod 1)$ have been laboriously computed for $n \leq 10000$, using multiple-precision arithmetic, and the most significant 35 bits of each of these numbers, stored on a disk file, have successfully been used as a source of uniform deviates. According to Theorem F, the probability that the powers $(\pi^n \bmod 1)$ are ∞-distributed is equal to 1; yet because there are uncountably many real numbers, this gives us no information as to whether the sequence is really ∞-distributed or not. It is a fairly safe bet that nobody in our lifetimes will ever *prove* that this particular sequence is *not* ∞-distributed; but it might not be. Because of these considerations, one may legitimately wonder if there is any *explicit* sequence that is ∞-distributed; i.e., *is there an algorithm to compute real numbers* U_n *for all* $n \geq 0$, *such that the sequence* $\langle U_n \rangle$ *is* ∞*-distributed?* The answer is yes, as shown for example by D. E. Knuth in *BIT* **5** (1965), 246–250. The sequence constructed there consists entirely of rational numbers; in fact, each number U_n has a terminating representation in the binary number system. Another construction of an explicit ∞-distributed sequence, somewhat more complicated than the sequence just cited, follows from Theorem W below. See also N. M. Korobov, *Izv. Akad. Nauk SSSR* **20** (1956), 649–660.

C. Does ∞-distributed = random? In view of all the above theory about ∞-distributed sequences, we can be sure of one thing: the concept of an ∞-distributed sequence is an important one in mathematics. There is also a good deal of evidence that the following statement is a valid formulation of the intuitive idea of randomness:

Definition R1. *A* $[0,1)$ *sequence is defined to be "random" if it is an* ∞*-distributed sequence.*

We have seen that sequences meeting this definition will satisfy all the statistical tests of Section 3.3.2 and many more.

Let us attempt to criticize this definition objectively. First of all, is every "truly random" sequence ∞-distributed? There are uncountably many sequences $U_0, U_1, \ldots$ of real numbers between zero and one. If a truly random number generator is sampled to give values $U_0, U_1, \ldots$, any of the possible sequences may be considered equally likely, and some of the sequences (indeed, uncountably many of them) are not even equidistributed. On the other hand, using any reasonable definition of probability on this space of all possible sequences leads us to conclude that a random sequence is ∞-distributed *with probability one.* We are therefore led to formalize Franklin's definition of randomness (as given at the beginning of this section) in the following way:

Definition R2. *A $[0,1)$ sequence $\langle U_n \rangle$ is defined to be "random" if, whenever P is a property such that $P(\langle V_n \rangle)$ holds with probability one for a sequence $\langle V_n \rangle$ of independent samples of random variables from the uniform distribution, then $P(\langle U_n \rangle)$ is true.*

Is it perhaps possible that Definition R1 is equivalent to Definition R2? Let us try out some possible objections to Definition R1, and see whether these criticisms are valid.

In the first place, Definition R1 deals only with limiting properties of the sequence as $n \to \infty$. There are ∞-distributed sequences in which the first million elements are all zero; should such a sequence be considered random?

This objection is not very valid. If ϵ is any positive number, there is no reason why the first million elements of a sequence should not all be less than ϵ. With probability one, a truly random sequence contains infinitely many runs of a million consecutive elements less than ϵ, so why can't this happen at the beginning of the sequence?

On the other hand, consider Definition R2 and let P be the property that all elements of the sequence are distinct; P is true with probability one, so any sequence with a million zeros is not random by *this* criterion.

Now let P be the property that *no* element of the sequence is equal to zero; again, P is true with probability one, so by Definition R2 any sequence with a zero element is nonrandom. More generally, however, let x_0 be any fixed number between zero and one, and let P be the property that no element of the sequence is equal to x_0; Definition R2 now says that no random sequence may contain the element x_0! We can now prove that *no sequence satisfies the condition of Definition R2.* (For if $U_0, U_1, \ldots$ is such a sequence, take $x_0 = U_0$.)

Therefore if R1 is too weak a definition, R2 is certainly too strong. The "right" definition must be less strict than R2. We have not really shown that R1 is too weak, however, so let us continue to attack it some more. As mentioned above, an ∞-distributed sequence of *rational* numbers has been constructed. (Indeed, this is not so surprising; see exercise 18.) Almost all real numbers are irrational; perhaps we should insist that

$$\Pr(U_n \text{ is rational}) = 0$$

for a random sequence.

Note that the definition of equidistribution says that $\Pr(u \le U_n < v) = v - u$. There is an obvious way to generalize this definition, using measure theory: "If $S \subseteq [0,1)$ is a set of measure μ, then

$$\Pr(U_n \in S) = \mu, \tag{27}$$

for all random sequences $\langle U_n \rangle$." In particular, if S is the set of rationals, it has measure zero, so no sequence of rational numbers is equidistributed in this generalized sense. It is reasonable to expect that Theorem B could be extended to Lebesgue integration instead of Riemann integration, if property (27) is stipulated. However, once again we find that definition (27) is too strict, for *no* sequence satisfies that property. If $U_0, U_1, \ldots$ is any sequence, the set $S = \{U_0, U_1, \ldots\}$ is of measure zero, yet $\Pr(U_n \in S) = 1$. Thus, by the force of the same argument we used to exclude rationals from random sequences, we can exclude all random sequences.

So far Definition R1 has proved to be defensible. There are, however, some quite valid objections to it. For example, if we have a random sequence in the intuitive sense, the infinite subsequence

$$U_0, U_1, U_4, U_9, \ldots, U_{n^2}, \ldots \tag{28}$$

should also be a random sequence. This is not always true for an ∞-distributed sequence. In fact, if we take any ∞-distributed sequence and set $U_{n^2} \leftarrow 0$ for all n, the counts $\nu_k(n)$ that appear in the test of k-distributivity are changed by at most $\sqrt{n}$, so the limits of the ratios $\nu_k(n)/n$ remain unchanged. Definition R1 unfortunately fails to satisfy this randomness criterion.

Perhaps we should strengthen R1 as follows:

Definition R3. *A $[0,1)$ sequence is said to be "random" if each of its infinite subsequences is ∞-distributed.*

Once again, however, the definition turns out to be too strict; any equidistributed sequence $\langle U_n \rangle$ has a monotonic subsequence with $U_{s_0} < U_{s_1} < U_{s_2} < \cdots$.

The secret is to restrict the subsequences so that they could be defined by somebody who does not look at U_n before deciding whether or not it is to be in the subsequence. The following definition now suggests itself:

Definition R4. *A $[0,1)$ sequence $\langle U_n \rangle$ is said to be "random" if, for every effective algorithm that specifies an infinite sequence of distinct nonnegative integers s_n for $n \ge 0$, the subsequence $U_{s_0}, U_{s_1}, U_{s_2}, \ldots$ corresponding to this algorithm is ∞-distributed.*

The algorithms referred to in Definition R4 are effective procedures that compute s_n, given n. (See Section 1.1.) Thus, for example, the sequence $\langle \pi^n \bmod 1 \rangle$ will *not* satisfy R4, since it is either not equidistributed or there is an effective algorithm that determines an infinite subsequence s_n with $(\pi^{s_0} \bmod 1) < (\pi^{s_1} \bmod 1) < (\pi^{s_2} \bmod 1) < \cdots$. Similarly, *no explicitly defined sequence can satisfy Definition R4*; this is appropriate, if we agree that no explicitly defined sequence can really be random. It is quite likely, however, that the sequence $\langle \theta^n \bmod 1 \rangle$ will satisfy Definition R4, for almost all real numbers $\theta > 1$; this is no

contradiction, since almost all θ are uncomputable by algorithms. The following facts are known, for example: (i) The sequence $\langle \theta^n \bmod 1 \rangle$ satisfies Definition R4 for almost all real $\theta > 1$, if "∞-distributed" is replaced by "1-distributed." This theorem was proved by J. F. Koksma, *Compositio Mathematica* **2** (1935), 250–258. (ii) The particular sequence $\langle \theta^{s(n)} \bmod 1 \rangle$ is ∞-distributed for almost all real $\theta > 1$, if $\langle s(n) \rangle$ is a sequence of integers for which $s(n+1) - s(n) \to \infty$ as $n \to \infty$. For example, we could have $s(n) = n^2$, or $s(n) = \lfloor n \lg n \rfloor$.

Definition R4 is much stronger than Definition R1; but it is still reasonable to claim that Definition R4 is too weak. For example, let $\langle U_n \rangle$ be a truly random sequence, and define the subsequence $\langle U_{s_n} \rangle$ by the following rules: $s_0 = 0$, and (for $n > 0$) s_n is the smallest integer $\geq n$ for which U_{s_n-1}, U_{s_n-2}, $\ldots$, U_{s_n-n} are all less than $\frac{1}{2}$. Thus we are considering the subsequence of values following the first consecutive run of n values less than $\frac{1}{2}$. Suppose that "$U_n < \frac{1}{2}$" corresponds to the value "heads" in the flipping of a coin. Gamblers tend to feel that a long run of "heads" makes the opposite condition, "tails," more probable, assuming that a true coin is being used; and the subsequence $\langle U_{s_n} \rangle$ just defined corresponds to a gambling system for a man who places his nth bet on the coin toss following the first run of n consecutive "heads." The gambler may think that $\Pr(U_{s_n} \geq \frac{1}{2})$ is more than $\frac{1}{2}$, but of course in a truly random sequence $\langle U_{s_n} \rangle$ will be completely random. No gambling system will ever be able to beat the odds! Definition R4 says nothing about subsequences formed according to such a gambling system, so apparently we need something more.

Let us define a "subsequence rule" $\mathcal{R}$ as an infinite sequence of functions $\langle f_n(x_1, \ldots, x_n) \rangle$ where, for $n \geq 0$, f_n is a function of n variables, and the value of $f_n(x_1, \ldots, x_n)$ is either 0 or 1. Here x_1, $\ldots$, x_n are elements of some set S. (Thus, in particular, f_0 is a constant function, either 0 or 1.) A subsequence rule $\mathcal{R}$ defines a subsequence of any infinite sequence $\langle X_n \rangle$ of elements of S as follows: *The nth term X_n is in the subsequence $\langle X_n \rangle \mathcal{R}$ if and only if* $f_n(X_0, X_1, \ldots, X_{n-1}) = 1$. Note that the subsequence $\langle X_n \rangle \mathcal{R}$ thus defined is not necessarily infinite, and it may in fact contain no elements at all.

For example, the gambler's subsequence just described corresponds to the following subsequence rule: "$f_0 = 1$; and for $n > 0$, $f_n(x_1, \ldots, x_n) = 1$ if and only if there is some k in the range $0 < k \leq n$ such that the k consecutive parameters x_m, x_{m-1}, $\ldots$, x_{m-k+1} are all $< \frac{1}{2}$ when $m = n$ but not when $k \leq m < n$."

A subsequence rule $\mathcal{R}$ is said to be *computable* if there is an effective algorithm that determines the value of $f_n(x_1, \ldots, x_n)$, when n and x_1, $\ldots$, x_n are given as input. We had better restrict ourselves to computable subsequence rules when trying to define randomness, lest we obtain an overly restrictive definition like R3 above. But effective algorithms cannot deal nicely with arbitrary real numbers as inputs; for example, if a real number x is specified by an infinite radix-10 expansion, there is no algorithm to determine if x is $< \frac{1}{3}$ or not, since all digits of the number $0.333\ldots$ have to be examined. Therefore computable subsequence rules do not apply to all $[0, 1)$ sequences, and it is convenient to base our next definition on b-ary sequences.

Definition R5. *A b-ary sequence is said to be "random" if every infinite subsequence defined by a computable subsequence rule is 1-distributed.*

A $[0,1)$ sequence $\langle U_n \rangle$ is said to be "random" if the b-ary sequence $\langle \lfloor bU_n \rfloor \rangle$ is "random" for all integers $b \geq 2$.

Note that Definition R5 says only "1-distributed," not "∞-distributed." It is interesting to verify that this may be done without loss of generality. For we may define an obviously computable subsequence rule $\mathcal{R}(a_1 \ldots a_k)$ as follows, given any b-ary number $a_1 \ldots a_k$: Let $f_n(x_1, \ldots, x_n) = 1$ if and only if $n \geq k-1$ and $x_{n-k+1} = a_1, \ldots, x_{n-1} = a_{k-1}, x_n = a_k$. Now if $\langle X_n \rangle$ is a k-distributed b-ary sequence, this rule $\mathcal{R}(a_1 \ldots a_k)$—which selects the subsequence consisting of those terms just following an occurrence of $a_1 \ldots a_k$—defines an infinite subsequence; and if this subsequence is 1-distributed, each of the $(k+1)$-tuples $a_1 \ldots a_k a_{k+1}$ for $0 \leq a_{k+1} < b$ occurs with probability $1/b^{k+1}$ in $\langle X_n \rangle$. Thus we can prove that a sequence satisfying Definition R5 is k-distributed for all k, by induction on k. Similarly, by considering the "composition" of subsequence rules—if $\mathcal{R}_1$ defines an infinite subsequence $\langle X_n \rangle \mathcal{R}_1$, then we can define $\mathcal{R}_1 \mathcal{R}_2$ to be the subsequence rule for which $\langle X_n \rangle \mathcal{R}_1 \mathcal{R}_2 = (\langle X_n \rangle \mathcal{R}_1) \mathcal{R}_2$—we find that all subsequences considered in Definition R5 are ∞-distributed. (See exercise 32.)

The fact that ∞-distribution comes out of Definition R5 as a very special case is encouraging, and it is a good indication that we may at last have found the definition of randomness we have been seeking. But alas, there still is a problem. It is not clear that sequences satisfying Definition R5 must satisfy Definition R4. The "computable subsequence rules" we have just specified always enumerate subsequences $\langle X_{s_n} \rangle$ for which $s_0 < s_1 < \cdots$, but $\langle s_n \rangle$ does not have to be monotone in R4; it must only satisfy the condition $s_n \neq s_m$ for $n \neq m$.

To meet this objection, we may combine Definitions R4 and R5 as follows:

Definition R6. *A b-ary sequence $\langle X_n \rangle$ is said to be "random" if, for every effective algorithm that specifies an infinite sequence of distinct nonnegative integers $\langle s_n \rangle$ as a function of n and the values of $X_{s_0}, \ldots, X_{s_{n-1}}$, the subsequence $\langle X_{s_n} \rangle$ corresponding to this algorithm is "random" in the sense of Definition R5.*

A $[0,1)$ sequence $\langle U_n \rangle$ is said to be "random" if the b-ary sequence $\langle \lfloor bU_n \rfloor \rangle$ is "random" for all integers $b \geq 2$.

The author contends* that this definition surely meets all reasonable philosophical requirements for randomness, so it provides an answer to the principal question posed in this section.

D. Existence of random sequences. We have seen that Definition R3 is too strong, in the sense that no sequence can satisfy that definition; and the formulation of Definitions R4, R5, and R6 above was carried out in an attempt to recapture the essential characteristics of Definition R3. In order to show that Definition R6 is not overly restrictive, it is still necessary for us to prove that sequences satisfying all these conditions exist. Intuitively, we feel quite sure that there is no problem,

*At least, he made such a contention when originally preparing the material for this section in 1966.

because we believe that a truly random sequence exists and satisfies R6; but a proof is really necessary to show that the definition is consistent.

An interesting method for constructing sequences satisfying Definition R5 has been found by A. Wald, starting with a very simple 1-distributed sequence.

Lemma T. *Let the sequence of real numbers $\langle V_n \rangle$ be defined in terms of the binary system as follows:*

$$V_0 = 0, \quad V_1 = .1, \quad V_2 = .01, \quad V_3 = .11, \quad V_4 = .001, \quad \ldots$$
$$V_n = .c_r \ldots c_1 1 \quad \text{if } n = 2^r + c_1 2^{r-1} + \cdots + c_r. \tag{29}$$

Let $I_{b_1 \ldots b_r}$ denote the set of all real numbers in $[0, 1)$ whose binary representation begins with $0.b_1 \ldots b_r$; thus

$$I_{b_1 \ldots b_r} = \big[(0.b_1 \ldots b_r)_2, (0.b_1 \ldots b_r)_2 + 2^{-r}\big). \tag{30}$$

Then if $\nu(n)$ denotes the number of V_k in $I_{b_1 \ldots b_r}$ for $0 \le k < n$, we have

$$|\nu(n)/n - 2^{-r}| \le 1/n. \tag{31}$$

Proof. Since $\nu(n)$ is the number of k for which $k \bmod 2^r = (b_r \ldots b_1)_2$, we have $\nu(n) = t$ or $t + 1$ when $\lfloor n/2^r \rfloor = t$. Hence $|\nu(n) - n/2^r| \le 1$. ∎

It follows from (31) that the sequence $\langle \lfloor 2^r V_n \rfloor \rangle$ is an equidistributed 2^r-ary sequence; hence by Theorem A, $\langle V_n \rangle$ is an equidistributed $[0, 1)$ sequence. Indeed, it is pretty clear that $\langle V_n \rangle$ is about as equidistributed as a $[0, 1)$ sequence can be. (For further discussion of this and related sequences, see J. G. van der Corput, *Proc. Koninklijke Nederl. Akad. Wetenschappen* **38** (1935), 813–821, 1058–1066; J. H. Halton, *Numerische Math.* **2** (1960), 84–90, 196; L. H. Ramshaw, *J. Number Theory*, to appear.)

Now let $\mathcal{R}_1, \mathcal{R}_2, \ldots$ be infinitely many subsequence rules; we seek a sequence $\langle U_n \rangle$ for which all the infinite subsequences $\langle U_n \rangle \mathcal{R}_j$ are equidistributed.

Algorithm W (*Wald sequence*). Given an infinite sequence of subsequence rules $\mathcal{R}_1, \mathcal{R}_2, \ldots$ that define subsequences of $[0, 1)$ sequences of rational numbers, this procedure defines a $[0, 1)$ sequence $\langle U_n \rangle$. The computation involves infinitely many auxiliary variables $C[a_1, \ldots, a_r]$, where $r \ge 1$ and where $a_j = 0$ or 1 for $1 \le j \le r$. These variables are initially all zero.

W1. [Initialize n.] Set $n \leftarrow 0$.

W2. [Initialize r.] Set $r \leftarrow 1$.

W3. [Test $\mathcal{R}_r$.] If the element U_n is to be in the subsequence defined by $\mathcal{R}_r$, based on the values of U_k for $0 \le k < n$, set $a_r \leftarrow 1$; otherwise set $a_r \leftarrow 0$.

W4. [$B[a_1, \ldots, a_r]$ full?] If $C[a_1, \ldots, a_r] < 3 \cdot 4^{r-1}$, go to W6.

W5. [Increase r.] Set $r \leftarrow r + 1$ and return to W3.

W6. [Set U_n.] Increase the value of $C[a_1, \ldots, a_r]$ by 1 and let k be its new value. Set $U_n \leftarrow V_k$, where V_k is defined in Lemma T above.

W7. [Advance n.] Increase n by 1 and return to W2. ∎

Strictly speaking, this is not an algorithm, since it doesn't terminate; it would of course be easy to modify the procedure to stop when n reaches a given value. The reader will find it easier to grasp the idea of the construction by trying it out manually, replacing the number $3 \cdot 4^{r-1}$ of step W4 by 2^r during this experiment.

Algorithm W is not meant to be a practical source of random numbers. It is intended to serve only a theoretical purpose:

Theorem W. *Let $\langle U_n \rangle$ be the sequence of rational numbers defined by Algorithm W, and let k be a positive integer. If the subsequence $\langle U_n \rangle \mathcal{R}_k$ is infinite, it is 1-distributed.*

Proof. Let $A[a_1, \ldots, a_r]$ denote the (possibly empty) subsequence of $\langle U_n \rangle$ containing precisely those elements U_n that, for all $j \le r$, belong to subsequence $\langle U_n \rangle \mathcal{R}_j$ if $a_j = 1$ and do not belong to subsequence $\langle U_n \rangle \mathcal{R}_j$ if $a_j = 0$.

It suffices to prove, for all $r \ge 1$ and all pairs of binary numbers $a_1 \ldots a_r$ and $b_1 \ldots b_r$, that $\Pr(U_n \in I_{b_1 \ldots b_r}) = 2^{-r}$ with respect to the subsequence $A[a_1, \ldots, a_r]$, whenever the latter is infinite. (See Eq. (30).) For if $r \ge k$, the infinite sequence $\langle U_n \rangle \mathcal{R}_k$ is the finite union of the disjoint subsequences $A[a_1, \ldots, a_r]$ for $a_k = 1$ and $a_j = 0$ or 1 for $1 \le j \le r$, $j \ne k$; and it follows that $\Pr(U_n \in I_{b_1 \ldots b_r}) = 2^{-r}$ with respect to $\langle U_n \rangle \mathcal{R}_k$. (See exercise 33.) This is enough to show that the sequence is 1-distributed, by Theorem A.

Let $B[a_1, \ldots, a_r]$ denote the subsequence of $\langle U_n \rangle$ that consists of the values for those n in which $C[a_1, \ldots, a_r]$ is increased by one in step W6 of the algorithm. By the algorithm, $B[a_1, \ldots, a_r]$ is a finite sequence with at most $3 \cdot 4^{r-1}$ elements. All but a finite number of the members of $A[a_1, \ldots, a_r]$ come from the subsequences $B[a_1, \ldots, a_r, \ldots, a_t]$, where $a_j = 0$ or 1 for $r < j \le t$.

Now assume that $A[a_1, \ldots, a_r]$ is infinite, and let $A[a_1, \ldots, a_r] = \langle U_{s_n} \rangle$, where $s_0 < s_1 < s_2 \le \cdots$. If N is a large integer, with $4^r \le 4^q < N \le 4^{q+1}$, it follows that the number of values of $k < N$ for which U_{s_k} is in $I_{b_1 \ldots b_r}$ is (except for finitely many elements at the beginning of the subsequence)

$$\nu(N) = \nu(N_1) + \cdots + \nu(N_m).$$

Here m is the number of subsequences $B[a_1, \ldots, a_t]$ listed above in which U_{s_k} appears for some $k < N$; N_j is the number of values of k with U_{s_k} in the corresponding subsequence; and $\nu(N_j)$ is the number of such values that are also in $I_{b_1 \ldots b_r}$. Therefore by Lemma T,

$$\begin{aligned} |\nu(N) - 2^{-r}N| &= |\nu(N_1) - 2^{-r}N_1 + \cdots + \nu(N_m) - 2^{-r}N_m| \\ &\le |\nu(N_1) - 2^{-r}N_1| + \cdots + |\nu(N_m) - 2^{-r}N_m| \\ &\le m \le 1 + 2 + 4 + \cdots + 2^{q-r+1} < 2^{q+1}. \end{aligned}$$

The inequality on m follows here from the fact that, by our choice of N, U_{s_N} is in $B[a_1, \ldots, a_t]$ for some $t \le q + 1$.

We have proved that

$$|\nu(N)/N - 2^{-r}| \le 2^{q+1}/N < 2/\sqrt{N}. \quad \blacksquare$$

To show finally that sequences satisfying Definition R5 exist, we note first that if $\langle U_n \rangle$ is a $[0,1)$ sequence of rational numbers and if $\mathcal{R}$ is a computable subsequence rule for a b-ary sequence, we can make $\mathcal{R}$ into a computable subsequence rule $\mathcal{R}'$ for $\langle U_n \rangle$ by letting $f'_n(x_1, \ldots, x_n)$ in $\mathcal{R}'$ equal $f_n(\lfloor bx_1 \rfloor, \ldots, \lfloor bx_n \rfloor)$ in $\mathcal{R}$. If the $[0,1)$ sequence $\langle U_n \rangle \mathcal{R}'$ is equidistributed, so is the b-ary sequence $\langle \lfloor bU_n \rfloor \rangle \mathcal{R}$. Now the set of all computable subsequence rules for b-ary sequences, for all values of b, is countable (since only countably many effective algorithms are possible), so they may be listed in some sequence $\mathcal{R}_1$, $\mathcal{R}_2$, ...; therefore Algorithm W defines a $[0,1)$ sequence that is random in the sense of Definition R5.

This brings us to a somewhat paradoxical situation. As we mentioned earlier, no effective algorithm can define a sequence that satisfies Definition R4, and for the same reason there is no effective algorithm that defines a sequence satisfying Definition R5. A proof of the existence of such random sequences is necessarily nonconstructive; how then can Algorithm W construct such a sequence?

There is no contradiction here; we have merely stumbled on the fact that the set of all effective algorithms cannot be enumerated by an effective algorithm. In other words, there is no effective algorithm to select the jth computable subsequence rule $\mathcal{R}_j$; this happens because there is no effective algorithm to determine if a computational method ever terminates. (We shall return to this topic in Chapter 11.) Important large classes of algorithms *can* be systematically enumerated; thus, for example, Algorithm W shows that it is possible to construct, with an effective algorithm, a sequence that satisfies Definition R5 if we restrict consideration to subsequence rules that are "primitive recursive."

By modifying step W6 of Algorithm W, so that it sets $U_n \leftarrow V_{k+t}$ instead of V_k, where t is any nonnegative integer depending on $a_1, \ldots, a_r$, we can show that there are *uncountably* many $[0,1)$ sequences satisfying Definition R5.

The following theorem shows still another way to prove the existence of uncountably many random sequences, using a less direct argument based on measure theory, even if the strong definition R6 is used:

Theorem M. *Let the real number x, $0 \le x < 1$, correspond to the binary sequence $\langle X_n \rangle$ if the binary representation of x is $(0.X_0X_1\ldots)_2$. Under this correspondence, almost all x correspond to binary sequences that are random in the sense of Definition R6.* (In other words, the set of all real x that correspond to a binary sequence that is nonrandom by Definition R6 has measure zero.)

Proof. Let $\mathcal{S}$ be an effective algorithm that determines an infinite sequence of distinct nonnegative integers $\langle s_n \rangle$, where the choice of s_n depends only on n and X_{s_k} for $0 \le k < n$; and let $\mathcal{R}$ be a computable subsequence rule. Then any binary sequence $\langle X_n \rangle$ leads to a subsequence $\langle X_{s_n} \rangle \mathcal{R}$, and Definition R6 says this subsequence must either be finite or 1-distributed. It suffices to prove that *for fixed $\mathcal{R}$ and $\mathcal{S}$ the set $N(\mathcal{R}, \mathcal{S})$ of all real x corresponding to $\langle X_n \rangle$, such that $\langle X_{s_n} \rangle \mathcal{R}$ is infinite and not 1-distributed, has measure zero.* For x has a nonrandom binary representation if and only if x is in $\bigcup N(\mathcal{R}, \mathcal{S})$, taken over the countably many choices of $\mathcal{R}$ and $\mathcal{S}$.

Therefore let $\mathcal{R}$ and $\mathcal{S}$ be fixed. Consider the set $T(a_1a_2\ldots a_r)$ defined for all binary numbers $a_1a_2\ldots a_r$ as the set of all x corresponding to $\langle X_n\rangle$, such that $\langle X_{s_n}\rangle\mathcal{R}$ has $\geq r$ elements whose first r elements are respectively equal to a_1, a_2, $\ldots$, a_r. Our first result is that

$$T(a_1a_2\ldots a_r) \text{ has measure } \leq 2^{-r}. \tag{32}$$

To prove this, we start by observing that $T(a_1a_2\ldots a_r)$ is a measurable set: Each element of $T(a_1a_2\ldots a_r)$ is a real number $x = (0.X_0X_1\ldots)_2$ for which there exists an integer m such that algorithm $\mathcal{S}$ determines distinct values $s_0, s_1, \ldots, s_m$, and rule $\mathcal{R}$ determines a subsequence of $X_{s_0}, X_{s_1}, \ldots, X_{s_m}$ such that X_{s_m} is the rth element of this subsequence. The set of all real $y = (0.Y_0Y_1\ldots)_2$ such that $Y_{s_k} = X_{s_k}$ for $0 \leq k \leq m$ also belongs to $T(a_1a_2\ldots a_r)$, and this is a measurable set consisting of the finite union of dyadic subintervals $I_{b_1\ldots b_t}$. Since there are only countably many such dyadic intervals, we see that $T(a_1a_2\ldots a_r)$ is a countable union of dyadic intervals, and it is therefore measurable. Furthermore, this argument can be extended to show that the measure of $T(a_1\ldots a_{r-1}\,0)$ equals the measure of $T(a_1\ldots a_{r-1}\,1)$, since the latter is a union of dyadic intervals obtained from the former by requiring that $Y_{s_k} = X_{s_k}$ for $0 \leq k < m$ and $Y_{s_m} \neq X_{s_m}$. Now since

$$T(a_1\ldots a_{r-1}\,0) \cup T(a_1\ldots a_{r-1}\,1) \subseteq T(a_1\ldots a_{r-1}),$$

the measure of $T(a_1a_2\ldots a_r)$ is at most one-half the measure of $T(a_1\ldots a_{r-1})$. The inequality (32) follows by induction on r.

Now that (32) has been established, the remainder of the proof is essentially to show that the binary representations of almost all real numbers are equidistributed. The next few paragraphs constitute a rather long but not difficult proof of this fact, and they serve to provide probability estimates that are useful in many other problems.

For $0 < \epsilon < 1$, let $B(r,\epsilon)$ be $\bigcup T(a_1\ldots a_r)$, where the union is taken over all binary numbers $a_1\ldots a_r$ for which the number $\nu(r)$ of zeros among $a_1\ldots a_r$ satisfies

$$|\nu(r) - \tfrac{1}{2}r| \geq 1 + \epsilon r.$$

The number of such binary numbers is $C(r,\epsilon) = \sum\binom{r}{k}$ summed over the values of k with $|k - \frac{1}{2}r| \geq 1 + \epsilon r$. A suitable estimate for the tail of the binomial distribution can be obtained by the following standard trick: Let x and p be any positive numbers less than 1, let $q = 1 - p$, and let $s = (p + \epsilon)r$. Then

$$\sum_{k\geq s}\binom{r}{k}p^kq^{r-k} \leq \sum_{k\geq s}\binom{r}{k}p^kq^{r-k}x^{s-k} \leq \sum_{k\geq 0}\binom{r}{k}p^kq^{r-k}x^{s-k} = x^s\Big(q + \frac{p}{x}\Big)^r.$$

By elementary calculus, the minimum value of $x^s(q + p/x)^r$ occurs when $x = (p/(p+\epsilon))/(q/(q-\epsilon))$, and this value of x yields

$$\sum_{k\geq(p+\epsilon)r}\binom{r}{k}p^kq^{r-k} \leq \left(\left(\frac{p}{p+\epsilon}\right)^{p+\epsilon}\left(\frac{q}{q-\epsilon}\right)^{q-\epsilon}\right)^r.$$

Now when $\epsilon \le \min(p,q)$ we have

$$\ln\bigl(p/(p+\epsilon)\bigr)^{p+\epsilon} = -\epsilon - \frac{\epsilon^2}{2p} + \frac{\epsilon^3}{6p^2} - \frac{\epsilon^4}{12p^3} + \cdots < -\epsilon,$$

$$\ln\bigl(q/(q-\epsilon)\bigr)^{q-\epsilon} = +\epsilon - \frac{\epsilon^2}{2q} - \frac{\epsilon^3}{6q^2} - \frac{\epsilon^4}{12q^3} - \cdots < \epsilon - \frac{\epsilon^2}{2q},$$

so we have the following bound for all $r > 0$ and $0 \le \epsilon \le \min(p,q)$:

$$\sum_{k \ge (p+\epsilon)r} \binom{r}{k} p^k q^{r-k} < e^{-\epsilon^2 r/(2q)}. \tag{33}$$

But $C(r,\epsilon)$ is 2^{r+1} times this left-hand quantity, in the special case $p = q = \frac{1}{2}$, hence by (32)

$$B(r,\epsilon) \text{ has measure } \le 2^{-r} C(r,\epsilon) < 2e^{-\epsilon^2 r}.$$

The next step is to define

$$B^*(r,\epsilon) = B(r,\epsilon) \cup B(r+1,\epsilon) \cup B(r+2,\epsilon) \cup \cdots.$$

The measure of $B^*(r,\epsilon)$ is at most $\sum_{k \ge r} 2e^{-\epsilon^2 k}$, and this is the remainder of a convergent series, so

$$\lim_{r\to\infty} \bigl(\text{measure of } B^*(r,\epsilon)\bigr) = 0. \tag{34}$$

Now if x is a real number whose binary expansion $(0.X_0X_1\ldots)_2$ leads to an infinite sequence $\langle X_{s_n}\rangle \mathcal{R}$ that is not 1-distributed, and if $\nu(r)$ denotes the number of zeros in the first r elements of the latter sequence, then

$$|\nu(r)/r - \tfrac{1}{2}| \ge 2\epsilon,$$

for some $\epsilon > 0$ and infinitely many r. This means x is in $B^*(r,\epsilon)$ for all r. So finally we find that

$$N(\mathcal{R},S) = \bigcup_{t\ge 2} \bigcap_{r\ge 1} B^*(r,1/t);$$

and, by (34), $\bigcap_{r\ge 1} B^*(r,1/t)$ has measure zero for all t. Hence $N(\mathcal{R},S)$ has measure zero. ∎

From the existence of *binary* sequences satisfying Definition R6, we can show the existence of $[0,1)$ sequences that are random in this sense. For details, see exercise 36. The consistency of Definition R6 is thereby established.

E. Random finite sequences. An argument was given above to indicate that it is impossible to define the concept of randomness for finite sequences; any given finite sequence is as likely as any other. Still, nearly everyone would agree that the sequence 011101001 is "more random" than 101010101, and even the latter

sequence is "more random" than 000000000. Although it is true that truly random sequences will exhibit locally nonrandom behavior, we would expect such behavior only in a long finite sequence, not in a short one.

Several ways for defining the randomness of a finite sequence have been proposed, and only a few of the ideas will be sketched here. Let us consider only the case of b-ary sequences.

Given a b-ary sequence $X_1, X_2, \ldots, X_N$, we can say that

$$\Pr\big(S(n)\big) \approx p, \qquad \text{if } |\nu(N)/N - p| \le 1/\sqrt{N},$$

where $\nu(n)$ is the quantity appearing in Definition A at the beginning of this section. The above sequence can be called "k-distributed" if

$$\Pr(X_n X_{n+1} \ldots X_{n+k-1} = x_1 x_2 \ldots x_k) \approx 1/b^k$$

for all b-ary numbers $x_1 x_2 \ldots x_k$. (Cf. Definition D; unfortunately a sequence may be k-distributed by this new definition when it is not $(k-1)$-distributed.)

A definition of randomness may now be given analogous to Definition R1, as follows:

Definition Q1. *A b-ary sequence of length N is "random" if it is k-distributed (in the above sense) for all positive integers $k \le \log_b N$.*

According to this definition, for example, there are 170 nonrandom binary sequences of length 11:

00000001111	10000000111	11000000011	11100000001
00000001110	10000000110	11000000010	11100000000
00000001101	10000000101	11000000001	10100000001
00000001011	10000000011	01000000011	01100000001
00000000111			

plus 01010101010 and all sequences with nine or more zeros, plus all sequences obtained from the preceding sequences by interchanging ones and zeros.

Similarly, we can formulate a definition for finite sequences analogous to Definition R6. Let **A** be a set of algorithms, each of which is a combination selection and choice procedure that gives a subsequence $\langle X_{s_n} \rangle \mathcal{R}$, as in the proof of Theorem M.

Definition Q2. *The b-ary sequence $X_1, X_2, \ldots, X_N$ is (n, ϵ)-random with respect to a set of algorithms* **A**, *if for every subsequence $X_{t_1}, X_{t_2}, \ldots, X_{t_m}$ determined by an algorithm of* **A** *we have either $m < n$ or*

$$\left| \frac{1}{m} \nu_a(X_{t_1}, \ldots, X_{t_m}) - \frac{1}{b} \right| \le \epsilon \qquad \text{for } 0 \le a < b.$$

Here $\nu_a(x_1, \ldots, x_m)$ is the number of a's in the sequence $x_1, \ldots, x_m$.

(In other words, every sufficiently long subsequence determined by an algorithm of **A** must be approximately equidistributed.) The basic idea in this case is to let **A** be a set of "simple" algorithms; the number (and the complexity) of the algorithms in **A** can grow as N grows.

As an example of Definition Q2, let us consider binary sequences, and let **A** be just the following four algorithms:

a) Take the whole sequence.
b) Take alternate terms of the sequence, starting with the first.
c) Take the terms of the sequence following a zero.
d) Take the terms of the sequence following a one.

Now a sequence $X_1, \ldots, X_8$ is $(4, \frac{1}{8})$-random if:

by (a), $|\frac{1}{8}(X_1 + \cdots + X_8) - \frac{1}{2}| \le \frac{1}{8}$, i.e., if there are 3, 4, or 5 ones;

by (b), $|\frac{1}{4}(X_1 + X_3 + X_5 + X_7) - \frac{1}{2}| \le \frac{1}{8}$, i.e., if there are two ones in odd-numbered positions;

by (c), there are three possibilities depending on how many zeros occupy positions $X_1, \ldots, X_7$: if there are 2 or 3 zeros here, there is no condition to test (since $n = 4$); if there are 4 zeros, they must respectively be followed by two zeros and two ones; and if there are 5 zeros, they must respectively be followed by two or three zeros;

by (d), we get conditions similar to those implied by (c).

It turns out that only the following binary sequences of length 8 are $(4, \frac{1}{8})$-random with respect to these rules:

00001011	00101001	01001110	01101000
00011010	00101100	01011011	01101100
00011011	00110010	01011110	01101101
00100011	00110011	01100010	01110010
00100110	00110110	01100011	01110110
00100111	00111001	01100110	

plus those obtained by interchanging 0 and 1 consistently.

It is clear that we could make the set of algorithms so large that no sequences satisfy the definition, when n and ϵ are reasonably small. A. N. Kolomogorov has proved that an (n, ϵ)-random binary sequence *will* always exist, for any given N, if the number of algorithms in **A** does not exceed

$$\frac{1}{2}e^{2n\epsilon^2(1-\epsilon)}. \tag{35}$$

This result is not nearly strong enough to show that sequences satisfying Definition Q1 will exist, but the latter can be constructed efficiently using the procedure of Rees in exercise 3.2.2–21.

Still another interesting approach to a definition of randomness has been taken by Per Martin-Löf [*Information and Control* **9** (1966), 602–619]. Given a

finite b-ary sequence $X_1, \ldots, X_N$, let $l(X_1, \ldots, X_N)$ be the length of the shortest Turing machine program that generates this sequence. (For a definition of Turing machines, see Chapter 11; alternatively, we could use certain classes of effective algorithms, such as those defined in exercise 1.1–8.) Then $l(X_1, \ldots, X_N)$ is a measure of the "patternlessness" of the sequence, and we may equate this idea with randomness. The sequences of length N that maximize $l(X_1, \ldots, X_N)$ may be called random. (From the standpoint of practical random number generation by computer, this is, of course, the worst definition of "randomness" that can be imagined!)

Essentially the same definition of randomness was independently given by G. Chaitin at about the same time; see *JACM* **16** (1969), 145–159. It is interesting to note that even though this definition makes no reference to equidistribution properties as our other definitions have, Martin-Löf and Chaitin have proved that random sequences of this type also have the expected equidistribution properties. In fact, Martin-Löf has demonstrated that such sequences satisfy *all* computable statistical tests for randomness, in an appropriate sense.

For further developments in the definition of random finite sequences, see A. K. Zvonkin and L. A. Levin, *Uspekhi Mat. Nauk* **25**,6 (Nov. 1970), 85–127 [English translation in *Russian Math. Surveys* **25**,6 (Nov. 1970), 83–124]; L. A. Levin, *Doklady Akad. Nauk SSSR* **212** (1973), 548–550.

F. Summary, history, and bibliography. We have defined several degrees of randomness that a sequence might possess.

An infinite sequence that is ∞-distributed satisfies a great many useful properties that are expected of random sequences, and there is a rich theory concerning ∞-distributed sequences. (The exercises that follow develop several important properties of ∞-distributed sequences that have not been mentioned in the text.) Definition R1 is therefore an appropriate basis for theoretical studies of randomness.

The concept of an ∞-distributed b-ary sequence was introduced in 1909 by Emile Borel. He essentially defined the concept of an (m, k)-distributed sequence, and showed that the b-ary representations of almost all real numbers are (m, k)-distributed for all m and k. He called such numbers *normal* to base b. An excellent discussion of this topic appears in his well-known book, *Leçons sur la théorie des fonctions* (2nd ed., 1914), 182–216.

The notion of an ∞-distributed sequence of *real* numbers, also called a *completely equidistributed sequence*, first appeared in a note by N. M. Korobov in *Doklady Akad. Nauk SSSR* **62** (1948), 21–22. Korobov and several of his colleagues developed the theory of such sequences quite extensively in a series of papers during the 1950s. Completely equidistributed sequences were independently studied by Joel N. Franklin, *Math. Comp.* **17** (1963), 28–59, in a paper that is particularly noteworthy because it was inspired by the problem of random number generation. The book *Uniform Distribution of Sequences* by L. Kuipers and H. Niederreiter (New York: Wiley, 1974) is an extraordinarily complete source of information about the rich mathematical literature concerning k-distributed sequences of all kinds.

We have seen, however, that ∞-distributed sequences need not be sufficiently haphazard to qualify completely as "random." Three definitions, R4, R5, and R6, were formulated above to provide the additional conditions; and Definition R6, in particular, seems to be an appropriate way to define the concept of an infinite random sequence. It is a precise, quantitative statement that may well coincide with the intuitive idea of true randomness.

Historically, the development of these definitions was primarily influenced by R. von Mises' quest for a good definition of "probability." In *Math. Zeitschrift* **5** (1919), 52–99, von Mises proposed a definition similar in spirit to Definition R5, although stated too strongly (like our Definition R3) so that no sequences satisfying the conditions could possibly exist. Many people noticed this discrepancy, and A. H. Copeland [*Amer. J. Math.* **50** (1928), 535–552] suggested weakening von Mises' definition by substituting what he called "admissible numbers" (or Bernoulli sequences). These are equivalent to ∞-distributed $[0,1)$ sequences in which all entries U_n have been replaced by 1 if $U_n < p$ or by 0 if $U_n \geq p$, for a given probability p. Thus Copeland was essentially suggesting a return to Definition R1. Then Abraham Wald showed that it is not necessary to weaken von Mises' definition so drastically, and he proposed substituting a countable set of subsequence rules. In an important paper [*Ergebnisse eines math. Kolloquiums* **8** (Vienna, 1937), 38–72], Wald essentially proved Theorem W, although he made the erroneous assertion that the sequence constructed by Algorithm W also satisfies the stronger condition that $\Pr(U_n \in A) =$ measure of A, for all Lebesgue measurable $A \subseteq [0,1)$. We have observed that no sequence can satisfy this property.

The concept of "computability" was still very much in its infancy when Wald wrote his paper, and A. Church [*Bull. Amer. Math. Soc.* **47** (1940), 130–135] showed how the precise notion of "effective algorithm" could be added to Wald's theory to make his definitions completely rigorous. The extension to Definition R6 was due essentially to A. N. Kolmogorov [*Sankhyā* (A) **25** (1963), 369–376], who proposed Definition Q2 for finite sequences in that same paper. Another definition of randomness for finite sequences, somewhere "between" Definitions Q1 and Q2, had been formulated many years earlier by A. S. Besicovitch [*Math. Zeitschrift* **39** (1934), 146–156].

The publications of Church and Kolmogorov considered only binary sequences for which $\Pr(X_n = 1) = p$ for a given probability p. Our discussion in this section has been slightly more general, since a $[0,1)$ sequence essentially represents all p at once. The von Mises–Wald–Church definition has been refined in yet another interesting way by J. V. Howard, *Zeitschr. f. math. Logik und Grundlagen d. Math.* **21** (1975), 215–224.

Another important contribution was made by Donald W. Loveland [*Zeitschr. f. math. Logik und Grundlagen d. Math.* **12** (1966), 279–294], who discussed Definitions R4, R5, R6, and several intermediate concepts. Loveland proved that there are R5-random sequences that do not satisfy R4, thereby establishing the need for a stronger definition such as R6. In fact, he defined a rather simple permutation $\langle f(n) \rangle$ of the nonnegative integers, and an Algorithm W′ analogous

to Algorithm W, such that $\overline{\Pr}(U_{f(n)} \geq \frac{1}{2}) - \underline{\Pr}(U_{f(n)} \geq \frac{1}{2}) \geq \frac{1}{2}$ for every R5-random sequence $\langle U_n \rangle$ produced by Algorithm W′ when it is given an infinite set of subsequence rules $\mathcal{R}_k$.

Although Definition R6 is intuitively much stronger than R4, it is apparently not a simple matter to prove this rigorously, and for several years it was an open question whether or not R4 implies R6. Finally Thomas Herzog and James C. Owings, Jr., discovered how to construct a large family of sequences that satisfy R4 but not R6. [See *Zeitschr. f. math. Logik und Grundlagen d. Math.* **22** (1976), 385–389.]

Kolmogorov wrote another significant paper [*Problemy Peredači Informatsii* **1** (1965), 3–11] in which he considered the problem of defining the "information content" of a sequence, and this work led to Chaitin and Martin-Löf's interesting definition of finite random sequences via "patternlessness." [See *IEEE Trans.* **IT-14** (1968), 662–664.]

For a philosophical discussion of random sequences, see K. R. Popper, *The Logic of Scientific Discovery* (London, 1959), especially the interesting construction on pp. 162–163, which he first published in 1934.

Further connections between random sequences and recursive function theory have been explored by D. W. Loveland, *Trans. Amer. Math. Soc.* **125** (1966), 497–510. See also C.-P. Schnorr [*Zeitschr. Wahr. verw. Geb.* **14** (1969), 27–35], who found strong relations between random sequences and the "species of measure zero" defined by L. E. J. Brouwer in 1919. Schnorr's subsequent book *Zufälligkeit und Wahrscheinlichkeit* [*Lecture Notes in Math.* **218** (Berlin: Springer, 1971)] gives a detailed treatment of the entire subject of randomness and makes an excellent introduction to the ever-growing advanced literature on the topic.

EXERCISES

1. [*10*] Can a periodic sequence be equidistributed?

2. [*10*] Consider the periodic binary sequence 0, 0, 1, 1, 0, 0, 1, 1, ... Is it 1-distributed? Is it 2-distributed? Is it 3-distributed?

3. [*M22*] Construct a periodic ternary sequence that is 3-distributed.

▶ **4.** [*HM22*] Let $U_n = (2^{\lfloor \lg(n+1) \rfloor}/3) \bmod 1$. What is $\Pr(U_n < \frac{1}{2})$?

5. [*HM14*] Prove that $\Pr(S(n) \text{ and } T(n)) + \Pr(S(n) \text{ or } T(n)) = \Pr(S(n)) + \Pr(T(n))$, for any two statements $S(n)$, $T(n)$, provided that at least three of the limits exist. For example, if a sequence is 2-distributed, we would find that

$$\Pr(u_1 \leq U_n < v_1 \text{ or } u_2 \leq U_{n+1} < v_2) = v_1 - u_1 + v_2 - u_2 - (v_1 - u_1)(v_2 - u_2).$$

6. [*HM23*] Let $S_1(n)$, $S_2(n)$, ... be an infinite sequence of statements about mutually disjoint events; i.e., $S_i(n)$ and $S_j(n)$ cannot simultaneously be true if $i \neq j$. Assume that $\Pr(S_j(n))$ exists for each $j \geq 1$. Show that $\underline{\Pr}(S_j(n) \text{ is true for some } j \geq 1) \geq \sum_{j \geq 1} \Pr(S_j(n))$, and give an example to show that equality need not hold.

7. [*HM27*] Let $\{S_{ij}(n)\}$ be a family of statements such that $\Pr(S_{ij}(n))$ exists for all $i, j \geq 1$. Assume that for all $n > 0$, $S_{ij}(n)$ is true for exactly one pair of integers i, j. If $\sum_{i,j\geq 1} \Pr(S_{ij}(n)) = 1$, does it follow that "$\Pr(S_{ij}(n)$ is true for some $j \geq 1$)" exists for all $i \geq 1$, and that it equals $\sum_{j\geq 1} \Pr(S_{ij}(n))$?

8. [*M15*] Prove (13).

9. [*HM20*] Prove Lemma E. [*Hint:* Consider $\sum_{1\leq j\leq m}(y_{jn} - \alpha)^2$.]

▶ **10.** [*HM22*] Where was the fact that m divides q used in the proof of Theorem C?

11. [*M10*] Use Theorem C to prove that if $\langle U_n\rangle$ is ∞-distributed, so is the subsequence $\langle U_{2n}\rangle$.

12. [*HM20*] Show that a k-distributed sequence passes the "maximum-of-k test," in the following sense: $\Pr(u \leq \max(U_n, U_{n+1}, \ldots, U_{n+k-1}) < v) = v^k - u^k$.

▶ **13.** [*HM27*] Show that an ∞-distributed $[0, 1)$ sequence passes the "gap test" in the following sense: If $0 \leq \alpha < \beta \leq 1$ and $p = \beta - \alpha$, let $f(0) = 0$, and for $n \geq 1$ let $f(n)$ be the smallest integer $m > f(n-1)$ such that $\alpha \leq U_m < \beta$; then

$$\Pr(f(n) - f(n-1) = k) = p(1-p)^{k-1}.$$

14. [*HM25*] Show that an ∞-distributed sequence passes the "run test" in the following sense: If $f(0) = 1$ and if, for $n \geq 1$, $f(n)$ is the smallest integer $m > f(n-1)$ such that $U_{m-1} > U_m$, then

$$\Pr(f(n) - f(n-1) = k) = 2k/(k+1)! - 2(k+1)/(k+2)!.$$

▶ **15.** [*HM30*] Show that an ∞-distributed sequence passes the "coupon-collector's test" when there are only two kinds of coupons, in the following sense: Let $X_1, X_2, \ldots$ be an ∞-distributed binary sequence. Let $f(0) = 0$, and for $n \geq 1$ let $f(n)$ be the smallest integer $m > f(n-1)$ such that $\{X_{f(n-1)+1}, \ldots, X_m\}$ is the set $\{0, 1\}$. Prove that $\Pr(f(n) - f(n-1) = k) = 2^{1-k}$, for $k \geq 2$. (Cf. exercise 7.)

16. [*HM38*] Does the coupon-collector's test hold for ∞-distributed sequences when there are more than two kinds of coupons? (Cf. the previous exercise.)

17. [*HM50*] If r is any given rational number, Franklin has proved that the sequence $\langle r^n \bmod 1\rangle$ is not 2-distributed. But is there any rational number r for which this sequence is equidistributed? In particular, is the sequence equidistributed when $r = \frac{3}{2}$? [Cf. K. Mahler, *Mathematika* **4** (1957), 122–124.]

▶ **18.** [*HM22*] Prove that if $U_0, U_1, \ldots$ is k-distributed, so is the sequence $V_0, V_1, \ldots$, where $V_n = \lfloor nU_n\rfloor/n$.

19. [*HM35*] Consider a modification of Definition R4 that requires the subsequences to be only 1-distributed instead of ∞-distributed. Is there a sequence that satisfies this weaker definition, but that is not ∞-distributed? (Is the weaker definition really weaker?)

20. [*HM47*] Does the sequence $\langle \theta^n \bmod 1\rangle$ satisfy Definition R4 for almost all real numbers $\theta > 1$?

21. [*HM20*] Let S be a set and let $\mathcal{M}$ be a collection of subsets of S. Suppose that p is a real-valued function of the sets in $\mathcal{M}$, such that $p(M)$ denotes the probability that a "randomly" chosen element of S lies in M. Generalize Definitions B and D to obtain a good definition of the concept of a k-distributed sequence $\langle Z_n\rangle$ of elements of S with respect to the probability distribution p.

▸ **22.** [*HM30*] (Hermann Weyl.) Show that the $[0,1)$ sequence $\langle U_n \rangle$ is k-distributed if and only if

$$\lim_{N\to\infty} \frac{1}{N} \sum_{0\le n<N} \exp(2\pi i(c_1U_n + \cdots + c_kU_{n+k-1})) = 0$$

for every set of integers $c_1, c_2, \ldots, c_k$ not all zero.

23. [*M34*] Show that a b-ary sequence $\langle X_n \rangle$ is k-distributed if and only if all of the sequences $\langle c_1X_n + c_2X_{n+1} + \cdots + c_kX_{n+k-1} \rangle$ are *equidistributed*, whenever $c_1, c_2, \ldots, c_k$ are integers with $\gcd(c_1, \ldots, c_k) = 1$.

24. [*M25*] Show that a $[0,1)$ sequence $\langle U_n \rangle$ is k-distributed if and only if all of the sequences $\langle c_1U_n + c_2U_{n+1} + \cdots + c_kU_{n+k-1} \rangle$ are *equidistributed*, whenever $c_1, c_2, \ldots, c_k$ are integers not all zero.

25. [*HM20*] A sequence is called a "white sequence" if all serial correlations are zero, i.e., if the equation in Corollary S is true for *all* $k \ge 1$. (By Corollary S, an ∞-distributed sequence is white.) Show that if a $[0,1)$ sequence is equidistributed, it is white if and only if

$$\lim_{n\to\infty} \frac{1}{n} \sum_{0\le j<n} (U_j - \tfrac{1}{2})(U_{j+k} - \tfrac{1}{2}) = 0, \qquad \text{for all } k \ge 1.$$

26. [*HM34*] (J. Franklin.) A white sequence, as defined in the previous exercise, can definitely fail to be random. Let $U_0, U_1, \ldots$ be an ∞-distributed sequence, and define the sequence $V_0, V_1, \ldots$ as follows:

$$\begin{aligned} (V_{2n-1}, V_{2n}) &= (U_{2n-1}, U_{2n}) \qquad \text{if } (U_{2n-1}, U_{2n}) \in G, \\ (V_{2n-1}, V_{2n}) &= (U_{2n}, U_{2n-1}) \qquad \text{if } (U_{2n-1}, U_{2n}) \notin G, \end{aligned}$$

where G is the set $\{(x,y) \mid x - \frac{1}{2} \le y \le x \text{ or } x + \frac{1}{2} \le y\}$. Show that (a) $V_0, V_1, \ldots$ is equidistributed and white; (b) $\Pr(V_n > V_{n+1}) = \frac{5}{8}$. (This points out the weakness of the serial correlation test.)

27. [*HM48*] What is the highest possible value for $\Pr(V_n > V_{n+1})$ in an equidistributed, white sequence? (D. Coppersmith has constructed such a sequence achieving the value $\frac{7}{8}$.)

▸ **28.** [*HM21*] Use the sequence (11) to construct a $[0,1)$ sequence that is 3-distributed, for which $\Pr(U_{2n} \ge \frac{1}{2}) = \frac{3}{4}$.

29. [*HM34*] Let $X_0, X_1, \ldots$ be a $(2k)$-distributed binary sequence. Show that

$$\overline{\Pr}(X_{2n} = 0) \le \frac{1}{2} + \binom{2k-1}{k} \Big/ 2^{2k}.$$

▸ **30.** [*M39*] Construct a binary sequence that is $(2k)$-distributed, and for which

$$\Pr(X_{2n} = 0) = \frac{1}{2} + \binom{2k-1}{k} \Big/ 2^{2k}.$$

(Therefore the inequality in the previous exercise is the best possible.)

31. [*M30*] Show that $[0,1)$ sequences exist that satisfy Definition R5, yet $\nu_n/n \geq \frac{1}{2}$ for all $n > 0$, where ν_n is the number of $j < n$ for which $U_n < \frac{1}{2}$. (This might be considered a nonrandom property of the sequence.)

32. [*M24*] Given that $\langle X_n \rangle$ is a "random" b-ary sequence according to Definition R5, and that $\mathcal{R}$ is a computable subsequence rule that specifies an infinite subsequence $\langle X_n \rangle \mathcal{R}$, show that the latter subsequence is not only 1-distributed, it is "random" by Definition R5.

33. [*HM22*] Let $\langle U_{r_n} \rangle$ and $\langle U_{s_n} \rangle$ be infinite disjoint subsequences of a sequence $\langle U_n \rangle$. (Thus, $r_0 < r_1 < r_2 < \cdots$ and $s_0 < s_1 < s_2 < \cdots$ are increasing sequences of integers and $r_m \neq s_n$ for any m, n.) Let $\langle U_{t_n} \rangle$ be the combined subsequence, so that $t_0 < t_1 < t_2 < \cdots$ and the set $\{t_n\} = \{r_n\} \cup \{s_n\}$. Show that if $\Pr(U_{r_n} \in A) = \Pr(U_{s_n} \in A) = p$, then $\Pr(U_{t_n} \in A) = p$.

▶ **34.** [*M25*] Define subsequence rules $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, ... such that Algorithm W can be used with these rules to give an effective algorithm to construct a $[0,1)$ sequence satisfying Definition R1.

▶ **35.** [*HM35*] (D. W. Loveland.) Show that if a binary sequence $\langle X_n \rangle$ is R5-random, and if $\langle s_n \rangle$ is any computable sequence as in Definition R4, then $\overline{\Pr}(X_{s_n} = 1) \geq \frac{1}{2}$ and $\underline{\Pr}(X_{s_n} = 1) \leq \frac{1}{2}$.

36. [*HM30*] Let $\langle X_n \rangle$ be a binary sequence that is "random" according to Definition R6. Show that the $[0,1)$ sequence $\langle U_n \rangle$ defined in binary notation by the scheme

$$\begin{aligned} U_0 &= (0.X_0)_2 \\ U_1 &= (0.X_1X_2)_2 \\ U_2 &= (0.X_3X_4X_5)_2 \\ U_3 &= (0.X_6X_7X_8X_9)_2 \\ &\ldots \end{aligned}$$

is random in the sense of Definition R6.

37. [*M37*] (D. Coppersmith.) Define a sequence that satisfies Definition R4 but not Definition R5. [*Hint:* Consider changing U_0, U_1, U_4, U_9, ... in a truly random sequence.]

38. [*M49*] (A. N. Kolmogorov.) Given N, n and ϵ, what is the smallest number of algorithms in a set $\mathbf{A}$ such that no (n,ϵ)-random binary sequences of length N exist with respect to $\mathbf{A}$? (If exact formulas cannot be given, can asymptotic formulas be found? The point of this problem is to discover how close the bound (35) comes to being "best possible.")

39. [*HM45*] (W. M. Schmidt.) Let U_n be a $[0,1)$ sequence, and let $\nu_n(u)$ be the number of nonnegative integers $j \leq n$ such that $0 \leq U_j < u$. Prove that there is a positive constant c such that, for any N and for any $[0,1)$ sequence $\langle U_n \rangle$, we have

$$|\nu_n(u) - un| > c \ln N$$

for some n and u with $0 \leq n < N$, $0 \leq u < 1$. (In other words, no $[0,1)$ sequence can be *too* equidistributed.)

▶ **40.** [*16*] (I. J. Good.) Can a valid table of random digits contain just one misprint?

3.6. SUMMARY

WE HAVE COVERED a fairly large number of topics in this chapter: how to generate random numbers, how to test them, how to modify them in applications, and how to derive theoretical facts about them. Perhaps the main question in many readers' minds will be, "What is the result of all this theory? What is a simple, virtuous generator I can use in my programs in order to have a reliable source of random numbers?"

The detailed investigations in this chapter suggest that the following procedure gives the "nicest" and "simplest" random number generator for the machine language of most computers: At the beginning of the program, set an integer variable X to some value X_0. This variable X is to be used only for the purpose of random number generation. Whenever a new random number is required by the program, set

$$X \leftarrow (aX + c) \bmod m \tag{1}$$

and use the new value of X as the random value. It is necessary to choose X_0, a, c, and m properly, and to use the random numbers wisely, according to the following principles:

i) The "seed" number X_0 may be chosen arbitrarily. If the program is run several times and a different source of random numbers is desired each time, set X_0 to the last value attained by X on the preceding run; or (if more convenient) set X_0 to the current date and time. If the program may need to be rerun later with the *same* random numbers (e.g., when debugging), be sure to print out X_0 if it isn't otherwise known.

ii) The number m should be large, say at least 2^{30}. It may conveniently be taken as the computer's word size, since this makes the computation of $(aX + c) \bmod m$ quite efficient. Section 3.2.1.1 discusses the choice of m in more detail. The computation of $(aX + c) \bmod m$ must be done *exactly*, with no roundoff error.

iii) If m is a power of 2 (i.e., if a binary computer is being used), pick a so that $a \bmod 8 = 5$. If m is a power of 10 (i.e., if a decimal computer is being used), choose a so that $a \bmod 200 = 21$. This choice of a together with the choice of c given below ensures that the random number generator will produce all m different possible values of X before it starts to repeat (see Section 3.2.1.2) and ensures high "potency" (see Section 3.2.1.3).

iv) The multiplier a should preferably be chosen between $.01m$ and $.99m$, and its binary or decimal digits should *not* have a simple, regular pattern. By choosing some haphazard constant like $a = 3141592621$ (which satisfies both of the conditions in (iii)), one almost always obtains a reasonably good multiplier. Further testing should of course be done if the random number generator is to be used extensively; for example, there should be no large quotients when Euclid's algorithm is used to find the gcd of a and m (see Section 3.3.3). The multiplier should pass the spectral test (Section 3.3.4)

and several tests of Section 3.3.2, before it is considered to have a truly clean bill of health.

v) The value of c is immaterial when a is a good multiplier, except that c must have no factor in common with m. Thus we may choose $c = 1$ or $c = a$.

vi) The least significant (right-hand) digits of X are not very random, so decisions based on the number X should always be influenced primarily by the most significant digits. It is generally best to think of X as a random fraction X/m between 0 and 1, that is, to visualize X with a decimal point at its left, rather than to regard X as a random integer between 0 and $m - 1$. To compute a random integer between 0 and $k - 1$, one should multiply by k and truncate the result (see the beginning of Section 3.4.2).

vii) An important limitation on the randomness of sequence (1) is discussed in Section 3.3.4, where it is shown that the "accuracy" in t dimensions will be only about one part in $\sqrt[t]{m}$. Monte Carlo applications requiring higher resolution can improve the randomness by employing techniques discussed in Section 3.2.2.

The above comments apply primarily to machine-language coding. In higher-level programming languages, we are often unable to use such machine-dependent features as integer arithmetic modulo the word size, and careful compilers will not allow us to compute the product of two large integers. Another technique that we might call the *subtractive method* (exercise 3.2.2–23) can be used to provide a "portable" random number generator that is efficiently describable in any higher-level programming language, since it makes use only of integer arithmetic between -10^9 and $+10^9$. Here is how the subtractive method might be coded in FORTRAN, as a subroutine that delivers an array of 55 random integers at once:

```
      FUNCTION IRN55(IA)
      DIMENSION IA(1)
C     ASSUMING THAT IA(1), ..., IA(55) HAVE BEEN SET UP PROPERLY,
C     THIS SUBROUTINE RESETS THE IA ARRAY TO THE NEXT 55 NUMBERS
C     OF A PSEUDO-RANDOM SEQUENCE, AND RETURNS THE VALUE 1.
      DO 1 I = 1, 24
        J = IA(I) - IA(I+31)
        IF (J .LT. 0)  J = J + 1000000000
        IA(I) = J
  1   CONTINUE
      DO 2 I = 25, 55
        J = IA(I) - IA(I-24)
        IF (J .LT. 0)  J = J + 1000000000
        IA(I) = J
  2   CONTINUE
      IRN55=1
      RETURN
      END
```

To use these numbers in a FORTRAN program, let JRAND be an auxiliary integer variable; we may obtain the next random number U (as a fraction between 0 and 1) by writing the following three statements:

```
      JRAND = JRAND + 1
      IF (JRAND .GT. 55)  JRAND = IRN55(IA)
      U = FLOAT(IA(JRAND)) * 1.0E-9
```

At the beginning of our program we need to write "DIMENSION IA(55)" and "JRAND=55" and we must also initialize the IA array. Appropriate initialization can be done by calling the following subroutine with IX set to any integer value (selected like X_0 in rule (i) above, preferably large):

```
      SUBROUTINE IN55(IA,IX)
      DIMENSION IA(1)
C     THIS SUBROUTINE SETS IA(1), ..., IA(55) TO STARTING
C     VALUES SUITABLE FOR LATER CALLS ON IRN55(IA).
C     IX IS AN INTEGER "SEED" VALUE BETWEEN 0 AND 999999999.
      IA(55) = IX
      J = IX
      K = 1
      DO 1 I = 1, 54
        II = MOD(21*I, 55)
        IA(II) = K
        K = J - K
        IF (K .LT. 0)  K = K + 1000000000
        J = IA(II)
  1   CONTINUE
C     THE NEXT THREE LINES "WARM UP" THE GENERATOR.
      IRN55(IA)
      IRN55(IA)
      IRN55(IA)
      RETURN
      END
```

(This subroutine computes a Fibonacci-like sequence; multiplication of indices by 21 spreads the values around so as to alleviate initial nonrandomness problems such as those in exercise 3.2.2–2. Note that $2^{29} < 10^9 < 2^{30}$; any large even number may actually be substituted for 10^9 in these routines, if a corresponding change is made in the computation of the random fraction U. Furthermore it would be possible to work directly with floating point numbers instead of integers by making appropriate changes to these programs, *provided* that the computer's floating point arithmetic is sufficiently accurate to give *exact* results in all the computations required here. Most binary computers will be able to meet this requirement when all of the numbers to be dealt with have the form $a/2^p$, where a is an integer, $0 \le a < 2^p$, and there are p bits of precision in floating point fractions. The numbers $(24, 55)$ in these routines may be replaced by any pair

of values (j, k) in Table 3.2.2–1, for $k \geq 50$; the constants 31, 25, 54, 21 should then be replaced by $k - j$, $j + 1$, $k - 1$, and d respectively, where d is relatively prime to k and $d \approx 0.382k$.)

Although a great deal is known about linear congruential sequences like (1), very little has yet been proved about the randomness properties of the subtractive method. Both approaches seem to be reliable in practice.

Unfortunately, quite a bit of published material in existence at the time this chapter was written recommends the use of generators that violate the suggestions above; and the most common generator in actual use, RANDU, is really horrible (cf. Section 3.3.4). The authors of many contributions to the science of random number generation were unaware that particular methods they were advocating would prove to be inadequate. Perhaps further research will show that even the random number generators recommended here are unsatisfactory; we hope this is not the case, but the history of the subject warns us to be cautious. The most prudent policy for a person to follow is to run each Monte Carlo program at least twice using quite different sources of random numbers, before taking the answers of the program seriously; this not only will give an indication of the stability of the results, it also will guard against the danger of trusting in a generator with hidden deficiencies. (Every random number generator will fail in at least one application.)

Excellent bibliographies of the pre-1972 literature on random number generation have been compiled by Richard E. Nance and Claude Overstreet, Jr., *Computing Reviews* **13** (1972), 495–508, and by E. R. Sowey, *International Stat. Review* **40** (1972), 355–371. The period 1972–1976 is covered by Sowey in *International Stat. Review* **46** (1978), 89–102.

For a detailed study of the use of random numbers in numerical analysis, see J. M. Hammersley and D. C. Handscomb, *Monte Carlo Methods* (London: Methuen, 1964). This book shows that some numerical methods are enhanced by using numbers that are "quasi-random," designed specifically for a certain purpose (not necessarily satisfying the statistical tests we have discussed).

Every reader is urged to work exercise 6 in the following set of problems.

EXERCISES

1. [*21*] Write a MIX subroutine with the following characteristics, using method (1):

Calling sequence:	JMP RANDI
Entry conditions:	rA $= k$, a positive integer < 5000.
Exit conditions:	rA $\leftarrow$ a random integer Y, $1 \leq Y \leq k$, with each integer about equally probable; rX =?; overflow off.

▸ **2.** [*15*] Some people have been afraid that computers will someday take over the world; but they are reassured by the statement that a machine cannot do anything really new, since it is only obeying the commands of its master, the programmer. Lady Lovelace wrote in 1844, "The Analytical Engine has no pretensions to *originate*

anything. It can do *whatever we know how to order it* to perform." Her statement has been further elaborated by many philosophers. Discuss this topic, with random number generators in mind.

3. [*32*] (*A dice game.*) Write a program that simulates a roll of two dice, each of which takes on the values 1, 2, ..., 6 with equal probability. If the total is 7 or 11 on the first roll, the game is won; a total of 2, 3, or 12 loses; and on any other total, call that total the "point" and continue rolling dice until either a 7 occurs (a loss) or the point occurs again (a win).

Play ten games. The result of each roll of the dice should be printed in the form $m\,n$, where m and n are the contents of the two dice, followed by some appropriate comment (like "snake eyes" or "little Joe" or "the hard way", etc.).

4. [*40*] (*Solitaire or patience.*) Some people spend a lot of valuable time playing card games of solitaire, and perhaps automation will make an important inroad in this area. Write a program that (a) shuffles a simulated deck of cards; (b) plays some common game of solitaire based on the order of the cards in the deck; (c) prints out the result of the game, i.e., how close the program came to winning. Several games should be played. The program might be set up to "cheat" upon request.

5. [*46*] (*Creative writing by computer.*) A television program entitled "The Thinking Machine," broadcast by the CBS television network on October 26, 1960, featured (among other things) two Western-style playlets that were written by a computer program. Here are the two scripts as they were printed out by the computer:

Saga #1. (The gun is in the right hand; the money is in the left hand; the glass is on the table; the bottle is on the table; the holster is on the robber; the sheriff's gun is in the sheriff's right hand; the sheriff's holster is on the sheriff.)

ROBBER: (The robber is at the window.) Go to door; open door; go thru door; close door; go to corner; put money down at corner; go to table; put gun on table; sit and fidgit; sit at table; pick up glass with right hand (empty); put glass on table; pick up bottle with right hand; pour; put bottle on table; pick up glass with right hand; take a drink from glass; put glass on table; pick up bottle with right hand; sit at table; sit at table; go to corner; go to window; go to table; put bottle on table; sit and fidgit; sit at table; sit and fidgit; go to window; go to table; pick up glass with right hand.

SHERIFF: (The sheriff is at the window.) See robber; (robber sees sheriff); go to door.

ROBBER: Put glass on table; pick up gun with right hand; check gun.

SHERIFF: Wait; open door; see robber; (robber sees sheriff); go thru door.

ROBBER: Go to window; aim; fire; SHERIFF NICKED.

SHERIFF: Go to window; aim; fire; MISSED; go to door; go to window.

ROBBER: Go to door; aim; aim.

SHERIFF: Aim; fire; MISSED.

ROBBER: Fire; SHERIFF NICKED.

SHERIFF: Go to door; aim; fire; MISSED; go thru door; aim.

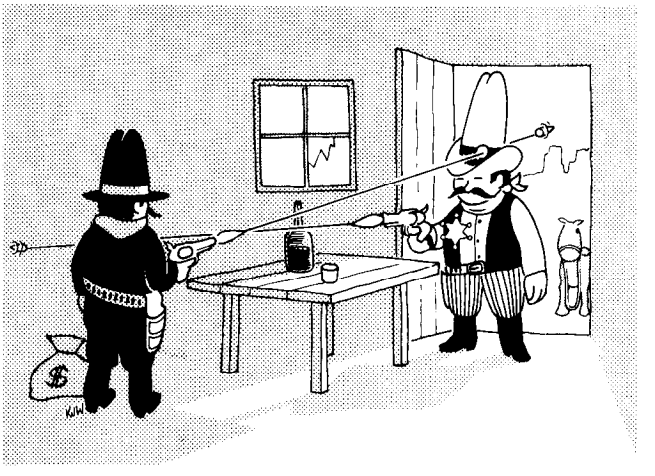
ROBBER: Aim; fire; MISSED; aim; fire; MISSED.

SHERIFF: Fire; MISSED; go to window; aim; fire; MISSED.

ROBBER: Aim; fire; MISSED; aim; fire; MISSED; aim; fire; SHERIFF NICKED.

SHERIFF: Aim; fire; ROBBER HIT.

ROBBER: Drop gun; robber dies.



SHERIFF: Put gun in holster; go to table; pick up glass with right hand (empty); take glass from right hand with left hand; pick up bottle with right hand; pour; put bottle on table; take glass from left hand with right hand; take a drink from glass; take glass from right hand with left hand; pick up bottle with right hand; pour; put bottle on table; take glass from left hand with right hand; take a drink from glass; put glass on table; go to corner; pick up money with right hand; go to door; go thru door; close door. CURTAIN.

Saga #2. (The gun is in the right hand; the money is in the left hand; the glass is on the table; the bottle is on the table; the holster is on the robber; the sheriff's gun is in the sheriff's right hand; the sheriff's holster is on the sheriff.)

ROBBER: (The robber is at the window.) Go to door; open door; go thru door; close door; go to corner; put money down at corner; go to window; put gun down at window; lean on window and look; lean on window and look; go to corner; count money; go to table; pick up glass with right hand (empty); take glass from right hand with left hand; pick up bottle with right hand; pour; put bottle on table; take glass from left hand with right hand; take a drink from glass; put glass on table; pick up bottle with right hand; pour; go to corner; put bottle down at corner; go to window; pick up gun with right hand; check gun; put gun in holster; go to table; pick up glass with right hand; take a drink from glass; go to window; put glass down at window.

SHERIFF: (The sheriff is at the window.) See robber; (robber sees sheriff); go to door.

ROBBER: Take gun from holster with right hand; check gun; go to door; check gun; put gun down at door.

SHERIFF: Open door; see robber; (robber sees sheriff); go thru door; go to window.

ROBBER: Pick up gun with right hand.

SHERIFF: Go to table.

ROBBER: Aim; fire; MISSED; aim; fire; SHERIFF HIT; blow out barrel; put gun in holster.

SHERIFF: Drop gun; sheriff dies.

ROBBER: Go to corner; pick up money with right hand; go to door; go thru door; close door. CURTAIN.

A careful reading of the above scripts reveals the highly intense drama present here. The computer program was careful to keep track of the locations of each player, the contents of his hands, etc. Actions taken by the players were random, governed by certain probabilities; the probability of a foolish action was increased depending on how much that player had had to drink and on how often he had been nicked in a shot. The reader will be able to deduce further properties of the program by studying the sample scripts.

Of course, even the best scripts are rewritten before they are produced, and this is especially true when an inexperienced writer has prepared the original draft. Here are the scripts just as they were actually used in the show:

Saga #1. Music up.
MS Robber peering thru window of shack.
CU Robber's face.
MS Robber entering shack.
CU Robber sees whiskey bottle on table.
CU Sheriff outside shack.
MS Robber sees sheriff.
LS Sheriff in doorway over shoulder of robber, both draw.
MS Sheriff drawing gun.
LS Shooting it out. Robber gets shot.
MS Sheriff picking up money bags.
MS Robber staggering.
MS Robber dying. Falls across table, after trying to take last shot at sheriff.
MS Sheriff walking thru doorway with money.
MS of robber's body, now still, lying across table top. Camera dollies back. (Laughter)

Saga #2. Music up.
CU of window. Robber appears.
MS Robber entering shack with two sacks of money.
MS Robber puts money bags on barrel.
CU Robber—sees whiskey on table.
MS Robber pouring himself a drink at table. Goes to count money. Laughs.
MS Sheriff outside shack.
MS thru window.
MS Robber sees sheriff thru window.
LS Sheriff entering shack. Draw. Shoot it out.
CU Sheriff. Writhing from shot.
M/2 shot Sheriff staggering to table for a drink . . . falls dead.
MS Robber leaves shack with money bags.*

[*Note:* CU = "close up", MS = "medium shot", etc. The above details were kindly furnished to the author by Thomas H. Wolf, producer of the television show, who suggested the idea of a computer-written playlet in the first place, and also by Douglas T. Ross and Harrison R. Morse who produced the computer program.]

The reader will undoubtedly have many ideas about how he could prepare his own computer program to do creative writing; and that is the point of this exercise.

▶ **6.** [*40*] Look at the subroutine library of each computer installation in your organization, and replace the random number generators by good ones. Try to avoid being too shocked at what you find.

▶ **7.** [*M40*] A programmer decided to encipher his files by using a linear congruential sequence $\langle X_n \rangle$ of period 2^{32} generated by (a, c, X_0) with $m = 2^{32}$. He took the most significant bits $\lfloor X_n/2^{16} \rfloor$ and exclusive-or'ed them onto his data, but kept the parameters a, c, and X_0 secret.

Show that this isn't a very secure scheme, by devising a method that deduces the multiplier a and the first difference $X_1 - X_0$ in a reasonable amount of time, given only the values of $\lfloor X_n/2^{16} \rfloor$ for $0 \leq n < 150$.

CHAPTER FOUR

ARITHMETIC

Seeing there is nothing (right well beloved Students in the Mathematickes) that is so troublesome to Mathematicall practise, nor that doth more molest and hinder Calculators, then the Multiplications, Divisions, square and cubical Extractions of great numbers, which besides the tedious expence of time are for the most part subject to many slippery errors. I began therefore to consider in my minde, by what certaine and ready Art I might remove those hindrances.

—JOHN NAPIER (1614)

I do hate sums. There is no greater mistake than to call arithmetic an exact science. There are . . . hidden laws of number which it requires a mind like mine to perceive. For instance, if you add a sum from the bottom up, and then again from the top down, the result is always different.

—MRS. LA TOUCHE (19th century)

I cannot conceive that anybody will require multiplications at the rate of 40,000, or even 4,000 per hour; such a revolutionary change as the octonary scale should not be imposed upon mankind in general for the sake of a few individuals.

—F. H. WALES (1936)

Most numerical analysts have no interest in arithmetic.

—B. PARLETT (1979)

THE CHIEF PURPOSE of this chapter is to make a careful study of the four basic processes of arithmetic: addition, subtraction, multiplication, and division. Many people regard arithmetic as a trivial thing that children learn and computers do, but we will see that arithmetic is a fascinating topic with many interesting facets. It is important to make a thorough study of efficient methods for calculating with numbers, since arithmetic underlies so many computer applications.

Arithmetic is, in fact, a lively subject that has played an important part in the history of the world, and it still is undergoing rapid development. In this chapter, we shall analyze algorithms for doing arithmetic operations on many types of quantities, such as "floating point" numbers, extremely large numbers, fractions (rational numbers), polynomials, and power series; and we will also discuss related topics such as radix conversion, factoring of numbers, and the evaluation of polynomials.

4.1. POSITIONAL NUMBER SYSTEMS

THE WAY WE DO ARITHMETIC is intimately related to the way we represent the numbers we deal with, so it is appropriate to begin our study of the subject with a discussion of the principal means for representing numbers.

Positional notation using base b (or *radix* b) is defined by the rule

$$(\ldots a_3a_2a_1a_0.a_{-1}a_{-2}\ldots)_b$$
$$= \cdots + a_3b^3 + a_2b^2 + a_1b^1 + a_0 + a_{-1}b^{-1} + a_{-2}b^{-2} + \cdots; \qquad (1)$$

for example, $(520.3)_6 = 5\cdot 6^2 + 2\cdot 6^1 + 0 + 3\cdot 6^{-1} = 192\frac{1}{2}$. Our conventional decimal number system is, of course, the special case when b is ten, and when the a's are chosen from the "decimal digits" 0, 1, 2, 3, 4, 5, 6, 7, 8, 9; in this case the subscript b in (1) may be omitted.

The simplest generalizations of the decimal number system are obtained when we take b to be an integer greater than 1 and when we require the a's to be integers in the range $0 \le a_k < b$. This gives us the standard binary ($b = 2$), ternary ($b = 3$), quaternary ($b = 4$), quinary ($b = 5$), ... number systems. In general, we could take b to be any nonzero number, and we could choose the a's from any specified set of numbers; this leads to some interesting situations, as we shall see.

The dot that appears between a_0 and a_{-1} in (1) is called the *radix point.* (When $b = 10$, it is also called the decimal point, and when $b = 2$, it is sometimes called the binary point, etc.) Continental Europeans often use a comma instead of a dot to denote the radix point; Englishmen often use a raised dot.

The a's in (1) are called the *digits* of the representation. A digit a_k for large k is often said to be "more significant" than the digits a_k for small k; accordingly, the leftmost or "leading" digit is referred to as the *most significant digit* and the rightmost or "trailing" digit is referred to as the *least significant digit.* In the standard binary system the binary digits are often called *bits*; in the standard hexadecimal system (radix sixteen) the hexadecimal digits zero through fifteen are usually denoted by

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

The historical development of number representations is a fascinating story, since it parallels the development of civilization itself. We would be going far afield if we were to examine this history in minute detail, but it will be instructive to look at its main features here.

The earliest forms of number representations, still found in primitive cultures, are generally based on groups of fingers, piles of stones, etc., usually with special conventions about replacing a larger pile or group of, say, five or ten objects by one object of a special kind or in a special place. Such systems lead naturally to the earliest ways of representing numbers in written form, as in the systems of Babylonian, Egyptian, Greek, Chinese, and Roman numerals; but such notations are comparatively inconvenient for performing arithmetic operations except in the simplest cases.

During the twentieth century, historians of mathematics have made extensive studies of early cuneiform tablets found by archeologists in the Middle East. These studies show that the Babylonian people actually had two distinct systems of number representation: Numbers used in everyday business transactions were written in a notation based on grouping by tens, hundreds, etc.; this notation was inherited from earlier Mesopotamian civilizations, and large numbers were seldom required. When more difficult mathematical problems were considered, however, Babylonian mathematicians made extensive use of a sexagesimal (radix sixty) positional notation that was highly developed at least as early as 1750 B.C. This notation was unique in that it was actually a *floating point* form of representation with exponents omitted; the proper scale factor or power of sixty was to be supplied by the context, so that, for example, the numbers 2, 120, 7200, and $\frac{1}{30}$ were all written in an identical manner. The notation was especially convenient for multiplication and division, using auxiliary tables, since radix-point alignment had no effect on the answer. As examples of this Babylonian notation, consider the following excerpts from early tables: The square of 30 is 15 (which may also be read, "The square of $\frac{1}{2}$ is $\frac{1}{4}$"); the reciprocal of $81 = (1\ 21)_{60}$ is $(44\ 26\ 40)_{60}$; and the square of the latter is $(32\ 55\ 18\ 31\ 6\ 40)_{60}$. The Babylonians had a symbol for zero, but because of their "floating point" philosophy, it was used only within numbers, not at the right end to denote a scale factor. For the interesting story of early Babylonian mathematics, see O. Neugebauer, *The Exact Sciences in Antiquity* (Princeton, N. J.: Princeton University Press, 1952), and B. L. van der Waerden, *Science Awakening*, tr. by A. Dresden (Groningen: P. Noordhoff, 1954); see also D. E. Knuth, *CACM* **15** (1972), 671–677; **19** (1976), 108.

Fixed point positional notation was apparently first conceived by the Maya Indians in central America 2000 years ago; their radix-20 system was highly developed, especially in connection with astronomical records and calendar dates. But the Spanish conquerors destroyed nearly all of the Maya books on history and science, so we have comparatively little knowledge about how sophisticated the native Americans had become at arithmetic; special-purpose multiplication tables have been found, but no examples of division are known [cf. J. Eric S. Thompson, *Contributions to Amer. Anthropology and History* **7** (Carnegie Inst. of Washington, 1942), 37–62].

Several centuries before Christ, the Greek people employed an early form of the abacus to do their arithmetical calculations, using sand and/or pebbles on a board that had rows or columns corresponding in a natural way to our decimal system. It is perhaps surprising to us that the same positional notation was never adapted to written forms of numbers, since we are so accustomed to reckoning with the decimal system using pencil and paper; but the greater ease of calculating by abacus (since handwriting was not a common skill, and since abacus calculation makes it unnecessary to memorize addition and multiplication tables) probably made the Greeks feel it would be silly even to suggest that computing could be done better on "scratch paper." At the same time Greek astronomers did make use of a sexagesimal positional notation for fractions, which they had learned from the Babylonians.

Our decimal notation, which differs from the more ancient forms primarily because of its fixed radix point, together with its symbol for zero to mark an empty position, was developed first in India within the Hindu culture. The exact date when this notation first appeared is quite uncertain; about 600 A.D. seems to be a good guess. Hindu science was rather highly developed at that time, particularly in astronomy. The earliest known Hindu manuscripts that show this notation have numbers written backwards (with the most significant digit at the right), but soon it became standard to put the most significant digit at the left.

About 750 A.D., the Hindu principles of decimal arithmetic were brought to Persia, as several important works were translated into Arabic; a picturesque account of this development is given in a Hebrew document, which has been translated into English in *AMM* **15** (1918), 99–108. Not long after this, al-Khwârizmî wrote his Arabic textbook on the subject. (As noted in Chapter 1, our word "algorithm" comes from al-Khwârizmî's name.) His work was translated into Latin and was a strong influence on Leonardo Pisano (Fibonacci), whose book on arithmetic (1202 A.D.) played a major role in the spreading of Hindu-Arabic numerals into Europe. It is interesting to note that the left-to-right order of writing numbers was unchanged during these two transitions, although Arabic is written from right to left while Hindu and Latin scholars generally wrote from left to right. A detailed account of the subsequent propagation of decimal numeration and arithmetic into all parts of Europe during the period from 1200 to 1600 A.D. has been given by David Eugene Smith in his *History of Mathematics* **1** (Boston: Ginn and Co., 1923), Chapters 6 and 8.

Decimal notation was applied at first only to integer numbers, not to fractions. Arabic astronomers, who required fractions in their star charts and other tables, continued to use the notation of Ptolemy (the famous Greek astronomer), a notation based on sexagesimal fractions. This system still survives today, in our trigonometric units of "degrees, minutes, and seconds," and also in our units of time, as a remnant of the original Babylonian sexagesimal notation. Early European mathematicians also used sexagesimal fractions when dealing with noninteger numbers; for example, Fibonacci gave the value

$$1^\circ\ 22'\ 7''\ 42'''\ 33^{IV}\ 4^{V}\ 40^{VI}$$

as an approximation to the root of the equation $x^3 + 2x^2 + 10x = 20$. (The correct answer is $1^\circ\ 22'\ 7''\ 42'''\ 33^{IV}\ 4^{V}\ 38^{VI}\ 30^{VII}\ 50^{VIII}\ 15^{IX}\ 43^{X}\ \ldots$.)

The use of decimal notation also for tenths, hundredths, etc., in a similar way seems to be a comparatively minor change; but, of course, it is hard to break with tradition, and sexagesimal fractions have an advantage over decimal fractions in that numbers such as $\frac{1}{3}$ can be expressed exactly, in a simple way.

Chinese mathematicians—who never used sexagesimals—were apparently the first people to work with the equivalent of decimal fractions, although their numeral system (lacking zero) was not originally a positional number system in the strict sense. Chinese units of weights and measures were decimal, so that Tsu Chhung-Chih (who died c. 500 A.D.) was able to express an approximation

to π in the following form:

3 chang, 1 chhih, 4 tshun, 1 fên, 5 li, 9 hao, 2 miao, 7 hu.

Here chang, ..., hu are units of length; 1 hu (the diameter of a silk thread) equals 1/10 miao, etc. The use of such decimal-like fractions was fairly widespread in China after about 1250 A.D.

The first known appearance of decimal fractions in a true positional system occurs in a 10th-century arithmetic text written in Damascus by an obscure mathematician named al-Uqlîdisî ("the Euclidean"). He used the symbol $'$ for a decimal point, for example in connection with a problem about compound interest, the computation of 135 times $(1.1)^n$ for $1 \leq n \leq 5$. [See A. S. Saidan, *The Arithmetic of al-Uqlîdisî* (Dordrecht: D. Reidel, 1975), 110, 114, 343, 355, 481–485.] But he did not develop the idea very fully, and his trick was soon forgotten; five centuries passed before decimal fractions were reinvented by a Persian mathematician, al-Kashî, who died c. 1436. Al-Kashî was a highly skillful calculator, who gave the value of 2π as follows, correct to 16 decimal places:

integer		fractions															
0	6	2	8	3	1	8	5	3	0	7	1	7	9	5	8	6	5

This was by far the best approximation to π known until Ludolph van Ceulen laboriously calculated 35 decimal places during the period 1596–1610.

The earliest known example of decimal fractions in Europe occurs in a 15th-century text where, for example, 153.5 is multiplied by 16.25 to get 2494.375; this was referred to as a "Turkish method." In 1525, Christof Rudolff of Germany discovered decimal fractions for himself; but like al-Uqlîdisî, his work seems to have had little influence. François Viète suggested the idea again in 1579. Finally, an arithmetic text by Simon Stevin of Belgium, who independently hit on the idea of decimal fractions in 1585, became popular. Stevin's work, and the discovery of logarithms soon afterwards, made decimal fractions commonplace in Europe during the 17th century. [See D. E. Smith, *History of Mathematics* **2** (Boston: Ginn and Co., 1925), 228–247, and C. B. Boyer, *History of Mathematics* (New York: Wiley, 1968), for further remarks and references.]

The binary system of notation has its own interesting history. Many primitive tribes in existence today are known to use a binary or "pair" system of counting (making groups of two instead of five or ten), but they do not count in a true radix-2 system, since they do not treat powers of 2 in a special manner. See *The Diffusion of Counting Practices* by Abraham Seidenberg, *Univ. Calif. Publ. in Math.* **3** (1960), 215–300, for interesting details about primitive number systems. Another "primitive" example of an essentially binary system is the conventional musical notation for expressing rhythms and durations of time.

Nondecimal number systems were discussed in Europe during the seventeenth century. For many years astronomers had occasionally used sexagesimal arithmetic both for the integer and the fractional parts of numbers, primarily when performing multiplication [see John Wallis, *Treatise of Algebra* (Oxford,

1685), 18–22, 30]. The fact that *any* integer greater than 1 could serve as radix was apparently first stated in print by Blaise Pascal in *De numeris multiplicibus*, which was written about 1658 [see Pascal's *Œuvres Complètes* (Paris: Éditions de Seuil, 1963), 84–89]. Pascal wrote, "Denaria enim ex institute hominum, non ex necessitate naturæ ut vulgus arbitratur, et sane satis inepte, posita est"; i.e., "The decimal system has been established, somewhat foolishly to be sure, according to man's custom, not from a natural necessity as most people would think." He stated that the duodecimal (radix twelve) system would be a welcome change, and he gave a rule for testing a duodecimal number for divisibility by nine. Erhard Weigel tried to drum up enthusiasm for the quaternary (radix four) system in a series of publications beginning in 1673. A detailed discussion of radix-twelve arithmetic was given by Joshua Jordaine, *Duodecimal Arithmetick* (London, 1687).

Although decimal notation was almost exclusively used for arithmetic during that era, other systems of weights and measures were rarely if ever based on multiples of 10, and many business transactions required a good deal of skill in adding quantities such as pounds, shillings, and pence. For centuries merchants had therefore learned to compute sums and differences of quantities expressed in peculiar units of currency, weights, and measures; and this was actually arithmetic in a nondecimal number system. The common units of liquid measure in England, dating from the 13th century or earlier, are particularly noteworthy:

2 gills = 1 chopin	2 demibushels = 1 bushel or firkin
2 chopins = 1 pint	2 firkins = 1 kilderkin
2 pints = 1 quart	2 kilderkins = 1 barrel
2 quarts = 1 pottle	2 barrels = 1 hogshead
2 pottles = 1 gallon	2 hogsheads = 1 pipe
2 gallons = 1 peck	2 pipes = 1 tun
2 pecks = 1 demibushel	

Quantities of liquid expressed in gallons, pottles, quarts, pints, etc. were essentially written in binary notation. Perhaps the true inventors of binary arithmetic were English wine merchants!

The first known appearance of pure binary notation was about 1605 in some unpublished manuscripts of Thomas Harriot (1560–1621). Harriot was a creative man, who first became famous by coming to America as a representative of Sir Walter Raleigh. He invented (among other things) a notation like that now used for "less than" and "greater than" relations; but for some reason he chose not to publish many of his discoveries. Excerpts from his notes on binary arithmetic have been reproduced by John W. Shirley, *Amer. J. Physics* **19** (1951), 452–454. The first published discussion of the binary system was given in a comparatively little-known work by a Spanish bishop, Juan Caramuel Lobkowitz, *Mathesis biceps* **1** (Campaniæ, 1670), 45–48; Caramuel discussed the representation of numbers in radices 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, and 60 at some length, but gave no examples of arithmetic operations in nondecimal systems (except for the trivial operation of adding unity).

Ultimately, an article by G. W. Leibniz [*Memoires de l'Academie Royale des Sciences* (Paris: 1703), 110–116], which illustrated binary addition, subtraction, multiplication, and division, really brought binary notation into the limelight, and this article is usually referred to as the birth of radix-2 arithmetic. Leibniz later referred to the binary system quite frequently. He did not recommend it for practical calculations, but he stressed its importance in number-theoretical investigations, since patterns in number sequences are often more apparent in binary notation than they are in decimal; he also saw a mystical significance in the fact that everything is expressible in terms of zero and one. Leibniz's unpublished manuscripts show that he had been interested in binary notation as early as 1679, when he referred to it as a "bimal" system (analogous to "decimal").

A careful study of Leibniz's early work with binary numbers has been made by Hans J. Zacher, *Die Hauptschriften zur Dyadik von G. W. Leibniz* (Frankfurt am Main: Klostermann, 1973). Zacher points out that Leibniz was familiar with John Napier's so-called "local arithmetic," a way for calculating with stones that amounts to using a radix-2 abacus. [Napier had published the idea of local arithmetic as an appendix to his little book *Rhabdologia* in 1617; it may be called the world's first "binary computer," and it is surely the world's cheapest, although Napier felt that it was more amusing than practical. See Martin Gardner's discussion in *Scientific American* **228** (April 1973), 106–111.]

It is interesting to note that the important concept of negative powers to the right of the radix point was not yet well understood at that time. Leibniz asked James Bernoulli to calculate π in the binary system, and Bernoulli "solved" the problem by taking a 35-digit approximation to π, multiplying it by 10^{35}, and then expressing this integer in the binary system as his answer. On a smaller scale this would be like saying that $\pi \approx 3.14$, and $(314)_{10} = (100111010)_2$; hence π in binary is 100111010! [See Leibniz, *Math. Schriften*, ed. by K. Gehrhardt, **3** (Halle: 1855), 97; two of the 118 bits in the answer are incorrect, due to computational errors.] The motive for Bernoulli's calculation was apparently to see whether any simple pattern could be observed in this representation of π.

Charles XII of Sweden, whose talent for mathematics perhaps exceeded that of all other kings in the history of the world, hit on the idea of radix-8 arithmetic about 1717. This was probably his own invention, although he had met Leibniz briefly in 1707. Charles felt that radix 8 or 64 would be more convenient for calculation than the decimal system, and he considered introducing octal arithmetic into Sweden; but he died in battle before decreeing such a change. [See *The Works of Voltaire* **21** (Paris: E. R. DuMont, 1901), 49; E. Swedenborg, *Gentleman's Magazine* **24** (1754), 423–424.]

Octal notation was proposed also in colonial America before 1750, by the Rev. Hugh Jones, rector of a parish in Maryland [cf. *Gentleman's Magazine* **15** (1745), 377–379; H. R. Phalen, *AMM* **56** (1949), 461–465].

More than a century later, a prominent Swedish-American civil engineer named John W. Nystrom decided to carry Charles XII's plans a step further, by devising a complete system of numeration, weights, and measures based on radix-16 arithmetic. He wrote, "I am not afraid, or do not hesitate, to advocate a

binary system of arithmetic and metrology. I know I have nature on my side; if I do not succeed to impress upon you its utility and great importance to mankind, it will reflect that much less credit upon our generation, upon our scientific men and philosophers." Nystrom devised special means for pronouncing hexadecimal numbers; e.g., $(\texttt{B0160})_{16}$ was to be read "vybong, bysanton." His entire system was called the Tonal System, and it is described in *J. Franklin Inst.* **46** (1863), 263–275, 337–348, 402–407. A similar system, but using radix 8, was worked out by Alfred B. Taylor [*Proc. Amer. Pharmaceutical Assoc.* **8** (1859), 115–216; *Proc. Amer. Philosophical Soc.* **24** (1887), 296–366]. Increased use of the French (metric) system of weights and measures prompted extensive debate about the merits of decimal arithmetic during that era; indeed, octal arithmetic was even being proposed in France [J. D. Colenne, *Le système octaval* (Paris: 1845); Aimé Mariage, *Numération par huit* (Paris: Le Nonnant, 1857)].

The binary system was well known as a curiosity ever since Leibniz's time, and about 20 early references to it have been compiled by R. C. Archibald [*AMM* **25** (1918), 139–142]. It was applied chiefly to the calculation of powers, as explained in Section 4.6.3, and to the analysis of certain games and puzzles. Giuseppe Peano [*Atti della R. Accademia delle Scienze di Torino* **34** (1898), 47–55] used binary notation as the basis of a "logical" character set of 256 symbols. Joseph Bowden [*Special Topics in Theoretical Arithmetic* (Garden City: 1936), 49] gave his own system of nomenclature for hexadecimal numbers.

The book *History of Binary and Other Nondecimal Numeration* by Anton Glaser (privately printed, 1971) contains an informative and nearly complete discussion of the development of binary notation, including English translations of many of the works cited above.

Much of the recent history of number systems is connected with the development of calculating machines. Charles Babbage's notebooks for 1838 show that he considered using nondecimal numbers in his Analytical Engine [cf. M. V. Wilkes, *Historia Math.* **4** (1977), 421]. Increased interest in mechanical devices for arithmetic, especially for multiplication, led several people in the 1930s to consider the binary system for this purpose. A particularly delightful account of such activity appears in the article "Binary Calculation" by E. William Phillips [*Journal of the Institute of Actuaries* **67** (1936), 187–221] together with a record of the discussion that followed a lecture he gave on the subject. Phillips began by saying, "The ultimate aim [of this paper] is to persuade the whole civilized world to abandon decimal numeration and to use octonal [i.e., radix 8] numeration in its place."

Modern readers of Phillips's article will perhaps be surprised to discover that a radix-8 number system was properly referred to as "octonary" or "octonal," according to all dictionaries of the English language at that time, just as the radix-10 number system is properly called either "denary" or "decimal"; the word "octal" did not appear in English language dictionaries until 1961, and it apparently originated as a term for the "base" of a certain class of vacuum tubes. The word "hexadecimal," which has crept into our language even more recently, is a mixture of Greek and Latin stems; more proper terms would be

"senidenary" or "sedecimal" or even "sexadecimal," but the latter is perhaps too risqué for computer programmers.

The comment by Mr. Wales that is quoted at the beginning of this chapter has been taken from the discussion printed with Phillips's paper. Another man who attended the same lecture objected to the octal system for business purposes: "5% becomes $3.\dot{1}46\dot{3}$ per 64, which sounds rather horrible."

Phillips got the inspiration for his proposals from an electronic circuit that was capable of counting in binary [C. E. Wynn-Williams, Proc. Roy. Soc. London **A136** (1932), 312–324]. Electromechanical and electronic circuitry for general arithmetic operations was developed during the late 1930s, notably by John V. Atanasoff and George R. Stibitz in the U.S.A., L. Couffignal and R. Valtat in France, Helmut Schreyer and Konrad Zuse in Germany. All of these inventors used the binary system, although Stibitz later developed excess-3 binary-coded-decimal notation. A fascinating account of these early developments, including reprints and translations of important contemporary documents, appears in Brian Randell's book *The Origins of Digital Computers* (Berlin: Springer, 1973).

The first American high-speed computers, built in the early 1940s, used decimal arithmetic. But in 1946, an important memorandum by A. W. Burks, H. H. Goldstine, and J. von Neumann, in connection with the design of the first stored-program computers, gave detailed reasons for the decision to make a radical departure from tradition and to use base-two notation [see John von Neumann, *Collected Works* **5**, 41–65]. Since then binary computers have multiplied. After a dozen years of experience with binary machines, a discussion of the relative advantages and disadvantages of radix-2 notation was given by W. Buchholz in his paper "Fingers or Fists?" [*CACM* **2** (December 1959), 3–11].

The MIX computer used in this book has been defined so that it can be either binary or decimal. It is interesting to note that nearly all MIX programs can be expressed without knowing whether binary or decimal notation is being used—even when we are doing calculations involving multiple-precision arithmetic. Thus we find that the choice of radix does not significantly influence computer programming. (Noteworthy exceptions to this statement, however, are the "Boolean" algorithms discussed in Section 7.1; see also Algorithm 4.5.2B.)

There are several different methods for representing *negative* numbers in a computer, and this sometimes influences the way arithmetic is done. In order to understand these other notations, let us first consider MIX as if it were a decimal computer; then each word contains 10 digits and a sign, for example

$$-12345\ 67890. \tag{2}$$

This is called the *signed-magnitude* representation. Such a representation agrees with common notational conventions, so it is preferred by many programmers. A potential disadvantage is that minus zero and plus zero can both be represented, while they usually should mean the same number; this possibility requires some care in practice, although it turns out to be useful at times.

Most mechanical calculators that do decimal arithmetic use another system called *ten's complement* notation. If we subtract 1 from 00000 00000, we get

99999 99999 in this notation; in other words, no explicit sign is attached to the number, and calculation is done *modulo* 10^{10}. The number —12345 67890 would appear as

$$87654\ 32110 \tag{3}$$

in ten's complement notation. It is conventional to regard any number whose leading digit is 5, 6, 7, 8, or 9 as a negative value in this notation, although with respect to addition and subtraction there is no harm in regarding (3) as the number +87654 32110 if it is convenient to do so. Note that there is no problem of "minus zero" in such a system.

The major difference between signed magnitude and ten's complement notations in practice is that shifting right does not divide the magnitude by ten; for example, the number $-11 = \ldots 99989$, shifted right one, gives $\ldots 99998 = -2$ (assuming that a shift to the right inserts "9" as the leading digit when the number shifted is negative). In general, x shifted right one digit in ten's complement notation will give $\lfloor x/10 \rfloor$, whether x is positive or negative.

A possible disadvantage of the ten's complement system is the fact that it is not symmetric about zero; the largest negative number representable in p digits is $500\ldots0$, and it is not the negative of any p-digit positive number. Thus it is possible that changing x to $-x$ will cause overflow. (See exercises 7 and 31 for a discussion of radix-complement notation with *infinite* precision.)

Another notation that has been used since the earliest days of high-speed computers is called *nines' complement* representation. In this case the number —12345 67890 would appear as

$$87654\ 32109. \tag{4}$$

Each digit of a negative number $(-x)$ is equal to 9 minus the corresponding digit of x. It is not difficult to see that the nines' complement notation for a negative number is always one less than the corresponding ten's complement notation. Addition and subtraction are done modulo $10^{10} - 1$, which means that a carry off the left end is to be added at the right end. (Cf. Section 3.2.1.1.) Again there is a potential problem with minus zero, since 99999 99999 and 00000 00000 denote the same value.

The ideas just explained for radix 10 arithmetic apply in a similar way to radix 2 arithmetic, where we have *signed magnitude*, *two's complement*, and *ones' complement* notations. The MIX computer, as used in the examples of this chapter, deals only with signed-magnitude arithmetic; however, alternative procedures for complement notations are discussed in the accompanying text when it is important to do so.

Most computer manuals tell us that the machine's circuitry assumes that the radix point is situated in a particular place within each computer word. This advice should usually be disregarded. It is better to learn the rules concerning where the radix point will appear in the result of an instruction if we assume that it lies in a certain place beforehand. For example, in the case of MIX we could regard our operands either as integers with the radix point at the extreme

right, or as fractions with the radix point at the extreme left, or as some mixture of these two extremes; the rules for the appearance of the radix point in each result are straightforward.

It is easy to see that there is a simple relation between radix b and radix b^k:

$$(\ldots a_3a_2a_1a_0.a_{-1}a_{-2}\ldots)_b = (\ldots A_3A_2A_1A_0.A_{-1}A_{-2}\ldots)_{b^k}, \tag{5}$$

where

$$A_j = (a_{kj+k-1}\ldots a_{kj+1}a_{kj})_b;$$

see exercise 8. Thus we have simple techniques for converting at sight between, say, binary and octal notation.

Many interesting variations on positional number systems are possible besides the standard b-ary systems discussed so far. For example, we might have numbers in base (-10), so that

$$\begin{aligned}(\ldots a_3a_2a_1a_0.a_{-1}a_{-2}\ldots)_{-10}&\\ &= \cdots + a_3(-10)^3 + a_2(-10)^2 + a_1(-10)^1 + a_0 + \cdots\\ &= \cdots - 1000a_3 + 100a_2 - 10a_1 + a_0 - \tfrac{1}{10}a_{-1} + \tfrac{1}{100}a_{-2} - \cdots.\end{aligned}$$

Here the individual digits satisfy $0 \le a_k \le 9$ just as in the decimal system. The number 12345 67890 appears in the "negadecimal" system as

$$(1\,93755\,73910)_{-10}, \tag{6}$$

since the latter represents 10305070900 — 9070503010. It is interesting to note that the negative of this number, $-12345\,67890$, would be written

$$(28466\,48290)_{-10}, \tag{7}$$

and, in fact, *every real number whether positive or negative can be represented without a sign* in the -10 system.

Negative-base systems were first considered by Vittorio Grünwald [*Giornale di matematiche di Battaglini* **23** (1885), 203–221, 367], who explained how to perform the four arithmetic operations in such systems; Grünwald also discussed root extraction, divisibility tests, and radix conversion. However, since his work was published in a rather obscure journal, it seems to have had no effect on other research, and it was soon forgotten. The next publication about negative-base systems was apparently by A. J. Kempner [*AMM* **43** (1936), 610–617], who discussed the properties of non-integer radices and remarked in a footnote that negative radices would be feasible too. After twenty more years the idea was rediscovered again, this time by Z. Pawlak and A. Wakulicz [*Bulletin de l'Academie Polonaise des Sciences*, Classe III, **5** (1957), 233–236; Série des sciences techniques **7** (1959), 713–721], and also by L. Wadel [*IRE Transactions* **EC-6**

(1957), 123]. For further references see *IEEE Transactions* **EC-12** (1963), 274–276; *Computer Design* **6** (May 1967), 52–63. There is evidence that the idea of negative bases occurred independently to quite a few people. For example, D. E. Knuth had discussed negative-base systems in 1955, together with a further generalization to complex-valued bases, in a short paper submitted to a "science talent search" contest for high-school seniors.

The base $2i$ gives rise to a system called the "quater-imaginary" number system (by analogy with "quaternary"), which has the unusual feature that *every complex number can be represented with the digits* 0, 1, 2, *and* 3 *without a sign.* [See D. E. Knuth, *CACM* **3** (1960), 245–247.] For example,

$$\begin{aligned}(11210.31)_{2i} &= 1\cdot 16 + 1\cdot(-8i) + 2\cdot(-4) + 1\cdot(2i) + 3\cdot(-\tfrac{1}{2}i) + 1(-\tfrac{1}{4})\\ &= 7\tfrac{3}{4} - 7\tfrac{1}{2}i.\end{aligned}$$

Here the number $(a_{2n}\ldots a_1a_0.a_{-1}\ldots a_{-2k})_{2i}$ is equal to

$$(a_{2n}\ldots a_2a_0.a_{-2}\ldots a_{-2k})_{-4} + 2i(a_{2n-1}\ldots a_3a_1.a_{-1}\ldots a_{-2k+1})_{-4},$$

so conversion to and from quater-imaginary notation reduces to conversion to and from negative quaternary representation of the real and imaginary parts. The interesting property of this system is that it allows multiplication and division of complex numbers to be done in a fairly unified manner without treating real and imaginary parts separately. For example, we can multiply two numbers in this system much as we do with any base, merely using a different "carry" rule: whenever a digit exceeds 3 we subtract 4 and "carry" -1 two columns to the left; when a digit is negative, we add 4 to it and "carry" $+1$ two columns to the left. A study of the following example shows this peculiar carry rule at work:

				1	2	2	3	1	$[9-10i]$
				1	2	2	3	1	$[9-10i]$
				1	2	2	3	1	
1	0	3	2	0	2	1	3		
		1	3	0	2	2			
	1	3	0	2	2				
1	2	2	3	1					
0	2	1	3	3	3	1	2	1	$[-19-180i]$

A similar system that uses just the digits 0 and 1 may be based on $\sqrt{2}\,i$, but this requires an infinite nonrepeating expansion for the simple number "i" itself. Vittorio Grünwald proposed using the digits 0 and $1/\sqrt{2}$ in odd-numbered positions, to avoid such a problem, but this actually spoils the whole system [cf. *Commentari dell' Ateneo di Brescia* (1886), 43–54].

Another "binary" complex number system may be obtained by using the base $i-1$, as suggested by W. Penney [*JACM* **12** (1965), 247–248]:

$$\begin{aligned}&(\ldots a_4a_3a_2a_1a_0.a_{-1}\ldots)_{i-1}\\ &\quad = \cdots - 4a_4 + (2{+}2i)a_3 - 2ia_2 + (i{-}1)a_1 + a_0 - \tfrac{1}{2}(i{+}1)a_{-1} + \cdots.\end{aligned}$$

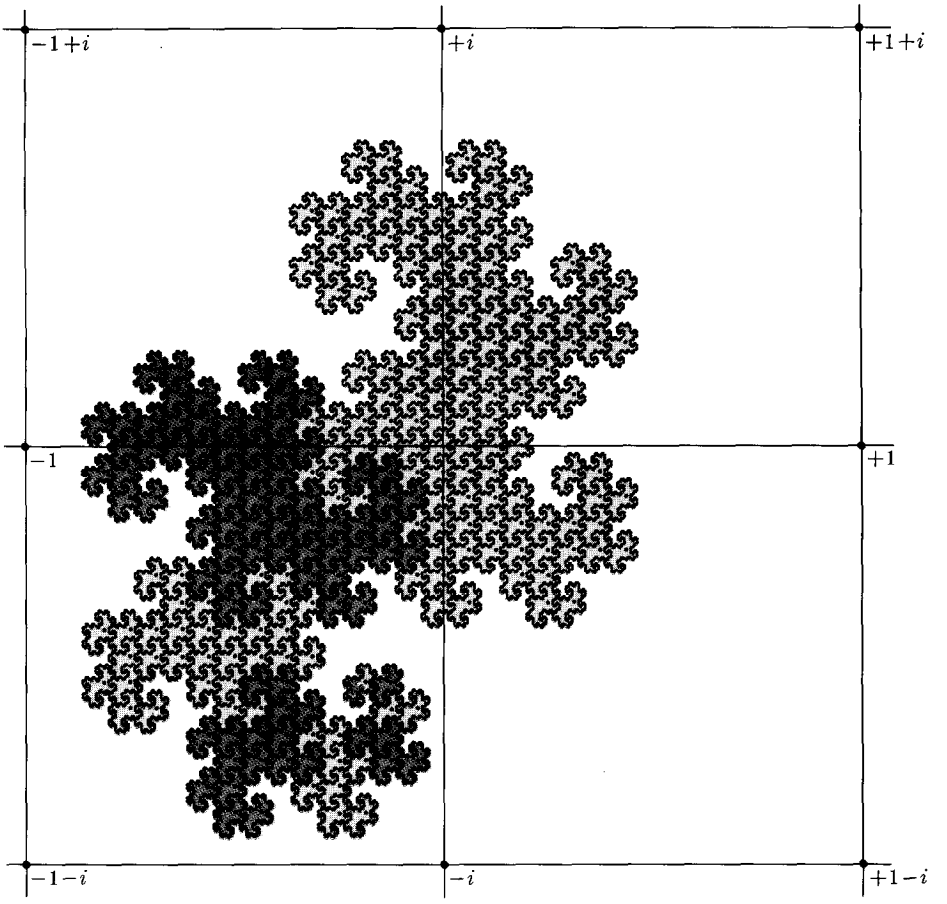


Fig. 1. The set S. (Illustration by P. M. Farmwald, R. W. Gosper, and R. E. Maas.)

In this system, only the digits 0 and 1 are needed. One way to demonstrate that every complex number has such a representation is to consider the interesting set S shown in Fig. 1; this set is, by definition, all points that can be written as $\sum_{k\geq 1} a_k(i-1)^{-k}$, for an infinite sequence $a_1, a_2, a_3, \ldots$ of zeros and ones. Figure 1 shows that S can be decomposed into 256 pieces congruent to $\frac{1}{16}S$; note that if the diagram of S is rotated counterclockwise by 135°, we obtain two adjacent sets congruent to $(1/\sqrt{2})\,S$ (since $(i-1)S = S \cup (S+1)$). For details of a proof that S contains all complex numbers that are of sufficiently small magnitude, see exercise 18.

Perhaps the prettiest number system of all is the *balanced ternary* notation, which consists of base-3 representation using -1, 0, and $+1$ as "trits" (ternary digits) instead of 0, 1, and 2. If we use the symbol $\bar{1}$ to stand for -1, we have the following examples of balanced ternary numbers:

Balanced ternary	Decimal
$1\,0\,\bar{1}$	8
$1\,1\,\bar{1}\,0.\bar{1}\,\bar{1}$	$32\frac{5}{9}$
$\bar{1}\,\bar{1}\,1\,0.1\,1$	$-32\frac{5}{9}$
$\bar{1}\,\bar{1}\,1\,0$	-33
$0.1\,1\,1\,1\,1\ldots$	$\frac{1}{2}$

One way to find the representation of a number in the balanced ternary system is to start by representing it in ternary notation; for example,

$$208.3 = (21201.022002200220\ldots)_3.$$

(A very simple pencil-and-paper method for converting to ternary notation is given in exercise 4.4–12.) Now add the infinite number $\ldots 11111.11111\ldots$ in ternary notation; we obtain, in the above example, the infinite number

$$(\ldots 11111210012.210121012101\ldots)_3.$$

Finally, subtract $\ldots 11111.11111\ldots$ by decrementing each digit; we get

$$208.3 = (10\bar{1}\bar{1}01.10\bar{1}010\bar{1}010\bar{1}0\ldots)_3. \tag{8}$$

This process may clearly be made rigorous if we replace the artificial infinite number $\ldots 11111.11111\ldots$ by a number with suitably many ones.

The balanced ternary number system has many pleasant properties:

a) The negative of a number is obtained by interchanging 1 and $\bar{1}$.

b) The sign of a number is given by its most significant nonzero "trit," and in general we can compare any two numbers by reading them from left to right and using lexicographic order, as in the decimal system.

c) The operation of rounding to the nearest integer is identical to truncation (i.e., deleting everything to the right of the radix point).

Addition in the balanced ternary system is quite simple, using the table

$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	0	1	1	1	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	0	1	1	1	$\bar{1}$	$\bar{1}$	$\bar{1}$	0	0	0	1	1	1
$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1	$\bar{1}$	0	1
$\bar{1}0$	$\bar{1}1$	$1\,\bar{1}$	$\bar{1}\,\bar{1}$	1	0	$\bar{1}$	0	1	$\bar{1}\,\bar{1}$	$\bar{1}$	0	$\bar{1}$	0	1	0	1	$1\bar{1}$	$\bar{1}$	0	1	0	$1\,\bar{1}$	$1\,\bar{1}$	1	$1\bar{1}$	10

(The three inputs to the addition are the digits of the numbers to be added and the carry digits.) Subtraction is negation followed by addition; and multiplication also reduces to negation and addition, as in the following example:

$$\begin{array}{rl}
 & 1\,\bar{1}\,0\,\bar{1} \quad [17] \\
 & 1\,\bar{1}\,0\,\bar{1} \quad [17] \\
 & \overline{\bar{1}\,1\,0\,1} \\
 \bar{1}\,1\,0\,1\,0 & \\
1\,\bar{1}\,0\,\bar{1} & \\
\overline{0\,1\,1\,\bar{1}\,\bar{1}\,0\,1} & \quad [289]
\end{array}$$

Representation of numbers in the balanced ternary system is implicitly present in a famous mathematical puzzle, which is commonly called "Bachet's problem of weights" although it was already stated by Fibonacci four centuries before Bachet wrote his book. [See W. Ahrens, *Mathematische Unterhaltungen und Spiele* **1** (Leipzig: Teubner, 1910), Section 3.4.] Positional number systems with negative digits have apparently been known for more than 1000 years in India; see J. Bharati, *Vedic Mathematics* (Delhi: Motilal Banarsidass, 1965). They were independently rediscovered by J. Colson [*Philos. Trans.* **34** (1726), 161–173], and by Sir John Leslie [*The Philosophy of Arithmetic* (Edinburgh, 1817); see pp. 33–34, 54, 64–65, 117, 150]; and also by A. Cauchy [*Comptes Rendus* **11** (Paris: 1840), 789–798], who pointed out that negative digits make it unnecessary for a person to memorize the multiplication table past 5×5. The first true appearance of "pure" balanced ternary notation was in an article by Léon Lalanne [*Comptes Rendus* **11** (Paris: 1840), 903–905], who was a designer of mechanical devices for arithmetic. The system was mentioned only rarely for 100 years after Lalanne's paper, until the development of the first electronic computers at the Moore School of Electrical Engineering in 1945–1946; at that time it was given serious consideration along with the binary system as a possible replacement for the decimal system. The complexity of arithmetic circuitry for balanced ternary arithmetic is not much greater than it is for the binary system, and a given number requires only $\ln 2 / \ln 3 \approx 63\%$ as many digit positions for its representation. Discussions of the balanced ternary number system appear in *AMM* **57** (1950), 90–93, and in *High-speed Computing Devices*, Engineering Research Associates (McGraw-Hill, 1950), 287–289. The experimental Russian computer SETUN was based on balanced ternary notation [see *CACM* **3** (1960), 149–150], and perhaps the symmetric properties and simple arithmetic of this number system will prove to be quite important some day—when the "flip-flop" is replaced by a "flip-flap-flop".

Positional notation generalizes in another important way to a *mixed-radix* system. Given a sequence of numbers $\langle b_n \rangle$ (where n may be negative), we define

$$\begin{bmatrix} \cdots, a_3, a_2, a_1, a_0; a_{-1}, a_{-2}, \cdots \\ \cdots, b_3, b_2, b_1, b_0; b_{-1}, b_{-2}, \cdots \end{bmatrix} \tag{9}$$
$$= \cdots + a_3 b_2 b_1 b_0 + a_2 b_1 b_0 + a_1 b_0 + a_0 + a_{-1}/b_{-1} + a_{-2}/b_{-1} b_{-2} + \cdots .$$

In the simplest mixed-radix systems, we work only with integers; we let b_0, b_1, b_2, ... be integers greater than one, and deal only with numbers that have no radix point, where a_n is required to lie in the range $0 \le a_n < b_n$.

One of the most important mixed-radix systems is the *factorial number system*, where $b_n = n + 2$. Using this system, we can represent every positive integer uniquely in the form

$$c_n\, n! + c_{n-1}\, (n-1)! + \cdots + c_2\, 2! + c_1, \tag{10}$$

where $0 \le c_k \le k$ for $1 \le k \le n$, and $c_n \ne 0$. (See Algorithm 3.3.2P.)

Mixed-radix systems are familiar in everyday life, when we deal with units of measure. For example, the quantity "3 weeks, 2 days, 9 hours, 22 minutes, 57 seconds, and 492 milliseconds" is equal to

$$\begin{bmatrix} 3, & 2, & 9, & 22, & 57; & 492 \\ & 7, & 24, & 60, & 60; & 1000 \end{bmatrix} \text{ seconds.}$$

The quantity "10 pounds, 6 shillings, and thruppence ha'penny" was once equal to $\begin{bmatrix} 10, & 6, & 3; & 1 \\ & 20, & 12; & 2 \end{bmatrix}$ pence in British currency, before Great Britain changed to a purely decimal monetary system.

It is possible to add and subtract mixed-radix numbers by using a straightforward generalization of the usual addition and subtraction algorithms, provided of course that the same mixed-radix system is being used for both operands (see exercise 4.3.1–9). Similarly, we can easily multiply or divide a mixed-radix number by small integer constants, using simple extensions of the familiar pencil-and-paper methods.

Mixed-radix systems were first discussed in full generality by Georg Cantor [*Zeitschrift für Math. und Physik* **14** (1869), 121–128]. Exercises 26 and 29 give further information about them.

Some questions concerning *irrational* radices have been investigated by W. Parry, *Acta Mathematica*, Acad. Sci. Hung., **11** (1960), 401–416.

Besides the systems described in this section, several other ways to represent numbers are mentioned elsewhere in this series of books: the binomial number system (exercise 1.2.6–56); the Fibonacci number system (exercises 1.2.8–34, 5.4.2–10); the phi number system (exercise 1.2.8–35); modular representations (Section 4.3.2); Gray code (Section 7.2.1); and roman numerals (Section 9.1).

EXERCISES

1. [*15*] Express $-10, -9, \ldots, 9, 10$ in the number system whose base is -2.

▶ **2.** [*24*] Consider the following four number systems: (a) binary (signed magnitude); (b) negabinary (radix -2); (c) balanced ternary; and (d) radix $b = \frac{1}{10}$. Use each of these four number systems to express each of the following three numbers: (i) -49; (ii) $-3\frac{1}{7}$ (show the repeating cycle); (iii) π (to a few significant figures).

3. [*20*] Express $-49 + i$ in the quater-imaginary system.

4. [*15*] Assume that we have a MIX program in which location A contains a number for which the radix point lies between bytes 3 and 4, while location B contains a number whose radix point lies between bytes 2 and 3. (The leftmost byte is number 1). Where will the radix point be, in registers A and X, after the following instructions?

a)
```
LDA  A
MUL  B
```

b)
```
LDA  A
SRAX 5
DIV  B
```

5. [*00*] Explain why a negative integer in nines' complement notation has a representation in ten's complement notation that is always one greater, if the representations are regarded as positive.

6. [*16*] What are the largest and smallest p-bit integers that can be represented in (a) signed-magnitude binary notation (including one bit for the sign), (b) two's complement notation, (c) ones' complement notation?

7. [*M20*] The text defines ten's complement notation only for integers represented in a single computer word. Is there a way to define a ten's complement notation *for all real numbers*, having "infinite precision," analogous to the text's definition? Is there a similar way to define a nines' complement notation for all real numbers?

8. [*M10*] Prove Eq. (5).

▶ **9.** [*15*] Change the following *octal* numbers to *hexadecimal* notation, using the hexadecimal digits 0, 1, ..., F: *12*; *5655*; *2550276*; *76545336*; *3726755*.

10. [*M22*] Generalize Eq. (5) to mixed-radix notation.

11. [*22*] Design an algorithm that uses the -2 number system to compute the sum of $(a_n \ldots a_1 a_0)_{-2}$ and $(b_n \ldots b_1 b_0)_{-2}$, obtaining the answer $(c_{n+2} \ldots c_1 c_0)_{-2}$.

12. [*23*] Specify algorithms that convert (a) the binary signed magnitude number $\pm(a_n \ldots a_0)_2$ to its negabinary form $(b_{n+1} \ldots b_0)_{-2}$; and (b) the negabinary number $(b_{n+1} \ldots b_0)_{-2}$ to its signed magnitude form $\pm(a_{n+1} \ldots a_0)_2$.

▶ **13.** [*M21*] In the decimal system there are some numbers with two infinite decimal expansions; e.g., $2.3599999\ldots = 2.3600000\ldots$. Does the *negadecimal* (base -10) system have unique expansions, or are there real numbers with two different infinite expansions in this base also?

14. [*14*] Multiply $(11321)_{2i}$ by itself in the quater-imaginary system using the method illustrated in the text.

15. [*M24*] What are the sets

$$S = \left\{ \sum_{k \geq 1} a_k b^{-k} \;\middle|\; a_k \text{ an allowable digit} \right\},$$

analogous to Fig. 1, for the negative decimal and for the quater-imaginary number systems?

16. [*M24*] Design an algorithm to add 1 to $(a_n \ldots a_1 a_0)_{i-1}$ in the $i-1$ number system.

17. [*M30*] It may seem peculiar that $i - 1$ has been suggested as a number-system base, instead of the similar but intuitively simpler number $i + 1$. Can every complex number $a+bi$, where a and b are integers, be represented in a positional number system to base $i + 1$, using only the digits 0 and 1?

18. [*HM32*] Show that the set S of Fig. 1 is a closed set that contains a neighborhood of the origin. (Consequently, every complex number has a "binary" representation to base $i - 1$.)

▶ **19.** [*23*] (David W. Matula.) Let D be a set of b integers, containing exactly one solution to the congruence $x \equiv j$ (modulo b) for $0 \leq j < b$. Prove that all integers m (positive, negative, or zero) can be represented in the form $m = (a_n \ldots a_0)_b$, where all the a_j are in D, if and only if all integers in the range $l \leq m \leq u$ can be so represented, where $l = -\max\{\, a \mid a \in D \,\}/(b-1)$, $u = -\min\{\, a \mid a \in D \,\}/(b-1)$. For example, $D = \{-1, 0, \ldots, b-2\}$ satisfies the conditions for all $b \geq 3$. [*Hint:* Design an algorithm that constructs a suitable representation.]

20. [*HM28*] (David W. Matula.) Consider a decimal number system that uses the digits $D = \{-1, 0, 8, 17, 26, 35, 44, 53, 62, 71\}$ instead of $\{0, 1, \ldots, 9\}$. The result of exercise 19 implies (as in exercise 18) that all real numbers have an infinite decimal expansion using digits from D.

In the usual decimal system, exercise 13 points out that some numbers have two representations. (a) Find a real number that has *more* than two D-decimal representations. (b) Show that no real number has infinitely many D-decimal representations. (c) Show that uncountably many numbers have two or more D-decimal representations.

▶ **21.** [*M22*] (C. E. Shannon.) Can every real number (positive, negative, or zero) be expressed in a "balanced decimal" system, i.e., in the form $\sum_{k\le n} a_k 10^k$, for some integer n and some sequence a_n, a_{n-1}, a_{n-2}, ..., where each a_k is one of the ten numbers $\{-4\frac{1}{2}, -3\frac{1}{2}, -2\frac{1}{2}, -1\frac{1}{2}, -\frac{1}{2}, \frac{1}{2}, 1\frac{1}{2}, 2\frac{1}{2}, 3\frac{1}{2}, 4\frac{1}{2}\}$? (Note that zero is not one of the allowed digits, but we implicitly assume that a_{n+1}, a_{n+2}, ... are zero.) Find all representations of zero in this number system, and find all representations of unity.

22. [*HM25*] Let $\alpha = -\sum_{m\ge 1} 10^{-m^2}$. Given $\epsilon > 0$ and any real number x, prove that there is a "decimal" representation such that $0 < \left|x - \sum_{0\le k\le n} a_k 10^k\right| < \epsilon$, where each a_k is allowed to be only one of the three values 0, 1, or α. (Note that no negative powers of 10 are used in this representation!)

23. [*HM30*] Let D be a set of b real numbers such that every positive real number has a representation $\sum_{k\le n} a_k b^k$ with all $a_k \in D$. Exercise 20 shows that there may be many numbers without *unique* representations; but prove that the set T of all such numbers has measure zero.

24. [*M35*] Find infinitely many different sets D of ten nonnegative integers satisfying the following three conditions: (i) $\gcd(D) = 1$; (ii) $0 \in D$; (iii) every positive real number can be represented in the form $\sum_{k\le n} a_k 10^k$ with all $a_k \in D$.

25. [*M25*] (S. A. Cook.) Let b, u, and v be positive integers, where $b \ge 2$ and $0 < v < b^m$. Show that the base b representation of u/v does not contain a run of m consecutive digits equal to $b - 1$, anywhere to the right of the radix point. (By convention, no runs of infinitely many $(b - 1)$'s are permitted in the standard base b representation.)

▶ **26.** [*HM30*] (N. S. Mendelsohn.) Let $\langle\beta_n\rangle$ be a sequence of real numbers defined for all integers n, $-\infty < n < \infty$, such that

$$\beta_n < \beta_{n+1}; \qquad \lim_{n\to\infty} \beta_n = \infty; \qquad \lim_{n\to-\infty} \beta_n = 0.$$

Let $\langle c_n\rangle$ be an arbitrary sequence of positive integers that is defined for all integers n, $-\infty < n < \infty$. Let us say that a number x has a "generalized representation" if there is an integer n and an infinite sequence of integers a_n, a_{n-1}, a_{n-2}, ... such that $x = \sum_{k\le n} a_k\beta_k$, where $a_n \neq 0$, $0 \le a_k \le c_k$, and $a_k < c_k$ for infinitely many k.

Show that every positive real number x has exactly one generalized representation if and only if $\beta_{n+1} = \sum_{k\le n} c_k\beta_k$ for all n. (Consequently, the mixed-radix systems with integer bases have this property; and mixed-radix systems with $\beta_1 = (c_0 + 1)\beta_0$, $\beta_2 = (c_1 + 1)(c_0 + 1)\beta_0$, ..., $\beta_{-1} = \beta_0/(c_{-1} + 1)$, ... are the most general number systems of this type.)

27. [*M21*] Show that every nonzero integer has a unique "reversing binary representation"

$$2^{e_0} - 2^{e_1} + \cdots + (-1)^t 2^{e_t},$$

where $e_0 < e_1 < \cdots < e_t$.

▸ **28.** [*M24*] Show that every nonzero complex number of the form $a + bi$ where a and b are integers has a unique "revolving binary representation"

$$(1+i)^{e_0} + i(1+i)^{e_1} - (1+i)^{e_2} - i(1+i)^{e_3} + \cdots + i^t(1+i)^{e_t},$$

where $e_0 < e_1 < \cdots < e_t$. (Cf. exercise 27.)

29. [*M35*] (N. G. de Bruijn.) Let S_0, S_1, S_2, ... be sets of nonnegative integers; we will say that the collection $\{S_0, S_1, S_2, \ldots\}$ has Property B if every nonnegative integer n can be written in the form

$$n = s_0 + s_1 + s_2 + \cdots, \qquad s_j \in S_j,$$

in exactly one way. (Property B implies that $0 \in S_j$ for all j, since $n = 0$ can only be represented as $0 + 0 + 0 + \cdots$.) Any mixed-radix number system with radices b_0, b_1, b_2, ... provides an example of a collection of sets satisfying Property B, if we let $S_j = \{0, B_j, \ldots, (b_j - 1)B_j\}$, where $B_j = b_0 b_1 \ldots b_{j-1}$; here the representation of $n = s_0 + s_1 + s_2 + \cdots$ corresponds in an obvious manner to its mixed-radix representation (9). Furthermore, if the collection $\{S_0, S_1, S_2, \ldots\}$ has Property B, and if A_0, A_1, A_2, ... is any partition of the nonnegative integers (so that we have $A_0 \cup A_1 \cup A_2 \cup \cdots = \{0, 1, 2, \ldots\}$ and $A_i \cap A_j = \emptyset$ for $i \neq j$; some A_j's may be empty), then the "collapsed" collection $\{T_0, T_1, T_2, \ldots\}$ also has Property B, where T_j is the set of all sums $\sum_{i \in A_j} s_i$ taken over all possible choices of $s_i \in S_i$.

Prove that *any* collection $\{T_0, T_1, T_2, \ldots\}$ that satisfies Property B may be obtained by collapsing some collection $\{S_0, S_1, S_2, \ldots\}$ that corresponds to a mixed-radix number system.

30. [*M39*] (N. G. de Bruijn.) The radix-(−2) number system shows us that every integer (positive, negative, or zero) has a unique representation of the form

$$(-2)^{e_1} + (-2)^{e_2} + \cdots + (-2)^{e_t}, \qquad e_1 > e_2 > \cdots > e_t \geq 0, \qquad t \geq 0.$$

The purpose of this exercise is to explore generalizations of this phenomenon.

a) Let b_0, b_1, b_2, ... be a sequence of integers such that every integer n has a unique representation of the form

$$n = b_{e_1} + b_{e_2} + \cdots + b_{e_t}, \qquad e_1 > e_2 > \cdots > e_t \geq 0, \qquad t \geq 0.$$

(Such a sequence $\langle b_n \rangle$ is called a "binary basis.") Show that there is an index j such that b_j is odd, but b_k is even for all $k \neq j$.

b) Prove that a binary basis $\langle b_n \rangle$ can always be rearranged into the form d_0, $2d_1$, $4d_2$, ... $= \langle 2^n d_n \rangle$, where each d_k is odd.

c) If each of d_0, d_1, d_2, ... in (b) is ± 1, prove that $\langle b_n \rangle$ is a binary basis if and only if there are infinitely many $+1$'s and infinitely many -1's.

d) Prove that 7, $-13 \cdot 2$, $7 \cdot 2^2$, $-13 \cdot 2^3$, ..., $7 \cdot 2^{2k}$, $-13 \cdot 2^{2k+1}$, ... is a binary basis, and find the representation of $n = 1$.

▶ **31.** [*M35*] A generalization of two's complement arithmetic, called "2-adic numbers," was invented about 1900 by K. Hensel. (In fact he treated *p-adic numbers*, for any prime p.) A 2-adic number may be regarded as a binary number

$$u = (\ldots u_3 u_2 u_1 u_0 . u_{-1} \ldots u_{-n})_2,$$

whose representation extends infinitely far to the left, but only finitely many places to the right, of the binary point. Addition, subtraction, and multiplication of 2-adic numbers are done according to the ordinary procedures of arithmetic, which can in principle be extended indefinitely to the left. For example,

$$\begin{aligned} 7 &= (\ldots 000000000000111)_2 & \tfrac{1}{7} &= (\ldots 110110110110111)_2 \\ -7 &= (\ldots 111111111111001)_2 & -\tfrac{1}{7} &= (\ldots 001001001001001)_2 \\ \tfrac{7}{4} &= (\ldots 000000000000001.11)_2 & \tfrac{1}{10} &= (\ldots 110011001100110.1)_2 \\ \sqrt{-7} &= (\ldots 100000010110101)_2 \quad \text{or} \quad (\ldots 011111101001011)_2. \end{aligned}$$

Here 7 appears as the ordinary binary integer seven, while -7 is its two's complement (extending infinitely to the left); it is easy to verify that the ordinary procedure for addition of binary numbers will give $-7 + 7 = (\ldots 00000)_2 = 0$, when the procedure is continued indefinitely. The values of $\frac{1}{7}$ and $-\frac{1}{7}$ are the unique 2-adic numbers that, when formally multiplied by 7, give 1 and -1, respectively. The values of $\frac{7}{4}$ and $\frac{1}{10}$ are examples of 2-adic numbers that are not 2-adic "integers," since they have nonzero bits to the right of the binary point. The two values of $\sqrt{-7}$, which are negatives of each other, are the only 2-adic numbers that, when formally squared, yield the value $(\ldots 111111111111001)_2$.

a) Prove that any 2-adic number u can be divided by any nonzero 2-adic number v to obtain a unique 2-adic number w satisfying $u = vw$. (Hence the set of 2-adic numbers forms a "field"; cf. Section 4.6.1.)

b) Prove that the 2-adic representation of the rational number $-1/(2n+1)$ may be obtained as follows, when n is a positive integer: First find the ordinary binary expansion of $+1/(2n+1)$, which has the periodic form $(0.\alpha\alpha\alpha\ldots)_2$ for some string α of 0's and 1's. Then $-1/(2n+1)$ is the 2-adic number $(\ldots\alpha\alpha\alpha)_2$.

c) Prove that the representation of a 2-adic number u is ultimately periodic (that is, $u_{N+\lambda} = u_N$ for all large N, for some $\lambda \geq 1$) if and only if u is rational (that is, $u = m/n$, for some integers m and n).

d) Prove that, when n is an integer, $\sqrt{n}$ is a 2-adic number if and only if it satisfies $n \bmod 2^{2k+3} = 2^{2k}$ for some nonnegative integer k. (Thus, the possibilities are either $n \bmod 8 = 1$, or $n \bmod 32 = 4$, etc.)

32. [*M40*] (I. Z. Ruzsa.) Prove that there are infinitely many integers whose ternary representation uses only 0's and 1's and whose quinary representation uses only 0's, 1's, and 2's.

33. [*M40*] (D. A. Klarner.) Let D be any set of integers, let b be any positive integer, and let k_n be the number of distinct integers that can be written as n-digit numbers $(a_{n-1}\ldots a_1 a_0)_b$ to base b with digits a_i in D. Prove that the sequence $\langle k_n \rangle$ satisfies a linear recurrence relation, and explain how to compute the generating function $\sum_n k_n z^n$. Illustrate your algorithm in the case $b = 3$ and $D = \{-1, 0, 3\}$.

4.2. FLOATING POINT ARITHMETIC

IN THIS SECTION, we shall study the basic principles of doing arithmetic on "floating point" numbers, by analyzing the internal mechanisms underlying such calculations. Perhaps many readers will have little interest in this subject, since their computers either have built-in floating point instructions or their computer manufacturer has supplied suitable subroutines. But, in fact, the material of this section should not merely be the concern of computer-design engineers or of a small clique of people who write library subroutines for new machines; every well-rounded programmer ought to have a knowledge of what goes on during the elementary steps of floating point arithmetic. This subject is not at all as trivial as most people think; it involves a surprising amount of interesting information.

4.2.1. Single-Precision Calculations

A. Floating point notation. We have discussed "fixed point" notation for numbers in Section 4.1; in such a case the programmer knows where the radix point is assumed to lie in the numbers he manipulates. For many purposes it is considerably more convenient to let the position of the radix point be dynamically variable or "floating" as a program is running, and to carry with each number an indication of its current radix point position. This idea has been used for many years in scientific calculations, especially for expressing very large numbers like Avogadro's number $N = 6.02252 \times 10^{23}$, or very small numbers like Planck's constant $\hbar = 1.0545 \times 10^{-27}$ erg sec.

In this section we shall work with *base b, excess q, floating point numbers with p digits*: Such numbers will be represented by pairs of values (e, f), denoting

$$(e, f) = f \times b^{e-q}. \tag{1}$$

Here e is an integer having a specified range, and f is a signed fraction. We will adopt the convention that

$$|f| < 1;$$

in other words, the radix point appears at the left of the positional representation of f. More precisely, the stipulation that we have p-digit numbers means that $b^p f$ is an integer, and that

$$-b^p < b^p f < b^p. \tag{2}$$

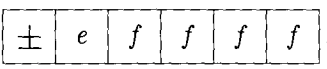
The term "floating binary" implies that $b = 2$, "floating decimal" implies $b = 10$, etc. Using excess-50 floating decimal numbers with 8 digits, we can write, for example,

$$\begin{aligned} &\text{Avogadro's number} \quad N = (74, +.60225200); \\ &\text{Planck's constant} \quad \hbar = (24, +.10545000). \end{aligned} \tag{3}$$

The two components e and f of a floating point number are called the exponent and the *fraction* parts, respectively. (Other names are occasionally

used for this purpose, notably "characteristic" and "mantissa"; but it is an abuse of terminology to call the fraction part a mantissa, since this concept has quite a different meaning in connection with logarithms. Furthermore the English word mantissa means "a worthless addition.")

The MIX computer assumes that its floating point numbers have the form



(4)

Here we have base b, excess q, floating point notation with four bytes of precision, where b is the byte size (e.g., $b = 64$ or $b = 100$), and q is equal to $\lfloor \frac{1}{2}b \rfloor$. The fraction part is $\pm f\,f\,f\,f$, and e is the exponent, which lies in the range $0 \le e < b$. This internal representation is typical of the conventions in most existing computers, although b is a much larger base than usual.

B. Normalized calculations. A floating point number (e, f) is *normalized* if the most significant digit of the representation of f is nonzero, so that

$$1/b \le |f| < 1; \tag{5}$$

or if $f = 0$ and e has its smallest possible value. It is possible to tell which of two normalized floating point numbers has a greater magnitude by comparing the exponent parts first, and then testing the fraction parts only if the exponents are equal.

Most floating point routines now in use deal almost entirely with normalized numbers: inputs to the routines are assumed to be normalized, and the outputs are always normalized. Under these conventions we lose the ability to represent a few numbers of very small magnitude—for example, the value $(0, .00000001)$ can't be normalized without producing a negative exponent—but we gain in speed, uniformity, and the ability to give relatively simple bounds on the relative error in our computations. (Unnormalized floating point arithmetic is discussed in Section 4.2.2.)

Let us now study the normalized floating point operations in detail. At the same time we can consider the construction of subroutines for these operations, assuming that we have a computer without built-in floating point hardware.

Machine-language subroutines for floating point arithmetic are usually written in a very machine-dependent manner, using many of the wildest idiosyncrasies of the computer at hand; so floating point addition subroutines for two different machines usually bear little superficial resemblance to each other. Yet a careful study of numerous subroutines for both binary and decimal computers reveals that these programs actually have quite a lot in common, and it is possible to discuss the topics in a machine-independent way.

The first (and by far the most difficult!) algorithm we shall discuss in this section is a procedure for floating point addition,

$$(e_u, f_u) \oplus (e_v, f_v) = (e_w, f_w). \tag{6}$$

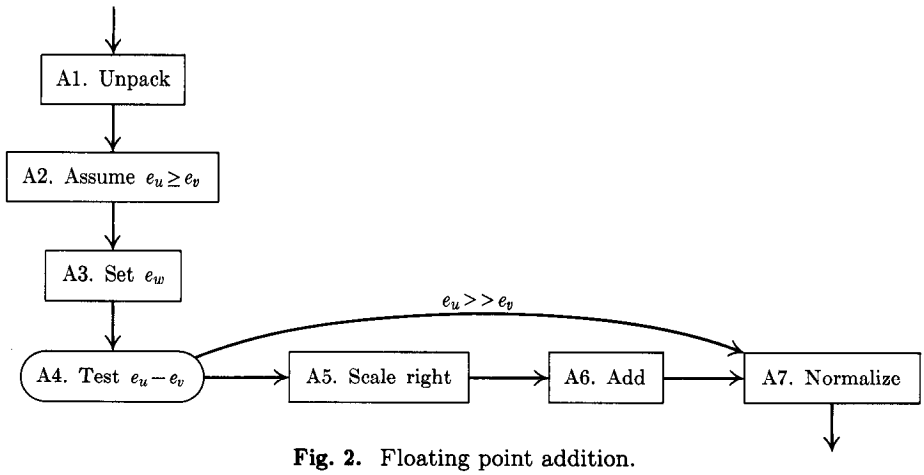


Fig. 2. Floating point addition.

Note: Since floating point arithmetic is inherently approximate, not exact, we will use "round" symbols

$$\oplus, \quad \ominus, \quad \otimes, \quad \oslash$$

to stand for floating point addition, subtraction, multiplication, and division, respectively, in order to distinguish approximate operations from the true ones.

The basic idea involved in floating point addition is fairly simple: Assuming that $e_u \geq e_v$, we take $e_w = e_u$, $f_w = f_u + f_v/b^{e_u - e_v}$ (thereby aligning the radix points for a meaningful addition), and normalize the result. Several situations can arise that make this process nontrivial, and the following algorithm explains the method more precisely.

Algorithm A (*Floating point addition*). Given base b, excess q, p-digit, normalized floating point numbers $u = (e_u, f_u)$ and $v = (e_v, f_v)$, this algorithm forms the sum $w = u \oplus v$. The same procedure may be used for floating point subtraction, if $-v$ is substituted for v.

A1. [Unpack.] Separate the exponent and fraction parts of the representations of u and v.

A2. [Assume $e_u \geq e_v$.] If $e_u < e_v$, interchange u and v. (In many cases, it is best to combine step A2 with step A1 or with some of the later steps.)

A3. [Set e_w.] Set $e_w \leftarrow e_u$.

A4. [Test $e_u - e_v$.] If $e_u - e_v \geq p + 2$ (large difference in exponents), set $f_w \leftarrow f_u$ and go to step A7. (Actually, since we are assuming that u is normalized, we could terminate the algorithm; but it is occasionally useful to be able to normalize a possibly unnormalized number by adding zero to it.)

A5. [Scale right.] Shift f_v to the right $e_u - e_v$ places; i.e., divide it by $b^{e_u - e_v}$. [*Note:* This will be a shift of up to $p + 1$ places, and the next step (which adds f_u to f_v) thereby requires an accumulator capable of holding $2p + 1$

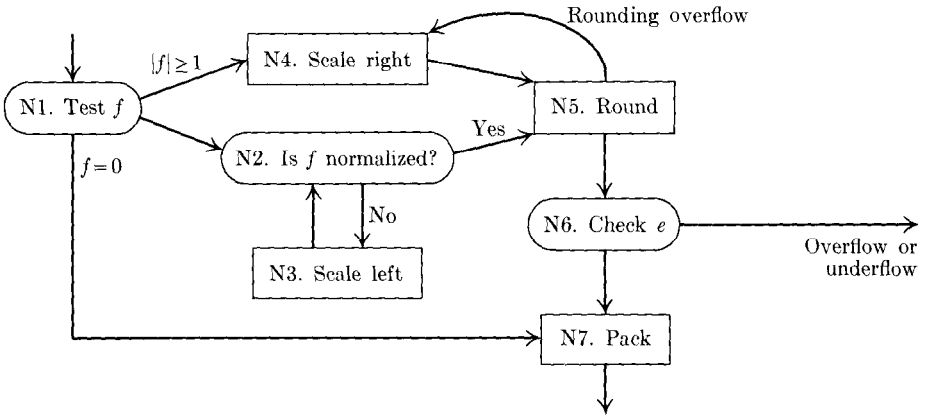


Fig. 3. Normalization of (e, f).

base-b digits to the right of the radix point. If such a large accumulator is not available, it is possible to shorten the requirement to $p + 2$ or $p + 3$ places if proper precautions are taken; the details are given in exercise 5.]

A6. [Add.] Set $f_w \leftarrow f_u + f_v$.

A7. [Normalize.] (At this point (e_w, f_w) represents the sum of u and v, but $|f_w|$ may have more than p digits, and it may be greater than unity or less than $1/b$.) Perform Algorithm N below, to normalize and round (e_w, f_w) into the final answer. ∎

Algorithm N (*Normalization*). A "raw exponent" e and a "raw fraction" f are converted to normalized form, rounding if necessary to p digits. This algorithm assumes that $|f| < b$.

N1. [Test f.] If $|f| \geq 1$ ("fraction overflow"), go to step N4. If $f = 0$, set e to its lowest possible value and go to step N7.

N2. [Is f normalized?] If $|f| \geq 1/b$, go to step N5.

N3. [Scale left.] Shift f to the left by one digit position (i.e., multiply it by b), and decrease e by 1. Return to step N2.

N4. [Scale right.] Shift f to the right by one digit position (i.e., divide it by b), and increase e by 1.

N5. [Round.] Round f to p places. (We take this to mean that f is changed to the nearest multiple of b^{-p}. It is possible that $(b^p f) \bmod 1 = \frac{1}{2}$ so that there are *two* nearest multiples; if b is even, we choose the one that makes $b^p f + \frac{1}{2}b$ odd. Further discussion of rounding appears in Section 4.2.2.) It is important to note that this rounding operation can make $|f| = 1$ ("rounding overflow"); in such a case, return to step N4.

N6. [Check e.] If e is too large, i.e., larger than its allowed range, an *exponent overflow* condition is sensed. If e is too small, an *exponent underflow* condition is sensed. (See the discussion below; since the result cannot

be expressed as a normalized floating point number in the required range, special action is necessary.)

N7. [Pack.] Put e and f together into the desired output representation. ∎

Some simple examples of floating point addition are given in exercise 4.

The following MIX subroutines, for addition and subtraction of numbers having the form (4), show how Algorithms A and N can be expressed as computer programs. The subroutines below are designed to take one input u from symbolic location ACC, and the other input v comes from register A upon entrance to the subroutine. The output w appears both in register A and location ACC. Thus, a fixed point coding sequence

```
LDA A; ADD B; SUB C; STA D                    (7)
```

would correspond to the floating point coding sequence

```
LDA A, STA ACC; LDA B, JMP FADD; LDA C, JMP FSUB; STA D.    (8)
```

Program A (*Addition, subtraction, and normalization*). The following program is a subroutine for Algorithm A, and it is also designed so that the normalization portion can be used by other subroutines that appear later in this section. In this program and in many others throughout this chapter, OFLO stands for a subroutine that prints out a message to the effect that MIX's overflow toggle was unexpectedly found to be "on." The byte size b is assumed to be a multiple of 4. The normalization routine NORM assumes that rI2 $= e$ and rAX $= f$, where rA $= 0$ implies rX $= 0$ and rI2 $< b$.

```
00  BYTE  EQU  1(4:4)        Byte size b
01  EXP   EQU  1:1           Definition of exponent field
02  FSUB  STA  TEMP          Floating point subtraction subroutine:
03        LDAN TEMP          Change sign of operand.
04  FADD  STJ  EXITF         Floating point addition subroutine:
05        JOV  OFLO          Ensure overflow is off.
06        STA  TEMP          TEMP ← v.
07        LDX  ACC           rX ← u.
08        CMPA ACC(EXP)      Steps A1, A2, A3 are combined here:
09        JGE  1F            Jump if e_v ≥ e_u.
10        STX  FU(0:4)       FU ← ±fffff0.
11        LD2  ACC(EXP)      rI2 ← e_w.
12        STA  FV(0:4)
13        LD1N TEMP(EXP)     rI1 ← −e_v.
14        JMP  4F
15  1H    STA  FU(0:4)       FU ← ±fffff0 (u, v interchanged).
16        LD2  TEMP(EXP)     rI2 ← e_w.
17        STX  FV(0:4)
18        LD1N ACC(EXP)      rI1 ← −e_v.
```

```
19  4H      INC1  0,2          rI1 ← e_u − e_v.  (Step A4 unnecessary.)
20  5H      LDA   FV           A5. Scale right.
21          ENTX  0            Clear rX.
22          SRAX  0,1          Shift right e_u − e_v places.
23  6H      ADD   FU           A6. Add.
24          JOV   N4           A7. Normalize. Jump if fraction overflow.
25          JXZ   NORM         Easy case?
26          CMPA  =0=(1:1)     Is f normalized?
27          JNE   N5           If so, round it.
28          SRC   5            |rX| ↔ |rA|.
29          DECX  1            (rX is positive.)
30          STA   TEMP         (Operands had opposite signs,
31          STA   HALF(0:0)      registers must be adjusted
32          LDAN  TEMP           before rounding and normalization.)
33          ADD   HALF
34          ADD   HALF         Complement least significant portion.
35          SRC   4            Jump into normalization routine.
36          JMP   N3A
37  HALF    CON   1//2         One half the word size (Sign varies)
38  FU      CON   0            Fraction part f_u
39  FV      CON   0            Fraction part f_v
40  NORM    JAZ   ZRO          N1. Test f.
41  N2      CMPA  =0=(1:1)     N2. Is f normalized?
42          JNE   N5           To N5 if leading byte nonzero.
43  N3      SLAX  1            N3. Scale left.
44  N3A     DEC2  1            Decrease e by 1.
45          JMP   N2           Return to N2.
46  N4      ENTX  1            N4. Scale right.
47          SRC   1            Shift right, insert "1" with proper sign.
48          INC2  1            Increase e by 1.
49  N5      CMPA  =BYTE/2=(5:5) N5. Round.
50          JL    N6           Is |tail| < ½b?
51          JG    5F
52          JXNZ  5F           Is |tail| > ½b?
53          STA   TEMP         |tail| = ½b; round to odd.
54          LDX   TEMP(4:4)
55          JXO   N6           To N6 if rX is odd.
56  5H      STA   *+1(0:0)     Store sign of rA.
57          INCA  BYTE         Add b^-4 to |f|.  (Sign varies)
58          JOV   N4           Check for rounding overflow.
59  N6      J2N   EXPUN        N6. Check e. Underflow if e < 0.
60  N7      ENTX  0,2          N7. Pack. rX ← e.
61          SRC   1
62  ZRO     DEC2  BYTE         rI2 ← e − b.
63  8H      STA   ACC
64  EXITF   J2N   *            Exit, unless e ≥ b.
65  EXPOV   HLT   2            Exponent overflow detected
66  EXPUN   HLT   1            Exponent underflow detected
67  ACC     CON   0            Floating point accumulator ∎
```

The rather long section of code from lines 25 to 37 is needed because MIX has only a 5-byte accumulator for adding signed numbers while in general $2p+1 = 9$ places of accuracy are required by Algorithm A. The program could be shortened to about half its present length if we were willing to sacrifice a little bit of its accuracy, but we shall see in the next section that full accuracy is important. Line 55 uses a nonstandard MIX instruction defined in Section 4.5.2. The running time for floating point addition and subtraction depends on several factors that are analyzed in Section 4.2.4.

Now let us consider multiplication and division, which are simpler than addition, and which are somewhat similar to each other.

Algorithm M (*Floating point multiplication or division*). Given base b, excess q, p-digit, normalized floating point numbers $u = (e_u, f_u)$ and $v = (e_v, f_v)$, this algorithm forms the product $w = u \otimes v$ or the quotient $w = u \oslash v$.

M1. [Unpack.] Separate the exponent and fraction parts of the representations of u and v. (Sometimes it is convenient, but not necessary, to test the operands for zero during this step.)

M2. [Operate.] Set

$$\begin{aligned} e_w &\leftarrow e_u + e_v - q, \qquad f_w \leftarrow f_u f_v \qquad \text{for multiplication;} \\ e_w &\leftarrow e_u - e_v + q + 1, \quad f_w \leftarrow (b^{-1} f_u)/f_v \quad \text{for division.} \end{aligned} \tag{9}$$

(Since the input numbers are assumed to be normalized, it follows that either $f_w = 0$, or $1/b^2 \le |f_w| < 1$, or a division-by-zero error has occurred.) If necessary, the representation of f_w may be reduced to $p+2$ or $p+3$ digits at this point, as in exercise 5.

M3. [Normalize.] Perform Algorithm N on (e_w, f_w) to normalize, round, and pack the result. (*Note:* Normalization is simpler in this case, since scaling left occurs at most once, and since rounding overflow cannot occur after division.) ∎

The following MIX subroutines, which are intended to be used in connection with Program A, illustrate the machine considerations necessary in connection with Algorithm M.

Program M (*Floating point multiplication and division*).

```
01  Q      EQU  BYTE/2        q is half the byte size
02  FMUL   STJ  EXITF         Floating point multiplication subroutine:
03         JOV  OFLO          Ensure overflow is off.
04         STA  TEMP          TEMP ← v.
05         LDX  ACC           rX ← u.
06         STX  FU(0:4)       FU ← ±fffff0.
07         LD1  TEMP(EXP)
08         LD2  ACC(EXP)
09         INC2 -Q,1          rI2 ← e_u + e_v − q.
10         SLA  1
11         MUL  FU            Multiply f_u times f_v.
12         JMP  NORM          Normalize, round, and exit.
```

```
13  FDIV  STJ   EXITF      Floating point division subroutine:
14        JOV   OFLO       Ensure overflow is off.
15        STA   TEMP       TEMP ← v.
16        STA   FV(0:4)    FV ← ±fffff0.
17        LD1   TEMP(EXP)
18        LD2   ACC(EXP)
19        DEC2  -Q,1       rI2 ← e_u − e_v + q.
20        ENTX  0
21        LDA   ACC
22        SLA   1          rA ← f_u.
23        CMPA  FV(1:5)
24        JL    *+3        Jump if |f_u| < |f_v|.
25        SRA   1          Otherwise, scale f_u right
26        INC2  1              and increase rI2 by 1.
27        DIV   FV         Divide.
28        JNOV  NORM       Normalize, round, and exit.
29  DVZRO HLT   3          Unnormalized or zero divisor  ∎
```

The most noteworthy feature of this program is the provision for division in lines 23–26, which is made in order to ensure enough accuracy to round the answer. If $|f_u| < |f_v|$, straightforward application of Algorithm M would leave a result of the form "$\pm 0 f f f f$" in register A, and this would not allow a proper rounding without a careful analysis of the remainder (which appears in register X). So the program computes $f_w \leftarrow f_u/f_v$ in this case, ensuring that f_w is either zero or normalized in all cases; rounding can proceed with five significant bytes, possibly testing whether the remainder is zero.

We occasionally need to convert values between fixed and floating point representations. A "fix-to-float" routine is easily obtained with the help of the normalization algorithm above; for example, in MIX, the following subroutine converts an integer to floating point form:

```
01  FLOT  STJ   EXITF    Assume that rA = u, an integer.
02        JOV   OFLO     Ensure overflow is off.
03        ENT2  Q+5      Set raw exponent.                (10)
04        ENTX  0
05        JMP   NORM     Normalize, round, and exit.  ∎
```

A "float-to-fix" subroutine is the subject of exercise 14.

The debugging of floating point subroutines is usually a difficult job, since there are so many cases to consider. Here is a list of common pitfalls that often trap a programmer or machine designer who is preparing floating point routines:

1) *Losing the sign.* On many machines (not MIX), shift instructions between registers will affect the sign, and the shifting operations used in normalizing and scaling numbers must be carefully analyzed. The sign is also lost frequently when minus zero is present. (For example, Program A is careful to retain the sign of register A in lines 30–34. See also exercise 6.)

2) *Failure to treat exponent underflow or overflow properly.* The size of e_w should not be checked until *after* the rounding and normalization, because preliminary tests may give an erroneous indication. Exponent underflow and overflow can occur on floating point addition and subtraction, not only during multiplication and division; and even though this is a rather rare occurrence, it must be tested each time. Enough information should be retained so that meaningful corrective actions are possible after overflow or underflow has occurred.

It has unfortunately become customary in many instances to ignore exponent underflow and simply to set underflowed results to zero with no indication of error. This causes a serious loss of accuracy in most cases (indeed, it is the loss of *all* the significant digits), and the assumptions underlying floating point arithmetic have broken down, so the programmer really must be told when underflow has occurred. Setting the result to zero is appropriate only in certain cases when the result is later to be added to a significantly larger quantity. When exponent underflow is not detected, we find mysterious situations in which $(u \otimes v) \otimes w$ is zero, but $u \otimes (v \otimes w)$ is not, since $u \otimes v$ results in exponent underflow but $u \otimes (v \otimes w)$ can be calculated without any exponents falling out of range. Similarly, we can find positive numbers a, b, c, d, and y such that

$$\begin{aligned} (a \otimes y \oplus b) \oslash (c \otimes y \oplus d) &\approx \tfrac{2}{3}, \\ (a \oplus b \oslash y) \oslash (c \oplus d \oslash y) &= 1 \end{aligned} \tag{11}$$

if exponent underflow is not detected. (See exercise 9.) Even though floating point routines are not precisely accurate, such a disparity as (11) is certainly unexpected when a, b, c, d, and y are all *positive*! Exponent underflow is usually not anticipated by a programmer, so he needs to be told about it.*

3) *Inserted garbage.* When scaling to the left it is important to keep from introducing anything but zeros at the right. For example, note the "`ENTX 0`" instruction in line 21 of Program A, and the all-too-easily-forgotten "`ENTX 0`" instruction in line 04 of the `FLOT` subroutine (10). (But it would be a mistake to clear register X after line 27 in the division subroutine.)

*On the other hand, it must be admitted that today's high-level programming languages give the programmer little or no satisfactory way to make use of the information that a floating point routine wants to tell him; and the `MIX` programs in this section, which simply '`HLT`' when errors are detected, are even worse. There are numerous important applications in which exponent underflow is relatively harmless, and it is desirable to find a way for programmers to cope with such situations easily and safely. The practice of silently replacing underflows by zero has been thoroughly discredited, but there is another alternative that has recently been gaining much favor, namely to modify the definition we have given for floating point numbers, allowing an unnormalized fraction part when the exponent has its smallest possible value. This idea of "gradual underflow," which was first embodied in the hardware of the Electrologica X8 computer, adds only a small amount of complexity to the algorithms, and it makes exponent underflow impossible during addition or subtraction. The simple formulas for relative error in Section 4.2.2 no longer hold in the presence of gradual underflow, so the topic is beyond the scope of this book. However, by using formulas like $\text{round}(x) = x(1-\delta)+\epsilon$, where $|\delta| < \frac{1}{2}b^{1-p}$ and $|\epsilon| < \frac{1}{2}b^{-p-q}$, one can show that gradual underflow succeeds in many important cases. See W. M. Kahan and J. Palmer, *ACM SIGNUM Newsletter* (Oct. 1979), 13–21.

4) *Unforeseen rounding overflow.* When a number like .999999997 is rounded to 8 digits, a carry will occur to the left of the decimal point, and the result must be scaled to the right. Many people have mistakenly concluded that rounding overflow is impossible during multiplication, since they look at the maximum value of $|f_u f_v|$, which is $1 - 2b^{-p} + b^{-2p}$; and this cannot round up to 1. The fallacy in this reasoning is exhibited in exercise 11. Curiously, it turns out that the phenomenon of rounding overflow *is* impossible during floating point division (see exercise 12).

There is a school of thought that says it is harmless to "round" a value like .999999997 to .99999999 instead of to 1.0000000, since this does not increase the worst-case bounds on relative error. The floating point number 1.0000000 may be said to represent all real values in the interval $[1.0000000 - 5 \times 10^{-8}, 1.0000000 + 5 \times 10^{-8}]$, while .99999999 represents all values in the much smaller interval $(.99999999 - 5 \times 10^{-9}, .99999999 + 5 \times 10^{-9})$. Even though the latter interval does not contain the original value .999999997, each number of the second interval is contained in the first, so subsequent calculations with the second interval are no less accurate than with the first. This ingenious argument is, however, incompatible with the mathematical philosophy of floating point arithmetic expressed in Section 4.2.2.

5) *Rounding before normalizing.* Inaccuracies are caused by premature rounding in the wrong digit position. This error is obvious when rounding is being done to the left of the appropriate position; but it is also dangerous in the less obvious cases where rounding is first done too far to the right, followed by rounding in the true position. For this reason it is a mistake to round during the "scaling-right" operation in step A5, except as prescribed in exercise 5. (The special case of rounding in step N5, then rounding again after rounding overflow has occurred, is harmless, however, because rounding overflow always yields ± 1.0000000 and this is unaffected by the subsequent rounding process.)

6) *Failure to retain enough precision in intermediate calculations.* Detailed analyses of the accuracy of floating point arithmetic, made in the next section, suggest strongly that normalizing floating point routines should always deliver a properly rounded result to the maximum possible accuracy. There should be no exceptions to this dictum, even in cases that occur with extremely low probability; the appropriate number of significant digits should be retained throughout the computations, as stated in Algorithms A and M.

C. Floating point hardware. Nearly every large computer intended for scientific calculations includes floating point arithmetic as part of its repertoire of built-in operations. Unfortunately, the design of such hardware usually includes some anomalies that result in dismally poor behavior in certain circumstances, and we hope that future computer designers will pay more attention to providing the proper behavior than they have in the past. It costs only a little more to build the machine right, and considerations in the following section show that substantial benefits will be gained. Yesterday's compromises are no longer appropriate for modern machines, based on what we know now.

The MIX computer, which is being used as an example of a "typical" machine in this series of books, has an optional "floating point attachment" (available at extra cost) that includes the following seven operations:

• FADD, FSUB, FMUL, FDIV, FLOT, FCMP (C = 1, 2, 3, 4, 5, 56, respectively; F = 6). The contents of rA after the operation "FADD V" are precisely the same as the contents of rA after the operations

```
STA ACC
LDA V
JMP FADD
```

where FADD is the subroutine that appears earlier in this section, except that both operands are automatically normalized before entry to the subroutine if they are not already in normalized form. (If exponent underflow occurs during this pre-normalization, but not during the normalization of the answer, no underflow is signalled.) Similar remarks apply to FSUB, FMUL, and FDIV. The contents of rA after the operation "FLOT" are the contents after "JMP FLOT" in the subroutine (10) above.

The contents of rA are unchanged by the operation "FCMP V"; this instruction sets the comparison indicator to less, equal, or greater, depending on whether the contents of rA are "definitely less than," "approximately equal to," or "definitely greater than" V; this subject is discussed in the next section, and the precise action is defined by the subroutine FCMP of exercise 4.2.2–17 with EPSILON in location 0.

No register other than rA is affected by any of the floating point operations. If exponent overflow or underflow occurs, the overflow toggle is turned on and the exponent of the answer is given modulo the byte size. Division by zero leaves undefined garbage in rA. Execution times: $4u$, $4u$, $9u$, $11u$, $3u$, $4u$, respectively.

• FIX (C = 5; F = 7). The contents of rA are replaced by the integer "round(rA)", rounding to the nearest integer as in step N5 of Algorithm N. However, if this answer is too large to fit in the register, the overflow toggle is set on and the result is undefined. Execution time: $3u$.

Sometimes it is helpful to use floating point operators in a nonstandard way. For example, if the operation FLOT had not been included as part of MIX's floating point attachment, we could easily achieve its effect on 4-byte numbers by writing

```
FLOT STJ  9F
     SLA  1
     ENTX Q+4
     SRC  1                    (12)
     FADD =0=
9H   JMP  *          ∎
```

This routine is not strictly equivalent to the FLOT operator, since it assumes that the 1:1 byte of rA is zero, and it destroys rX. The handling of more general situations is a little tricky, because rounding overflow can occur even during a FLOT operation.

Similarly, suppose MIX had a FADD operation but not FIX. If we wanted to round a number u from floating point form to the nearest fixed point integer, and if we knew that the number was nonnegative and would fit in at most three bytes, we could write

```
FADD FUDGE
```

where location FUDGE contains the constant

+	Q+4	1	0	0	0	;

the result in rA would be

+	Q+4	1	round(u)	.	(13)

D. History and bibliography. The origins of floating point notation can be traced back to Babylonian mathematicians (1800 B.C. or earlier), who made extensive use of radix-60 floating point arithmetic but did not have a notation for the exponents. The appropriate exponent was always somehow "understood" by the man doing the calculations. At least one case has been found in which the wrong answer was given because addition was performed with improper alignment of the operands, but such examples are very rare; see O. Neugebauer, *The Exact Sciences in Antiquity* (Princeton, N. J.: Princeton University Press, 1952), 26–27. Another early contribution to floating point notation is due to the Greek mathematician Apollonius (3rd century B.C.), who apparently was the first to explain how to simplify multiplication by collecting powers of 10 separately from their coefficients, at least in simple cases. [For a discussion of Apollonius's method, see Pappus, *Mathematical Collections* (4th century A.D.).] After the Babylonian civilization died out, the first significant uses of floating point notation for products and quotients did not emerge until much later, about the time logarithms were invented (1600) and shortly afterwards when Oughtred invented the slide rule (1630). The modern notation "x^n" for exponents was being introduced at about the same time; separate symbols for x squared, x cubed, etc., had been in use before this.

Floating point arithmetic was incorporated into the design of some of the earliest computers. It was independently proposed by Leonardo Torres y Quevedo in Madrid, 1914; by Konrad Zuse in Berlin, 1936; and by George Stibitz in New Jersey, 1939. Zuse's machines used a floating binary representation that he called "semi-logarithmic notation"; he also incorporated conventions for dealing with special quantities like "∞" and "undefined." The first American computers to operate with floating point arithmetic hardware were the Bell Laboratories' Model V and the Harvard Mark II, both of which were relay calculators designed in 1944. [See B. Randell, *The Origins of Digital Computers* (Berlin: Springer, 1973), 100, 155, 163–164, 259–260; *Proc. Symp. Large-Scale Digital Calculating Machinery* (Harvard, 1947), 41–68, 69–79; *Datamation* **13** (April 1967), 35–44 (May 1967), 45–49; *Zeit. für angew. Math. und Physik* **1** (1950), 345–346.]

The use of floating binary arithmetic was seriously considered in 1944–1946 by researchers at the Moore School in their plans for the first *electronic* digital computers, but it turned out to be much harder to implement floating point circuitry with tubes than with relays. The group realized that scaling was a problem in programming; but at the time it was only a very small part of a total programming job, and it seemed to be worth the time and trouble it took, since it tended to keep a programmer aware of the numerical accuracy he was getting. Furthermore, they argued that floating point representation would take up valuable memory space, since the exponents must be stored, and that it would be difficult to adapt floating point arithmetic to multiple-precision calculations. [See von Neumann's *Collected Works* **5** (New York: Macmillan, 1963), 43, 73–74.] At this time, of course, they were designing the first stored-program computer and the second electronic computer, and their choice had to be either fixed point *or* floating point arithmetic, not both. They anticipated the coding of floating binary routines, and in fact "shift left" and "shift right" instructions were put into their machine primarily to make such routines more efficient. The first machine to have both kinds of arithmetic in its hardware was apparently a computer developed at General Electric Company [see *Proc. 2nd Symp. Large-Scale Digital Calculating Machinery* (Cambridge: Harvard University Press, 1951), 65–69].

Floating point subroutines and interpretive systems for early machines were coded by D. J. Wheeler and others, and the first publication of such routines was in *The Preparation of Programs for an Electronic Digital Computer* by Wilkes, Wheeler, and Gill (Reading, Mass.: Addison-Wesley, 1951), subroutines A1–A11, pp. 35–37, 105–117. It is interesting to note that floating *decimal* subroutines are described here, although a binary computer was being used; in other words, the numbers were represented as $10^e f$, not $2^e f$, and therefore the scaling operations required multiplication or division by 10. On this particular machine such decimal scaling was about as easy as shifting, and the decimal approach greatly simplified input/output conversions.

Most published references to the details of floating point arithmetic routines are scattered in "technical memorandums" distributed by various computer manufacturers, but there have been occasional appearances of these routines in the open literature. Besides the reference above, the following are of historical interest: R. H. Stark and D. B. MacMillan, *Math. Comp.* **5** (1951), 86–92, where a plugboard-wired program is described; D. McCracken, *Digital Computer Programming* (New York: Wiley, 1957), 121–131; J. W. Carr III, *CACM* **2**,5 (May 1959), 10–15; W. G. Wadey, *JACM* **7** (1960), 129–139; D. E. Knuth, *JACM* **8** (1961), 119–128; O. Kesner, *CACM* **5** (1962), 269–271; F. P. Brooks and K. E. Iverson, *Automatic Data Processing* (New York: Wiley, 1963), 184–199. For a discussion of floating point arithmetic from a computer designer's standpoint, see "Floating point operation" by S. G. Campbell, in *Planning a computer System*, ed. by W. Buchholz (New York: McGraw-Hill, 1962), 92–121. A set of algorithms by J. Coonen, W. M. Kahan, and H. S. Stone, submitted to the IEEE Microprocessor Floating-Point Standards Committee during 1978–1980, represented

the state of the floating point art as of 1980; these carefully considered procedures will probably be published some day. Additional references, which deal primarily with the accuracy of floating point methods, are given in Section 4.2.2.

EXERCISES

1. [*10*] How would Avogadro's number and Planck's constant be represented in base 100, excess 50, four-digit floating point notation? (This would be the representation used by MIX, as in (4), if the byte size is 100.)

2. [*12*] Assume that the exponent e is constrained to lie in the range $0 \le e \le E$; what are the largest and smallest positive values that can be written as base b, excess q, p-digit floating point numbers? What are the largest and smallest positive values that can be written as *normalized* floating point numbers with these specifications?

3. [*11*] (K. Zuse, 1936.) Show that if we are using normalized floating binary arithmetic, there is a way to increase the precision slightly without loss of memory space: A p-bit fraction part can be represented using only $p - 1$ bit positions of a computer word, if the range of exponent values is decreased very slightly.

▶ **4.** [*12*] Assume that $b = 10$, $p = 8$. What result does Algorithm A give for $(50, +.98765432) \oplus (49, +.33333333)$? For $(53, -.99987654) \oplus (54, +.10000000)$? For $(45, -.50000001) \oplus (54, +.10000000)$?

▶ **5.** [*24*] Let us say that $x \sim y$ (with respect to a given radix b) if x and y are real numbers satisfying the following conditions:

$$\lfloor x/b \rfloor = \lfloor y/b \rfloor;$$

$$\begin{aligned}
x \bmod b = 0 \quad &\text{iff} \quad y \bmod b = 0;\\
0 < x \bmod b < \tfrac{1}{2}b \quad &\text{iff} \quad 0 < y \bmod b < \tfrac{1}{2}b;\\
x \bmod b = \tfrac{1}{2}b \quad &\text{iff} \quad y \bmod b = \tfrac{1}{2}b;\\
\tfrac{1}{2}b < x \bmod b < b \quad &\text{iff} \quad \tfrac{1}{2}b < y \bmod b < b.
\end{aligned}$$

Prove that if f_v is replaced by $b^{-p-2}F_v$ between steps A5 and A6 of Algorithm A, where $F_v \sim b^{p+2}f_v$, the result of that algorithm will be unchanged. (If F_v is an integer and b is even, this operation essentially truncates f_v to $p + 2$ places while remembering whether any nonzero digits have been dropped, thereby minimizing the length of register that is needed for the addition in step A6.)

6. [*20*] If the result of a FADD instruction is zero, what will be the sign of rA, according to the definitions of MIX's floating point attachment given in this section?

7. [*27*] Discuss floating point arithmetic using balanced ternary notation.

8. [*20*] Give examples of normalized eight-digit floating decimal numbers u and v for which addition yields (a) exponent underflow, (b) exponent overflow, assuming that exponents must satisfy $0 \le e < 100$.

9. [*M24*] (W. M. Kahan.) Assume that the occurrence of exponent underflow causes the result to be replaced by zero, with no error indication given. Using excess zero, eight-digit floating decimal numbers with e in the range $-50 \le e < 50$, find positive values of a, b, c, d, and y such that (11) holds.

10. [*12*] Give an example of normalized eight-digit floating decimal numbers u and v for which rounding overflow occurs in addition.

▸ **11.** [*M20*] Give an example of normalized, excess 50, eight-digit floating decimal numbers u and v for which rounding overflow occurs in multiplication.

12. [*M25*] Prove that rounding overflow cannot occur during the normalization phase of floating point division.

13. [*30*] When doing "interval arithmetic" we don't want to round the results of a floating point computation; we want rather to implement operations such as $\triangledown$ and $\triangle$, which give the tightest possible representable bounds on the true sum:

$$u \mathbin{\triangledown} v \;\le\; u + v \;\le\; u \mathbin{\triangle} v.$$

How should the algorithms of this section be modified for such a purpose?

14. [*25*] Write a MIX subroutine that begins with an arbitrary floating point number in register A, not necessarily normalized, and converts it to the nearest fixed point integer (or determines that the number is too large in absolute value to make such a conversion possible).

▸ **15.** [*28*] Write a MIX subroutine, to be used in connection with the other subroutines of this section, that calculates $u \bmod 1$, that is, $u - \lfloor u \rfloor$ rounded to nearest floating point number, given a floating point number u. Note that when u is a very small negative number, $u \bmod 1$ will be rounded so that the result is unity (even though $u \bmod 1$ has been defined to be always *less* than unity, as a real number).

16. [*HM21*] (Robert L. Smith.) Design an algorithm to compute the real and imaginary parts of the complex number $(a+bi)/(c+di)$, given real floating point values a, b, c, and d. Avoid the computation of c^2+d^2, since it would cause floating point overflow even when $|c|$ or $|d|$ is approximately the square root of the maximum allowable floating point value.

17. [*40*] (John Cocke.) Explore the idea of extending the range of floating point numbers by defining a single-word representation in which the precision of the fraction decreases as the magnitude of the exponent increases.

18. [*25*] Consider a binary computer with 36-bit words, on which positive floating binary numbers are represented as $(0e_1e_2\ldots e_8f_1f_2\ldots f_{27})_2$; here $(e_1e_2\ldots e_8)_2$ is an excess $(10000000)_2$ exponent and $(f_1f_2\ldots f_{27})_2$ is a 27-bit fraction. Negative floating point numbers are represented by the *two's complement* of the corresponding positive representation (see Section 4.1). Thus, 1.5 is *201|600000000* in octal notation, while -1.5 is *576|200000000*; the octal representations of 1.0 and -1.0 are *201|400000000* and *576|400000000*, respectively. (A vertical line is used here to show the boundary between exponent and fraction.) Note that bit f_1 of a normalized positive number is always 1, while it is almost always zero for negative numbers; the exceptional cases are representations of -2^k.

Suppose that the exact result of a floating point operation has the octal code *572|740000000|01*; this (negative) 33-bit fraction must be normalized and rounded to 27 bits. If we shift left until the leading fraction bit is zero, we get *576|000000000|20*, but this rounds to the illegal value *576|000000000*; we have over-normalized, since the correct answer is *575|400000000*. On the other hand if we start (in some other problem) with the value *572|740000000|05* and stop before over-normalizing it, we get *575|400000000|50*, which rounds to the unnormalized number *575|400000001*; subsequent normalization yields *576|000000002* while the correct answer is *576|000000001*.

Give a simple, correct rounding rule that resolves this dilemma on such a machine (without abandoning two's complement notation).

Round numbers are always false.

—SAMUEL JOHNSON (1750)

I shall speak in round numbers, not absolutely accurate, yet not so wide from truth as to vary the result materially.

—THOMAS JEFFERSON (1824)

19. [24] What is the running time for the FADD subroutine in Program A, in terms of relevant characteristics of the data? What is the maximum running time, over all inputs that do not cause overflow or underflow?

4.2.2. Accuracy of Floating Point Arithmetic

Floating point computation is by nature inexact, and it is not difficult to misuse it so that the computed answers consist almost entirely of "noise." One of the principal problems of numerical analysis is to determine how accurate the results of certain numerical methods will be. A "credibility-gap" problem is involved here: we don't know how much of the computer's answers to believe. Novice computer users solve this problem by implicitly trusting in the computer as an infallible authority; they tend to believe that all digits of a printed answer are significant. Disillusioned computer users have just the opposite approach, they are constantly afraid that their answers are almost meaningless. Many a serious mathematician has attempted to give rigorous analyses of a sequence of floating point operations, but has found the task to be so formidable that he has tried to content himself with plausibility arguments instead.

A thorough examination of error analysis techniques is, of course, beyond the scope of this book, but in this section we shall study some of the characteristics of floating point arithmetic errors. Our goal is to discover how to perform floating point arithmetic in such a way that reasonable analyses of error propagation are facilitated as much as possible.

A rough (but reasonably useful) way to express the behavior of floating point arithmetic can be based on the concept of "significant figures" or *relative error.* If we are representing an exact real number x inside a computer by using the approximation $\hat{x} = x(1 + \epsilon)$, the quantity $\epsilon = (\hat{x} - x)/x$ is called the relative error of approximation. Roughly speaking, the operations of floating point multiplication and division do not magnify the relative error by very much; but floating point subtraction of nearly equal quantities (and floating point addition, $u \oplus v$, where u is nearly equal to $-v$) can very greatly increase the relative error. So we have a general rule of thumb, that a substantial loss of accuracy is expected from such additions and subtractions, but not from multiplications and divisions. On the other hand, the situation is somewhat paradoxical and needs to be understood properly, since "bad" additions and subtractions are performed with perfect accuracy! (See exercise 25.)

One of the consequences of the possible unreliability of floating point addition is that the associative law breaks down:

$$(u \oplus v) \oplus w \neq u \oplus (v \oplus w), \qquad \text{for many } u, v, w. \tag{1}$$

For example,

$$\begin{aligned}(11111113. \oplus -11111111.) \oplus 7.5111111 &= 2.0000000 \oplus 7.5111111\\ &= 9.5111111;\\ 11111113. \oplus (-11111111. \oplus 7.5111111) &= 11111113. \oplus -11111103.\\ &= 10.000000.\end{aligned}$$

(All examples in this section are given in eight-digit floating decimal arithmetic, with exponents indicated by an explicit decimal point. Recall that, as in Section 4.2.1, the symbols $\oplus$, $\ominus$, $\otimes$, $\oslash$ are used to stand for floating point operations corresponding to the exact operations $+$, $-$, $\times$, $/$.)

In view of the failure of the associative law, the comment of Mrs. La Touche that appears at the beginning of this chapter [taken from *Math. Gazette* **12** (1924), 95] makes a good deal of sense with respect to floating point arithmetic. Mathematical notations like "$a_1 + a_2 + a_3$" or "$\sum_{1 \le k \le n} a_k$" are inherently based upon the assumption of associativity, so a programmer must be especially careful that he does not implicitly assume the validity of the associative law.

A. An axiomatic approach. Although the associative law is not valid, the commutative law

$$u \oplus v = v \oplus u \tag{2}$$

does hold, and this law can be a valuable conceptual asset in programming and in the analysis of programs. This example suggests that we should look for important laws that *are* satified by $\oplus$, $\ominus$, $\otimes$, and $\oslash$; it is not unreasonable to say that *floating point routines should be designed to preserve as many of the ordinary mathematical laws as possible.* If more axioms are valid, it becomes easier to write good programs, and programs also become more portable from machine to machine.

Let us therefore consider some of the other basic laws that are valid for normalized floating point operations as described in the previous section. First we have

$$u \ominus v = u \oplus -v; \tag{3}$$

$$-(u \oplus v) = -u \oplus -v; \tag{4}$$

$$u \oplus v = 0 \quad \text{if and only if} \quad v = -u; \tag{5}$$

$$u \oplus 0 = u. \tag{6}$$

From these laws we can derive further identities; for example (exercise 1),

$$u \ominus v = -(v \ominus u). \tag{7}$$

Identities (2) to (6) are easily deduced from the algorithms in Section 4.2.1. The following rule is slightly less obvious:

$$\text{if} \quad u \le v \quad \text{then} \quad u \oplus w \le v \oplus w. \tag{8}$$

Instead of attempting to prove this rule by analyzing Algorithm 4.2.1A, let us go back to the principle underlying the design of that algorithm. (Algorithmic proofs aren't always easier than mathematical ones.) Our idea was that the floating point operations should satisfy

$$\begin{aligned} u \oplus v &= \text{round}(u + v), \qquad u \ominus v = \text{round}(u - v), \\ u \otimes v &= \text{round}(u \times v), \qquad u \oslash v = \text{round}(u \;/\; v), \end{aligned} \tag{9}$$

where round(x) denotes the best floating point approximation to x as defined in Algorithm 4.2.1N. We have

$$\text{round}(-x) = -\text{round}(x), \tag{10}$$

$$x \le y \quad \text{implies} \quad \text{round}(x) \le \text{round}(y), \tag{11}$$

and these fundamental relations prove properties (2) through (8) immediately. We can also write down several more identities:

$$u \otimes v = v \otimes u, \qquad (-u) \otimes v = -(u \otimes v), \qquad 1 \otimes v = v;$$

$$u \otimes v = 0 \quad \text{if and only if} \quad u = 0 \text{ or } v = 0;$$

$$(-u) \oslash v = u \oslash (-v) = -(u \oslash v);$$

$$0 \oslash v = 0, \qquad u \oslash 1 = u, \qquad u \oslash u = 1.$$

If $u \le v$ and $w > 0$, then $u \otimes w \le v \otimes w$ and $u \oslash w \le v \oslash w$ and $w \oslash u \ge w \oslash v$. If $u \oplus v = u + v$, then $(u \oplus v) \ominus v = u$; and if $u \otimes v = u \times v \neq 0$, then $(u \otimes v) \oslash v = u$. We see that a good deal of regularity is present in spite of the inexactness of the floating point operations, when things have been defined properly.

Several familiar rules of algebra are still, of course, conspicuously absent from the collection of identities above; the associative law for floating point multiplication is not strictly true, as shown in exercise 3, and the distributive law between $\otimes$ and $\oplus$ can fail rather badly: Let $u = 20000.000$, $v = -6.0000000$, and $w = 6.0000003$; then

$$(u \otimes v) \oplus (u \otimes w) = -120000.00 \oplus 120000.01 = .010000000$$

$$u \otimes (v \oplus w) = 20000.000 \otimes .00000030000000 = .0060000000$$

so

$$u \otimes (v \oplus w) \neq (u \otimes v) \oplus (u \otimes w). \tag{12}$$

On the other hand we do have $b \otimes (v \oplus w) = (b \otimes v) \oplus (b \otimes w)$, when b is the floating point radix, since

$$\text{round}(bx) = b\,\text{round}(x). \tag{13}$$

(Strictly speaking, the identities and inequalities we are considering in this section implicitly assume that exponent underflow and overflow do not occur. The function round(x) is undefined when $|x|$ is too small or too large, and equations such as (13) hold only when both sides are defined.)

The failure of Cauchy's fundamental inequality

$$(x_1^2 + \cdots + x_n^2)(y_1^2 + \cdots + y_n^2) \geq (x_1y_1 + \cdots + x_ny_n)^2$$

is another important example of the breakdown of traditional algebra in the presence of floating point arithmetic. Exercise 7 shows that Cauchy's inequality can fail even in the simple case $n = 2$, $x_1 = x_2 = 1$. Novice programmers who calculate the standard deviation of some observations by using the textbook formula

$$\sigma = \sqrt{\Bigg(n \sum_{1\leq k\leq n} x_k^2 - \Bigg(\sum_{1\leq k\leq n} x_k\Bigg)^2\Bigg) \Big/ n(n-1)} \tag{14}$$

often find themselves taking the square root of a negative number! A much better way to calculate means and standard deviations with floating point arithmetic is to use the recurrence formulas

$$M_1 = x_1, \qquad M_k = M_{k-1} \oplus (x_k \ominus M_{k-1}) \oslash k, \tag{15}$$

$$S_1 = 0, \qquad S_k = S_{k-1} \oplus (x_k \ominus M_{k-1}) \otimes (x_k \ominus M_k), \tag{16}$$

for $2 \leq k \leq n$, where $\sigma = \sqrt{S_n/(n-1)}$. [Cf. B. P. Welford, *Technometrics* **4** (1962), 419–420.] With this method S_n can never be negative, and we avoid other serious problems encountered by the naïve method of accumulating sums, as shown in exercise 16. (See exercise 19 for a summation technique that provides an even better guarantee on the accuracy.)

Although algebraic laws do not always hold exactly, we can often show that they aren't too far off base. When $b^{e-1} \leq x < b^e$ we have $\text{round}(x) = x + \rho(x)$, where $|\rho(x)| \leq \frac{1}{2}b^{e-p}$; hence

$$\text{round}(x) = x\big(1 + \delta(x)\big), \tag{17}$$

where the relative error is bounded independently of x:

$$|\delta(x)| \leq \tfrac{1}{2}/(b^{1-p} + \tfrac{1}{2}) < \tfrac{1}{2}b^{1-p}. \tag{18}$$

We can use this inequality to estimate the relative error of normalized floating point calculations in a simple way, since $u \oplus v = (u + v)(1 + \delta(u + v))$, etc.

As an example of typical error-estimation procedures, let us consider the associative law for multiplication. Exercise 3 shows that $(u \otimes v) \otimes w$ is not in general equal to $u \otimes (v \otimes w)$; but the situation in this case is much better than it was with respect to the associative law of addition (1) and the distributive law (12). In fact, we have

$$(u \otimes v) \otimes w = \big((uv)(1+\delta_1)\big) \otimes w = uvw(1+\delta_1)(1+\delta_2),$$
$$u \otimes (v \otimes w) = u \otimes \big((vw)(1+\delta_3)\big) = uvw(1+\delta_3)(1+\delta_4),$$

for some δ_1, δ_2, δ_3, δ_4, provided that no exponent underflow or overflow occurs, where $|\delta_j| < \frac{1}{2}b^{1-p}$ for each j. Hence

$$\frac{(u \otimes v) \otimes w}{u \otimes (v \otimes w)} = \frac{(1+\delta_1)(1+\delta_2)}{(1+\delta_3)(1+\delta_4)} = 1+\delta,$$

where

$$|\delta| < 2b^{1-p}/(1 - \tfrac{1}{2}b^{1-p})^2. \tag{19}$$

The number b^{1-p} occurs so often in such analyses, it has been given a special name, one *ulp*, meaning one "unit in the last place" of the fraction part. Floating point operations are correct to within half an ulp, and the calculation of uvw by two floating point multiplications will be correct within about one ulp (ignoring second-order terms). Hence the associative law for multiplication holds to within about two ulps of relative error.

We have shown that $(u \otimes v) \otimes w$ is approximately equal to $u \otimes (v \otimes w)$, except when exponent overflow or underflow is a problem. It is worthwhile to study this intuitive idea of being "approximately equal" in more detail; can we make such a statement more precise in a reasonable way?

A programmer using floating point arithmetic almost never wants to test if two computed values are exactly equal to each other (or at least he hardly ever should try to do so), because this is an extremely improbable occurrence. For example, if a recurrence relation

$$x_{n+1} = f(x_n)$$

is being used, where the theory in some textbook says that x_n approaches a limit as $n \to \infty$, it is usually a mistake to wait until $x_{n+1} = x_n$ for some n, since the sequence x_n might be periodic with a longer period due to the rounding of intermediate results. The proper procedure is to wait until $|x_{n+1} - x_n| < \delta$, for some suitably chosen number δ; but since we don't necessarily know the order of magnitude of x_n in advance, it is even better to wait until

$$|x_{n+1} - x_n| \le \epsilon |x_n|; \tag{20}$$

now ϵ is a number that is much easier to select. This relation (20) is another way of saying that x_{n+1} and x_n are approximately equal; and our discussion

indicates that a relation of "approximately equal" would be more useful than the traditional relation of equality, when floating point computations are involved, if we could only define a suitable approximation relation.

In other words, the fact that strict equality of floating point values is of little importance implies that we ought to have a new operation, *floating point comparison*, which is intended to help assess the relative values of two floating point quantities. The following definitions seem to be appropriate for base b, excess q, floating point numbers $u = (e_u, f_u)$ and $v = (e_v, f_v)$:

$$u \prec v \quad (\epsilon) \qquad \text{if and only if} \qquad v - u > \epsilon \max(b^{e_u - q}, b^{e_v - q}); \tag{21}$$

$$u \sim v \quad (\epsilon) \qquad \text{if and only if} \qquad |v - u| \leq \epsilon \max(b^{e_u - q}, b^{e_v - q}); \tag{22}$$

$$u \succ v \quad (\epsilon) \qquad \text{if and only if} \qquad u - v > \epsilon \max(b^{e_u - q}, b^{e_v - q}); \tag{23}$$

$$u \approx v \quad (\epsilon) \qquad \text{if and only if} \qquad |v - u| \leq \epsilon \min(b^{e_u - q}, b^{e_v - q}). \tag{24}$$

These definitions apply to unnormalized values as well as to normalized ones. Note that exactly one of the conditions $u \prec v$ (definitely less than), $u \sim v$ (approximately equal to), or $u \succ v$ (definitely greater than) must always hold for any given pair of values u and v. The relation $u \approx v$ is somewhat stronger than $u \sim v$, and it might be read "u is essentially equal to v." All of the relations are given in terms of a positive real number ϵ that measures the degree of approximation being considered.

One way to view the above definitions is to associate a "neighborhood" set $N(u) = \{ x \mid |x - u| \leq \epsilon b^{e_u - q} \}$ with each floating point number u; thus, $N(u)$ represents a set of values near u based on the exponent of u's floating point representation. In these terms, we have $u \prec v$ if and only if $N(u) < v$ and $u < N(v)$; $u \sim v$ if and only if $u \in N(v)$ or $v \in N(u)$; $u \succ v$ if and only if $u > N(v)$ and $N(u) > v$; $u \approx v$ if and only if $u \in N(v)$ and $v \in N(u)$. (Here we are assuming that the parameter ϵ, which measures the degree of approximation, is a constant; a more complete notation would indicate the dependence of $N(u)$ upon ϵ.)

Here are some simple consequences of the above definitions:

$$\text{if} \quad u \prec v \quad (\epsilon) \quad \text{then} \quad v \succ u \quad (\epsilon); \tag{25}$$

$$\text{if} \quad u \approx v \quad (\epsilon) \quad \text{then} \quad u \sim v \quad (\epsilon); \tag{26}$$

$$u \approx u \quad (\epsilon); \tag{27}$$

$$\text{if} \quad u \prec v \quad (\epsilon) \quad \text{then} \quad u < v; \tag{28}$$

$$\text{if} \quad u \prec v \quad (\epsilon_1) \quad \text{and} \quad \epsilon_1 \geq \epsilon_2 \quad \text{then} \quad u \prec v \quad (\epsilon_2); \tag{29}$$

$$\text{if} \quad u \sim v \quad (\epsilon_1) \quad \text{and} \quad \epsilon_1 \leq \epsilon_2 \quad \text{then} \quad u \sim v \quad (\epsilon_2); \tag{30}$$

$$\text{if} \quad u \approx v \quad (\epsilon_1) \quad \text{and} \quad \epsilon_1 \leq \epsilon_2 \quad \text{then} \quad u \approx v \quad (\epsilon_2); \tag{31}$$

$$\text{if} \quad u \prec v \quad (\epsilon_1) \quad \text{and} \quad v \prec w \quad (\epsilon_2) \quad \text{then} \quad u \prec w \quad (\epsilon_1 + \epsilon_2); \tag{32}$$

$$\text{if} \quad u \approx v \quad (\epsilon_1) \quad \text{and} \quad v \approx w \quad (\epsilon_2) \quad \text{then} \quad u \sim w \quad (\epsilon_1 + \epsilon_2). \tag{33}$$

Moreover, we can prove without difficulty that

$$|u - v| \le \epsilon|u| \quad \text{and} \quad |u - v| \le \epsilon|v| \quad \text{implies} \quad u \approx v \quad (\epsilon); \tag{34}$$
$$|u - v| \le \epsilon|u| \quad \text{or} \quad |u - v| \le \epsilon|v| \quad \text{implies} \quad u \sim v \quad (\epsilon); \tag{35}$$

and conversely, for *normalized* floating point numbers u and v, when $\epsilon < 1$,

$$u \approx v \quad (\epsilon) \quad \text{implies} \quad |u - v| \le b\epsilon|u| \quad \text{and} \quad |u - v| \le b\epsilon|v|; \tag{36}$$
$$u \sim v \quad (\epsilon) \quad \text{implies} \quad |u - v| \le b\epsilon|u| \quad \text{or} \quad |u - v| \le b\epsilon|v|. \tag{37}$$

Let $\epsilon_0 = b^{1-p}$ be one ulp. The derivation of (17) establishes the inequality $|x - \text{round}(x)| = |\rho(x)| < \frac{1}{2}\epsilon_0 \min\bigl(|x|, |\text{round}(x)|\bigr)$, hence

$$x \approx \text{round}(x) \quad (\tfrac{1}{2}\epsilon_0); \tag{38}$$

it follows that $u \oplus v \approx u + v \ (\frac{1}{2}\epsilon_0)$, etc. The approximate associative law for multiplication derived above can be recast as follows: We have

$$|(u \otimes v) \otimes w - u \otimes (v \otimes w)| < \frac{2\epsilon_0}{(1 - \frac{1}{2}\epsilon_0)^2}|u \otimes (v \otimes w)|$$

by (19), and the same inequality is valid with $(u \otimes v) \otimes w$ and $u \otimes (v \otimes w)$ interchanged. Hence by (34),

$$(u \otimes v) \otimes w \approx u \otimes (v \otimes w) \quad (\epsilon) \tag{39}$$

whenever $\epsilon \ge 2\epsilon_0/(1 - \frac{1}{2}\epsilon_0)^2$. For example, if $b = 10$ and $p = 8$ we may take $\epsilon = 0.00000021$.

The relations $\prec$, $\sim$, $\succ$, and $\approx$ are useful within numerical algorithms, and it is therefore a good idea to provide routines for comparing floating point numbers as well as for doing arithmetic on them.

Let us now shift our attention back to the question of finding *exact* relations that are satisfied by the floating point operations. It is interesting to note that floating point addition and subtraction are not completely intractable from an axiomatic standpoint, since they do satisfy the nontrivial identities stated in the following theorems.

Theorem A. *Let u and v be normalized floating point numbers. Then*

$$\bigl((u \oplus v) \ominus u\bigr) + \bigl((u \oplus v) \ominus \bigl((u \oplus v) \ominus u\bigr)\bigr) = u \oplus v, \tag{40}$$

provided that no exponent overflow or underflow occurs.

This rather cumbersome-looking identity can be rewritten in a simpler manner: Let

$$\begin{aligned} u' &= (u \oplus v) \ominus v, \qquad & v' &= (u \oplus v) \ominus u; \\ u'' &= (u \oplus v) \ominus v', \qquad & v'' &= (u \oplus v) \ominus u'. \end{aligned} \tag{41}$$

Intuitively, u' and u'' should be approximations to u, and v' and v'' should be approximations to v. Theorem A tells us that

$$u \oplus v = u' + v'' = u'' + v'. \tag{42}$$

This is a stronger statement than the identity

$$u \oplus v = u' \oplus v'' = u'' \oplus v', \tag{43}$$

which follows by rounding (42).

Proof. Let us say that t is a *tail* of x modulo b^e if

$$t \equiv x \text{ (modulo } b^e), \qquad |t| \le \tfrac{1}{2} b^e; \tag{44}$$

thus, $x - \text{round}(x)$ is always a tail of x. The proof of Theorem A rests largely on the following simple fact proved in exercise 11:

Lemma T. *If t is a tail of the floating point number x, then $x \ominus t = x - t$.* ∎

Let $w = u \oplus v$. Theorem A holds trivially when $w = 0$. By multiplying all variables by a suitable power of b, we may assume without loss of generality that $e_w = p$. Then $u + v = w + r$, where r is a tail of $u + v$ modulo 1. Furthermore $u' = \text{round}(w - v) = \text{round}(u - r) = u - r - t$, where t is a tail of $u - r$ modulo b^e and $e = e_{u'} - p$.

If $e \le 0$, then $t \equiv u - r \equiv -v$ (modulo b^e), hence t is a tail of $-v$ and $v'' = \text{round}(w - u') = \text{round}(v + t) = v + t$; this proves (40). If $e > 0$, then $|u - r| \ge b^p - \frac{1}{2}$; and since $|r| \le \frac{1}{2}$, we have $|u| \ge b^p - 1$. It follows that r is a tail of v modulo 1. If $u' = u$, we have $v'' = \text{round}(w - u) = \text{round}(v - r) = v - r$. Otherwise the relation $\text{round}(u - r) \ne u$ implies that $|u| = b^p - 1$, $|r| = \frac{1}{2}$, $|u'| = b^p$; we have $v'' = \text{round}(w - u') = \text{round}(v + r) = v + r$, because r is also a tail of $-v$ in this case. ∎

Theorem A exhibits a regularity property of floating point addition, but it doesn't seem to be an especially useful result. The following identity is more significant:

Theorem B. *Under the hypotheses of Theorem A and* (41),

$$u + v = (u \oplus v) + \big((u \ominus u') \oplus (v \ominus v'')\big). \tag{45}$$

Proof. In fact, we can show that $u \ominus u' = u - u'$, $v \ominus v'' = v - v''$, and $(u - u') \oplus (v - v'') = (u - u') + (v - v'')$, hence (45) will follow from Theorem A.

Using the notation of the preceding proof, these relations are respectively equivalent to

$$\text{round}(t + r) = t + r, \qquad \text{round}(t) = t, \qquad \text{round}(r) = r. \tag{46}$$

Exercise 12 establishes the theorem in the special case $|e_u - e_v| \geq p$. Otherwise $u + v$ has at most $2p$ significant digits and it is easy to see that $\text{round}(r) = r$. If now $e > 0$, the proof of Theorem A shows that $t = -r$ or $t = r = \pm\frac{1}{2}$. If $e \leq 0$ we have $t + r \equiv u$ and $t \equiv -v$ (modulo b^e); this is enough to prove that $t + r$ and r round to themselves, provided that $e_u \geq e$ and $e_v \geq e$. But either $e_u < 0$ or $e_v < 0$ would contradict our hypothesis that $|e_u - e_v| < p$, since $e_w = p$. ∎

Theorem B gives *an explicit formula for the difference* between $u + v$ and $u \oplus v$, in terms of quantities that can be calculated directly using five operations of floating point arithmetic. If the radix b is 2 or 3, we can improve on this result, obtaining the exact value of the correction term with only two floating point operations and one (fixed point) comparison of absolute values:

Theorem C. *If $b \leq 3$ and $|u| \geq |v|$, then*

$$u + v = (u \oplus v) + \big(u \ominus (u \oplus v)\big) \oplus v. \tag{47}$$

Proof. Following the conventions of preceding proofs again, we wish to show that $v \ominus v' = r$. It suffices to show that $v' = w - u$, because (46) will then yield $v \ominus v' = \text{round}(v - v') = \text{round}(u + v - w) = \text{round}(r) = r$.

We shall in fact prove (47) whenever $b \leq 3$ and $e_u \geq e_v$. If $e_u \geq p$, then r is a tail of v modulo 1, hence $v' = w \ominus u = v \ominus r = v - r = w - u$ as desired. If $e_u < p$, then we must have $e_u = p - 1$, and $w - u$ is a multiple of b^{-1}; it will therefore round to itself if its magnitude is less than $b^{p-1} + b^{-1}$. Since $b \leq 3$, we have indeed $|w - u| \leq |w - u - v| + |v| \leq \frac{1}{2} + (b^{p-1} - b^{-1}) < b^{p-1} + b^{-1}$. This completes the proof. ∎

The proofs of Theorems A, B, and C do not rely on the precise definitions of $\text{round}(x)$ in the ambiguous cases when x is exactly midway between consecutive floating point numbers; any way of resolving the ambiguity will suffice for the validity of everything we have proved so far.

No rounding rule can be best for every application. For example, we generally want a special rule when computing our income tax. But for most numerical calculations the best policy appears to be the rounding scheme specified in Algorithm 4.2.1N, which insists that the least significant digit should always be made even (or always odd) when an ambiguous value is rounded. This is not a trivial technicality, of interest only to nit-pickers; it is an important practical consideration, since the ambiguous case arises surprisingly often and a biased rounding rule produces significantly poor results. For example, consider decimal arithmetic and assume that remainders of 5 are always rounded upwards. Then if $u = 1.0000000$ and $v = 0.55555555$ we have $u \oplus v = 1.5555556$; and if

we floating-subtract v from this result we get $u' = 1.0000001$. Adding and subtracting v from u' gives 1.0000002, and the next time we get 1.0000003, etc.; the result keeps growing although we are adding and subtracting the same value.

This phenomenon, called *drift*, will not occur when we use a stable rounding rule based on the parity of the least significant digit. More precisely:

Theorem D. $(((u \oplus v) \ominus v) \oplus v) \ominus v = (u \oplus v) \ominus v$.

For example, if $u = 1.2345679$ and $v = -0.23456785$, we find $u \oplus v = 1.0000000$, $(u \oplus v) \ominus v = 1.2345678$, $((u \oplus v) \ominus v) \oplus v = 0.99999995$, and $(((u \oplus v) \ominus v) \oplus v) \ominus v = 1.2345678$. The proof for general u and v seems to require a case analysis even more detailed than that in the above theorems; see the references at the end of this section. ∎

Theorem D is valid both for "round to even" and "round to odd"; how should we choose between these possibilities? When the radix b is odd, ambiguous cases never arise except during floating point division, and the rounding in such cases is comparatively unimportant. For *even* radices, there is reason to prefer the following rule: "Round to even when $b/2$ is odd, round to odd when $b/2$ is even." The least significant digit of a floating point fraction occurs frequently as a remainder to be rounded off in subsequent calculations, and this rule avoids generating the digit $b/2$ in the least significant position whenever possible; its effect is to provide some memory of an ambiguous rounding so that subsequent rounding will tend to be unambiguous. For example, if we were to round to odd in the decimal system, repeated rounding of the number 2.44445 to one less place each time leads to the sequence 2.4445, 2.445, 2.45, 2.5, 3; but if we round to even, such situations do not occur. [Roy A. Keir, *Inf. Proc. Letters* **3** (1975), 188–189.] On the other hand, some people prefer rounding to even in all cases, so that the remainder will tend to be 0 more often. Neither alternative conclusively dominates the other; fortunately the base is usually $b = 2$ or $b = 10$, when everyone agrees that round-to-even is best.

A reader who has checked some of the details of the above proofs will realize the immense simplification that has been afforded by the simple rule $u \oplus v = \text{round}(u + v)$. If our floating point addition routine would fail to give this result even in a few rare cases, the proofs would become enormously more complicated and perhaps they would even break down completely.

Theorem B fails if truncation arithmetic is used in place of rounding, i.e., if we let $u \oplus v = \text{trunc}(u + v)$ and $u \ominus v = \text{trunc}(u - v)$, where $\text{trunc}(x)$ takes all positive real x into the largest floating point number $\leq x$. An exception to Theorem B would then occur for cases such as $(20, +.10000001) \oplus (10, -.10000001) = (20, +.10000000)$, when the difference between $u + v$ and $u \oplus v$ cannot be expressed exactly as a floating point number; and also for cases such as $12345678 \oplus .012345678$, when it can be.

Many people feel that, since floating point arithmetic is inexact by nature, there is no harm in making it just a little bit less exact in certain rather rare cases, if it is convenient to do so. This policy saves a few cents in the design of computer

hardware, or a small percentage of the average running time of a subroutine. But the above discussion shows that such a policy is mistaken. We could save about five percent of the running time of the FADD subroutine, Program 4.2.1A, and about 25 percent of its space, if we took the liberty of rounding incorrectly in a few cases, but we are much better off leaving it as it is. The reason is not to glorify "bit chasing"; a more fundamental issue is at stake here: *Numerical subroutines should deliver results that satisfy simple, useful mathematical laws whenever possible.* The crucial formula $u \oplus v = \text{round}(u + v)$ is a "regularity" property that makes a great deal of difference between whether mathematical analysis of computational algorithms is worth doing or worth avoiding. Without any underlying symmetry properties, the job of proving interesting results becomes extremely unpleasant. *The enjoyment of one's tools is an essential ingredient of successful work.*

B. Unnormalized floating point arithmetic. The policy of normalizing all floating point numbers may be construed in two ways: We may look on it favorably by saying that it is an attempt to get the maximum possible accuracy obtainable with a given degree of precision, or we may consider it to be potentially dangerous since it tends to imply that the results are more accurate than they really are. When we normalize the result of $(1, +.31428571) \ominus (1, +.31415927)$ to $(-2, +.12644000)$, we are suppressing information about the possibly greater inaccuracy of the latter quantity. Such information would be retained if the answer were left as $(1, +.00012644)$.

The input data to a problem is frequently not known as precisely as the floating point representation allows. For example, the values of Avogadro's number and Planck's constant are not known to eight significant digits, and it might be more appropriate to denote them, respectively, by

$$(27, +.00060225) \qquad \text{and} \qquad (-23, +.00010545)$$

instead of by $(24, +.60225200)$ and $(-26, +.10545000)$. It would be nice if we could give our input data for each problem in an unnormalized form that expresses how much precison is assumed, and if the output would indicate just how much precision is known in the answer. Unfortunately, this is a terribly difficult problem, although the use of unnormalized arithmetic can help to give some indication. For example, we can say with a fair degree of certainty that the product of Avogadro's number by Planck's constant is $(0, +.00063507)$, and that their sum is $(27, +.00060225)$. (The purpose of this example is not to suggest that any important physical significance should be attached to the sum and product of these fundamental constants; the point is that it is possible to preserve a little of the information about precision in the result of calculations with imprecise quantities, when the original operands are independent of each other.)

The rules for unnormalized arithmetic are simply this: Let l_u be the number of leading zeros in the fraction part of $u = (e_u, f_u)$, so that l_u is the largest integer $\leq p$ with $|f_u| < b^{-l_u}$. Then addition and subtraction are performed

just as in Algorithm 4.2.1A, except that all scaling to the left is suppressed. Multiplication and division are performed as in Algorithm 4.2.1M, except that the answer is scaled right or left so that precisely $\max(l_u, l_v)$ leading zeros appear. Essentially the same rules have been used in manual calculation for many years.

It follows that, for unnormalized computations,

$$e_{u\oplus v},\ e_{u\ominus v} = \max(e_u, e_v) + (0 \text{ or } 1) \tag{48}$$

$$e_{u\otimes v} = e_u + e_v - q - \min(l_u, l_v) - (0 \text{ or } 1) \tag{49}$$

$$e_{u\oslash v} = e_u - e_v + q - l_u + l_v + \max(l_u, l_v) + (0 \text{ or } 1). \tag{50}$$

When the result of a calculation is zero, an unnormalized zero (often called an "order of magnitude zero") is given as the answer; this indicates that the answer may not truly be zero, we just don't know any of its significant digits.

Error analysis takes a somewhat different form with unnormalized floating point arithmetic. Let us define

$$\delta_u = \tfrac{1}{2} b^{e_u - q - p} \qquad \text{if } u = (e_u, f_u). \tag{51}$$

This quantity depends on the representation of u, not just on the value $b^{e_u-q} f_u$. Our rounding rule tells us that

$$|u \oplus v - (u + v)| \le \delta_{u\oplus v}, \qquad |u \ominus v - (u - v)| \le \delta_{u\ominus v},$$
$$|u \otimes v - (u \times v)| \le \delta_{u\otimes v}, \qquad |u \oslash v - (u \,/\, v)| \le \delta_{u\oslash v}.$$

These inequalities apply to normalized as well as unnormalized arithmetic; the main difference between the two types of error analysis is the definition of the exponent of the result of each operation (Eqs. (48) to (50)).

We have remarked that the relations $\prec$, $\sim$, $\succ$, and $\approx$ defined earlier in this section are valid and meaningful for unnormalized numbers as well as for normalized numbers. As an example of the use of these relations, let us prove an approximate associative law for unnormalized addition, analogous to (39):

$$(u \oplus v) \oplus w \approx u \oplus (v \oplus w) \quad (\epsilon), \tag{52}$$

for suitable ϵ. We have

$$\begin{aligned} |(u \oplus v) \oplus w - (u + v + w)| &\le |(u \oplus v) \oplus w - ((u \oplus v) + w)| \\ &\quad + |u \oplus v - (u + v)| \\ &\le \delta_{(u\oplus v)\oplus w} + \delta_{u\oplus v} \\ &\le 2\delta_{(u\oplus v)\oplus w}. \end{aligned}$$

A similar formula holds for $|u \oplus (v \oplus w) - (u + v + w)|$. Now since $e_{(u\oplus v)\oplus w} = \max(e_u, e_v, e_w) + (0, 1, \text{or } 2)$, we have $\delta_{(u\oplus v)\oplus w)} \le b^2 \delta_{u\oplus(v\oplus w)}$. Therefore we

find that (52) is valid when $\epsilon \geq 2b^{2-p}$; unnormalized addition is not as erratic as normalized addition with respect to the associative law.

It should be emphasized that unnormalized arithmetic is by no means a panacea. There are examples where it indicates greater accuracy than is present (e.g., addition of a great many small quantities of about the same magnitude, or evaluation of x^n for large n); and there are many more examples when it indicates poor accuracy while normalized arithmetic actually does produce good results. There is an important reason why no straightforward one-operation-at-a-time method of error analysis can be completely satisfactory, namely the fact that operands are usually not independent of each other. This means that errors tend to cancel or reinforce each other in strange ways. For example, suppose that x is approximately $\frac{1}{2}$, and suppose that we have an approximation $y = x + \delta$ with absolute error δ. If we now wish to compute $x(1-x)$, we can form $y(1-y)$; if $x = \frac{1}{2} + \epsilon$ we find $y(1-y) = x(1-x) - 2\epsilon\delta - \delta^2$, so the error has decreased substantially: it has been multiplied by a factor of $2\epsilon + \delta$. This is just one case where multiplication of imprecise quantities can lead to a quite accurate result when the operands are not independent of each other. A more obvious example is the computation of $x \ominus x$, which can be obtained with perfect accuracy regardless of how bad an approximation to x we begin with.

The extra information that unnormalized arithmetic gives us can often be more important than the information it destroys during an extended calculation, but (as usual) we must use it with care. Examples of the proper use of unnormalized arithmetic are discussed by R. L. Ashenhurst and N. Metropolis in *Computers and Computing, AMM* Slaught Memorial Papers **10** (February, 1965), 47–59; by N. Metropolis in *Numer. Math.* **7** (1965), 104–112; and by R. L. Ashenhurst in *Error in Digital Computation* **2**, ed. by L. B. Rall (New York: Wiley, 1965), 3–37. Appropriate methods for computing standard mathematical functions with both input and output in unnormalized form are given by R. L. Ashenhurst in *JACM* **11** (1964), 168–187. An extension of unnormalized arithmetic, which remembers that certain values are known to be *exact*, has been discussed by N. Metropolis in *IEEE Trans.* **C–22** (1973), 573–576.

C. Interval arithmetic. Another approach to the problem of error determination is the so-called interval or range arithmetic, in which upper and lower bounds on each number are maintained during the calculations. Thus, for example, if we know that $u_0 \leq u \leq u_1$ and $v_0 \leq v \leq v_1$, we represent this by the interval notation $u = [u_0, u_1]$, $v = [v_0, v_1]$. The sum $u \oplus v$ is $[u_0 \bigtriangledown v_0, u_1 \bigtriangleup v_1]$, where $\bigtriangledown$ denotes "lower floating point addition," the greatest representable number less than or equal to the true sum, and $\bigtriangleup$ is defined similarly (see exercise 4.2.1–13). Furthermore $u \ominus v = [u_0 \bigtriangledown v_1, u_1 \bigtriangleup v_0]$; and if u_0 and v_0 are positive, we have $u \otimes v = [u_0 \bigtriangledown v_0, u_1 \bigtriangleup v_1]$, $u \oslash v = [u_0 \bigtriangledown v_1, u_1 \bigtriangleup v_0]$. For example, we might represent Avogadro's number and Planck's constant as

$$N = [(24, +.60222400), (24, +.60228000)],$$
$$\hbar = [(-26, +.10544300), (-26, +.10545700)];$$

their sum and product would then turn out to be

$$N \oplus \hbar = [(24, +.60222400), (24, +.60228001)],$$
$$N \otimes \hbar = [(-3, +.63500305), (-3, +.63514642)].$$

If we try to divide by $[v_0, v_1]$ when $v_0 < 0 < v_1$, there is a possibility of division by zero. Since the philosophy underlying interval arithmetic is to provide rigorous error estimates, a divide-by-zero error should be signalled in this case. However, overflow and underflow need not be treated as errors in interval arithmetic, if special conventions are introduced as discussed in exercise 24.

Interval arithmetic takes only about twice as long as ordinary arithmetic, and it provides truly reliable error estimates. Considering the difficulty of mathematical error analyses, this is indeed a small price to pay. Since the intermediate values in a calculation often depend on each other, as explained above, the final estimates obtained with interval arithmetic will tend to be pessimistic; and iterative numerical methods often have to be redesigned if we want to deal with intervals. The prospects for effective use of interval arithmetic look very good, however, and efforts should be made to increase its availability.

D. History and bibliography. Jules Tannery's classic treatise on decimal calculations, *Leçons d'Arithmétique* (Paris: Colin, 1894), stated that positive numbers should be rounded upwards if the first discarded digit is 5 or more; since exactly half of the decimal digits are 5 or more, he felt that this rule would round upwards exactly half of the time, on the average, so it would produce compensating errors. The idea of "round to even" in the ambiguous cases seems to have been mentioned first by James B. Scarborough in the first edition of his pioneering book *Numerical Mathematical Analysis* (Baltimore: Johns Hopkins Press, 1930), p. 2; in the second (1950) edition he amplified his earlier remarks, stating that "It should be obvious to any thinking person that when a 5 is cut off, the preceding digit should be increased by 1 in only *half* the cases," and he recommended round-to-even in order to achieve this.

The first analysis of floating point arithmetic was given by F. L. Bauer and K. Samelson, *Zeitschrift für angewandte Math. und Physik* **4** (1953), 312–316. The next publication was not until over five years later: J. W. Carr III, *CACM* **2**, 5 (May 1959), 10–15. See also P. C. Fischer, *Proc. ACM Nat. Meeting* **13** (Urbana, Illinois, 1958), paper 39. The book *Rounding Errors in Algebraic Processes* (Englewood Cliffs: Prentice-Hall, 1963), by J. H. Wilkinson, shows how to apply error analysis of the individual arithmetic operations to the error analysis of large-scale problems; see also his treatise on *The Algebraic Eigenvalue Problem* (Oxford: Clarendon Press, 1965).

More recent work on floating point accuracy is summarized in two important papers that can be especially recommended for further study: W. M. Kahan, *Proc. IFIP Congress* (1971), **2**, 1214–1239; R. P. Brent, *IEEE Trans.* **C–22** (1973), 601–607. Both papers include useful theory and demonstrate that it pays off in practice.

The relations $\prec$, $\sim$, $\succ$, $\approx$ introduced in this section are similar to ideas published by A. van Wijngaarden in *BIT* **6** (1966), 66–81. Theorems A and B above were inspired by some related work of Ole Møller, *BIT* **5** (1965), 37–50, 251–255; Theorem C is due to T. J. Dekker, *Numer. Math.* **18** (1971), 224–242. Extensions and refinements of all three theorems have been published by S. Linnainmaa, *BIT* **14** (1974), 167–202. W. M. Kahan introduced Theorem D in some unpublished notes; for a complete proof and further commentary, see J. F. Reiser and D. E. Knuth, *Inf. Proc. Letters* **3** (1975), 84–87, 164.

Unnormalized floating point arithmetic was recommended by F. L. Bauer and K. Samelson in the article cited above, and it was independently used by J. W. Carr III at the University of Michigan in 1953. Several years later, the MANIAC III computer was designed to include both kinds of arithmetic in its hardware; see R. L. Ashenhurst and N. Metropolis, *JACM* **6** (1959), 415–428, *IEEE Trans.* **EC–12** (1963), 896–901; R. L. Ashenhurst, *Proc. Spring Joint Computer Conf.* **21** (1962), 195–202. See also H. L. Gray and C. Harrison, Jr., *Proc. Eastern Joint Computer Conf.* **16** (1959), 244–248, and W. G. Wadey, *JACM* **7** (1960), 129–139, for further early discussions of unnormalized arithmetic.

For early developments in interval arithmetic, and some modifications, see A. Gibb, *CACM* **4** (1961), 319–320; B. A. Chartres, *JACM* **13** (1966), 386–403; and the book *Interval Analysis* by Ramon E. Moore (Prentice–Hall, 1966). The subsequent flourishing of this subject is described in Moore's later book, *Methods and Applications of Interval Analysis* (SIAM, 1979).

The book *Grundlagen des Numerischen Rechnens: Mathematische Begrundung der Rechenarithmetik* by Ulrich Kulisch (Mannheim: Bibl. Inst., 1976) is entirely devoted to the study of floating point arithmetic systems; see also Kulisch's article in *IEEE Trans.* **C–26** (1977), 610–621, and his more recent book written jointly with W. L. Miranker, entitled *Computer Arithmetic in Theory and Practice* (New York: Academic Press, 1980).

EXERCISES

Note: Normalized floating point arithmetic is assumed unless the contrary is specified.

1. [*M18*] Prove that identity (7) is a consequence of (2) through (6).

2. [*M20*] Use identities (2) through (8) to prove that $(u \oplus x) \oplus (v \oplus y) \geq u \oplus v$ whenever $x \geq 0$ and $y \geq 0$.

3. [*M20*] Find eight-digit floating decimal numbers u, v, and w such that

$$u \otimes (v \otimes w) \neq (u \otimes v) \otimes w,$$

and such that no exponent overflow or underflow occurs during the computations.

4. [*10*] Is it possible to have floating point numbers u, v, and w for which exponent overflow occurs during the calculation of $u \otimes (v \otimes w)$ but not during the calculation of $(u \otimes v) \otimes w$?

5. [*M20*] Is $u \oslash v = u \otimes (1 \oslash v)$ an identity, for all floating point numbers u and $v \neq 0$ such that no exponent overflow or underflow occurs?

6. [*M22*] Are either of the following two identities valid for all floating point numbers u? (a) $0 \ominus (0 \ominus u) = u$; (b) $1 \oslash (1 \oslash u) = u$.

7. [*M21*] Let $u^{(2)}$ stand for $u \otimes u$. Find floating binary numbers u and v such that $2(u^{(2)} + v^{(2)}) < (u \oplus v)^{(2)}$.

▶ **8.** [*20*] Let $\epsilon = 0.0001$; which of the relations

$$u \prec v \quad (\epsilon), \qquad u \sim v \quad (\epsilon), \qquad u \succ v \quad (\epsilon), \qquad u \approx v \quad (\epsilon)$$

hold for the following pairs of base 10, excess 0, eight-digit floating point numbers?

a) $u = (1, +.31415927)$, $v = (1, +.31416000)$;
b) $u = (0, +.99997000)$, $v = (1, +.10000039)$;
c) $u = (24, +.60225200)$, $v = (27, +.00060225)$;
d) $u = (24, +.60225200)$, $v = (31, +.00000006)$;
e) $u = (24, +.60225200)$, $v = (32, +.00000000)$.

9. [*M22*] Prove (33), and explain why the conclusion cannot be strengthened to the relation $u \approx w\ (\epsilon_1 + \epsilon_2)$.

▶ **10.** [*M25*] (W. M. Kahan.) A certain computer performs floating point arithmetic without proper rounding, and, in fact, its floating point multiplication routine ignores all but the first p most significant digits of the $2p$-digit product $f_u f_v$. (Thus when $f_u f_v < 1/b$, the least-significant digit of $u \otimes v$ always comes out to be zero, due to subsequent normalization.) Show that this causes the monotonicity of multiplication to fail; i.e., there are positive normalized floating point numbers u, v, w such that $u < v$ but $u \otimes w > v \otimes w$.

11. [*M20*] Prove Lemma T.

12. [*M24*] Carry out the proof of Theorem B and (46) when $|e_u - e_v| \geq p$.

▶ **13.** [*M25*] Some programming languages (and even some computers) make use of floating point arithmetic only, with no provision for exact calculations with integers. If operations on integers are desired, we can, of course, represent an integer as a floating point number; and when the floating point operations satisfy the basic definitions in (9), we know that all floating point operations will be exact, provided that the operands and the answer can each be represented exactly with p significant digits. Therefore—so long as we know that the numbers aren't too large—we can add, subtract, or multiply integers with no inaccuracy due to rounding errors.

But suppose that a programmer wants to determine if m is an exact multiple of n, when m and $n \neq 0$ are integers. Suppose further that a subroutine is available to calculate the quantity $\text{round}(u \bmod 1) = u \bmod 1$ for any given floating point number u, as in exercise 4.2.1–15. One good way to determine whether or not m is a multiple of n might be to test whether or not $(m \oslash n) \bmod 1 = 0$, using the assumed subroutine; but perhaps rounding errors in the floating point calculations will invalidate this test in certain cases.

Find suitable conditions on the range of integer values $n \neq 0$ and m, such that m is a multiple of n if and only if $(m \oslash n) \bmod 1 = 0$. In other words, show that if m and n are not too large, this test is valid.

14. [*M27*] Find a suitable ϵ such that $(u \otimes v) \otimes w \approx u \otimes (v \otimes w)$ (ϵ), when *unnormalized* multiplication is being used. (This generalizes (39), since unnormalized multiplication is exactly the same as normalized multiplication when the input operands u, v, and w are normalized.)

▶ **15.** [*M24*] (H. Björk.) Does the computed midpoint of an interval always lie between the endpoints? (In other words, does $u \le v$ imply that $u \le (u \oplus v) \oslash 2 \le v$?)

16. [*M28*] (a) What is $(\cdots((x_1 \oplus x_2) \oplus x_3) \oplus \cdots \oplus x_n)$ when $n = 10^6$ and $x_k = 1.1111111$ for all k, using eight-digit floating decimal arithmetic? (b) What happens when Eq. (14) is used to calculate the standard deviation of these particular values x_k? What happens when Eqs. (15) and (16) are used instead? (c) Prove that $S_k \ge 0$ in (16), for all choices of $x_1, \ldots, x_k$.

17. [*28*] Write a MIX subroutine, FCMP, that compares the floating point number u in location ACC with the floating point number v in register A, and that sets the comparison indicator to LESS, EQUAL, or GREATER, according as $u \prec v$, $u \sim v$, or $u \succ v$ (ϵ); here ϵ is stored in location EPSILON as a nonnegative fixed point quantity with the decimal point assumed at the left of the word. Assume normalized inputs.

18. [*M40*] In unnormalized arithmetic is there a suitable number ϵ such that

$$u \otimes (v \oplus w) \approx (u \otimes v) \oplus (u \otimes w) \quad (\epsilon)?$$

▶ **19.** [*M30*] (W. M. Kahan.) Consider the following procedure for floating point summation of $x_1, \ldots, x_n$:

$$s_0 = c_0 = 0;$$
$$y_k = x_k \ominus c_{k-1}, \quad s_k = s_{k-1} \oplus y_k, \quad c_k = (s_k \ominus s_{k-1}) \ominus y_k, \qquad \text{for } 1 \le k \le n.$$

Let the relative errors in these operations be defined by the equations

$$y_k = (x_k - c_{k-1})(1 + \eta_k), \qquad s_k = (s_{k-1} + y_k)(1 + \sigma_k),$$
$$c_k = ((s_k - s_{k-1})(1 + \gamma_k) - y_k)(1 + \delta_k),$$

where $|\eta_k|, |\sigma_k|, |\gamma_k|, |\delta_k| \le \epsilon$. Prove that $s_n = \sum_{1 \le k \le n} (1 + \theta_k) x_k$, where $|\theta_k| \le 2\epsilon + O(n\epsilon^2)$. [Theorem C says that if $b = 2$ and $|s_{k-1}| \ge |y_k|$ we have $s_{k-1} + y_k = s_k - c_k$ exactly. But in this exercise we want to obtain an estimate that is valid *even when floating point operations are not carefully rounded*, assuming only that each operation has bounded relative error.]

20. [*25*] (S. Linnainmaa.) Find all u, v for which $|u| \ge |v|$ and (47) fails.

21. [*M35*] (T. J. Dekker.) Theorem C shows how to do exact addition of floating binary numbers. Explain how to do *exact multiplication*: Express the product uv in the form $w + w'$, where w and w' are computed from two given floating binary numbers u and v, using only the operations $\oplus$, $\ominus$, and $\otimes$.

22. [*M30*] Can drift occur in floating point multiplication/division? Consider the sequence $x_0 = u$, $x_{2n+1} = x_{2n} \otimes v$, $x_{2n+2} = x_{2n+1} \oslash v$, given u and v; what is the largest subscript k such that $x_k \ne x_{k+2}$ is possible?

▶ **23.** [*M26*] Prove or disprove: $u \ominus (u \bmod 1) = \lfloor u \rfloor$, for all floating point u.

24. [*M27*] Consider the set of all intervals $[u_l, u_r]$, where u_l and u_r are either nonzero floating point numbers or the special symbols $+0$, -0, $+\infty$, $-\infty$; each interval must have $u_l \le u_r$, and $u_l = u_r$ is allowed only when u_l is finite and nonzero. The interval $[u_l, u_r]$ stands for all floating point x such that $u_l \le x \le u_r$, where we regard

$$-\infty < -x < -0 < +0 < +x < +\infty$$

for all positive x. (Thus, $[1, 2]$ means $1 \le x \le 2$; $[+0, 1]$ means $0 < x \le 1$; $[-0, 1]$ means $0 \le x \le 1$; $[-0, +0]$ denotes the single value 0; and $[-\infty, +\infty]$ stands for everything.) Show how to define appropriate arithmetic operations on all such intervals, without resorting to "overflow" or "underflow" or other anomalous indications except when dividing by an interval that includes zero.

▶ **25.** [*15*] When people speak about inaccuracy in floating point arithmetic they often ascribe errors to "cancellation" that occurs during the subtraction of nearly equal quantities. But when u and v are approximately equal, the difference $u \ominus v$ is obtained exactly, with no error. What do these people really mean?

26. [*HM30*] (H. G. Diamond.) Suppose $f(x)$ is a strictly increasing function on some interval $[x_0, x_1]$, and let $g(x)$ be the inverse function. (For example, f and g might be "exp" and "ln", or "tan" and "arctan".) If x is a floating point number such that $x_0 \le x \le x_1$, let $\hat{f}(x) = \text{round}(f(x))$, and if y is a floating point number such that $f(x_0) \le y \le f(x_1)$, let $\hat{g}(y) = \text{round}(g(y))$; furthermore, let $h(x) = \hat{g}(\hat{f}(x))$, whenever this is defined. Although $h(x)$ won't always be equal to x, due to rounding, we expect $h(x)$ to be "near" x.

Prove that if the precision b^p is at least 3, and if f is strictly concave or strictly convex (i.e., $f''(x) < 0$ or $f''(x) > 0$ for all x in $[x_0, x_1]$), then repeated application of h will be *stable* in the sense that

$$h(h(h(x))) = h(h(x)),$$

for all x such that both sides of this equation are defined. In other words, there will be no "drift" if the subroutines are properly implemented.

▶ **27.** [*M25*] (W. M. Kahan.) Let $f(x) = 1 + x + \cdots + x^{106} = (1 - x^{107})/(1 - x)$ for $x < 1$, and let $g(y) = f((\frac{1}{3} - y^2)(3 + 3.45y^2))$ for $0 < y < 1$. Evaluate $g(y)$ on one or more "pocket calculators," for $y = 10^{-3}$, 10^{-4}, 10^{-5}, 10^{-6}, and explain all inaccuracies in the results you obtain. (Since most present-day calculators do not round correctly, the results are often surprising. Note that $g(\epsilon) = 107 - 10491.35\epsilon^2 + O(\epsilon^4)$.)

*4.2.3. Double-Precision Calculations

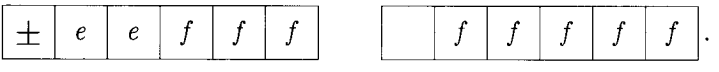
Up to now we have considered "single-precision" floating point arithmetic, which essentially means that the floating point values we have dealt with can be stored in a single machine word. When single-precision floating point arithmetic does not yield sufficient accuracy for a given application, the precision can be increased by suitable programming techniques that use two or more words of memory to represent each number.

Although we shall discuss the general question of high-precision calculations in Section 4.3, it is appropriate to give a separate discussion of double-precision

here. Special techniques apply to double precision that are comparatively inappropriate for higher precisions; and double precision is a reasonably important topic in its own right, since it is the first step beyond single precision and it is applicable to many problems that do not require extremely high precision.

Double-precision calculations are almost always required for floating point rather than fixed point arithmetic, except perhaps in statistical work where fixed point double-precision is commonly used to calculate sums of squares and cross products; since fixed point versions of double-precision arithmetic are simpler than floating point versions, we shall confine our discussion here to the latter.

Double precision is quite frequently desired not only to extend the precision of the fraction parts of floating point numbers, but also to increase the range of the exponent part. Thus we shall deal in this section with the following two-word format for double-precision floating point numbers in the MIX computer:



(1)

Here two bytes are used for the exponent and eight bytes are used for the fraction. The exponent is "excess $b^2/2$," where b is the byte size. The sign will appear in the most significant word; it is convenient to ignore the sign of the other word completely.

Our discussion of double-precision arithmetic will be quite machine-oriented, because it is only by studying the problems involved in coding these routines that a person can properly appreciate the subject. A careful study of the MIX programs below is therefore essential to the understanding of the material.

In this section we shall depart from the idealistic goals of accuracy stated in the previous two sections; our double-precision routines will *not* round their results, and a little bit of error will sometimes be allowed to creep in. Users dare not trust these routines too much. There was ample reason to squeeze out every possible drop of accuracy in the single-precision case, but now we face a different situation: (a) The extra programming required to ensure true double-precision rounding in all cases is considerable; fully accurate routines would take, say, twice as much space and half again as much more time. It was comparatively easy to make our single-precision routines perfect, but double precision brings us face to face with our machine's limitations. (A similar situation occurs with respect to other floating point subroutines; we can't expect the cosine routine to compute round($\cos x$) exactly for all x, since that turns out to be virtually impossible. Instead, the cosine routine should provide the best relative error it can achieve with reasonable speed, for all reasonable values of x. Of course, the designer of the routine should try to make the computed function satisfy simple mathematical laws whenever possible (e.g., $\textcircled{\cos}(-x) = \textcircled{\cos}\, x$, $|\textcircled{\cos}\, x| \le 1$, and $\textcircled{\cos}\, x \ge \textcircled{\cos}\, y$ for $0 \le x \le y < \pi$).) (b) Single-precision arithmetic is a "staple food" that everybody who wants to employ floating point arithmetic must use, but double precision is usually for situations where such clean results aren't as important. The difference between seven- and eight-place accuracy can be noticeable, but we rarely care about the difference between 15- and 16-place

accuracy. Double precision is most often used for intermediate steps during the calculation of single-precision results; its full potential isn't needed. (c) It will be instructive for us to analyze these procedures in order to see how inaccurate they can be, since they typify the types of short cuts generally taken in bad single-precision routines (see exercises 7 and 8).

Let us now consider addition and subtraction operations from this standpoint. Subtraction is, of course, converted to addition by changing the sign of the second operand. Addition is performed by separately adding together the least-significant halves and the most-significant halves, propagating "carries" appropriately.

A difficulty arises, however, since we are doing signed-magnitude arithmetic: it is possible to add the least-significant halves and to get the wrong sign (namely, when the signs of the operands are opposite and the least-significant half of the smaller operand is bigger than the least-significant half of the larger operand). The simplest solution is to anticipate the correct sign; so in step A2 (cf. Algorithm 4.2.1A), we not only assume that $e_u \geq e_v$, we also assume that $|u| \geq |v|$. This means we can be sure that the final sign will be the sign of u. In other respects, double-precision addition is very much like its single-precision counterpart, only everything is done twice.

Program A (*Double-precision addition*). The subroutine DFADD adds a double-precision floating point number v, having the form (1), to a double-precision floating point number u, assuming that v is initially in rAX (i.e., registers A and X), and that u is initially stored in locations ACC and ACCX. The answer appears both in rAX and in (ACC, ACCX). The subroutine DFSUB subtracts v from u under the same conventions.

Both input operands are assumed to be normalized, and the answer is normalized. The last portion of this program is a double-precision normalization procedure that is used by other subroutines of this section. Exercise 5 shows how to improve the program significantly.

```
01  ABS     EQU   1:5          Field definition for absolute value
02  SIGN    EQU   0:0          Field definition for sign
03  EXPD    EQU   1:2          Double-precision exponent field
04  DFSUB   STA   TEMP         Double-precision subtraction:
05          LDAN  TEMP         Change sign of v.
06  DFADD   STJ   EXITDF       Double-precision addition:
07          CMPA  ACC(ABS)     Compare |u| with |v|.
08          JG    1F
09          JL    2F
10          CMPX  ACCX(ABS)
11          JLE   2F
12  1H      STA   ARG          If |u| < |v|, interchange u ↔ v.
13          STX   ARGX
14          LDA   ACC
15          LDX   ACCX
16          ENT1  ACC          (ACC and ACCX are in consecutive
17          MOVE  ARG(2)           locations.)
```

```
18  2H      STA   TEMP          Now ACC has the sign of the answer.
19          LD1N  TEMP(EXPD)    rI1 ← −e_v.
20          LD2   ACC(EXPD)     rI2 ← e_u.
21          INC1  0,2           rI1 ← e_u − e_v.
22          SLAX  2             Remove exponent.
23          SRAX  1,1           Scale right.
24          STA   ARG           0 v1 v2 v3 v4
25          STX   ARGX                    v5 v6 v7 v8 v9
26          STA   ARGX(SIGN)    Store true sign in both halves.
27          LDA   ACC
28          LDX   ACCX
29          SLAX  2             Remove exponent.
30          STA   ACC           u1 u2 u3 u4 u5
31          SLAX  4
32          ENTX  1
33          STX   EXPO          EXPO ← 1 (see below).
34          SRC   1             1 u5 u6 u7 u8
35          STA   1F(SIGN)      A trick, see comments in text.
36          ADD   ARGX(0:4)     Add 0 v5 v6 v7 v8.
37          SRAX  4
38  1H      DECA  1             Recover from inserted 1. (Sign varies)
39          ADD   ACC(0:4)      Add most significant halves.
40          ADD   ARG           (Overflow cannot occur)
41  DNORM   JANZ  1F            Normalization routine:
42          JXNZ  1F            f_w in rAX, e_w = EXPO + rI2.
43  DZERO   STA   ACC           If f_w = 0, set e_w ← 0.
44          JMP   9F
45  2H      SLAX  1             Normalize to left.
46          DEC2  1
47  1H      CMPA  =0=(1:1)      Is the leading byte zero?
48          JE    2B
49          SRAX  2             (Rounding omitted)
50          STA   ACC
51          LDA   EXPO          Compute final exponent.
52          INCA  0,2
53          JAN   EXPUND        Is it negative?
54          STA   ACC(EXPD)
55          CMPA  =1(3:3)=      Is it more than two bytes?
56          JL    8F
57  EXPOVD  HLT   20
58  EXPUND  HLT   10
59  8H      LDA   ACC           Bring answer into rA.
60  9H      STX   ACCX
61  EXITDF  JMP   *             Exit from subroutine.
62  ARG     CON   0
63  ARGX    CON   0
64  ACC     CON   0             floating point accumulator
65  ACCX    CON   0
66  EXPO    CON   0             Part of "raw exponent"  ∎
```

When the least-significant halves are added together in this program, an extra digit "1" is inserted at the left of the word that is known to have the correct sign. After the addition, this byte can be 0, 1, or 2, depending on the circumstances, and all three cases are handled simultaneously in this way. (Compare this with the rather cumbersome method of complementation that is used in Program 4.2.1A.)

It is worth noting that register A can be zero after the instruction on line 40 has been performed; and, because of the way MIX defines the sign of a zero result, the accumulator contains the correct sign that is to be attached to the result if register X is nonzero. If lines 39 and 40 were interchanged, the program would be incorrect, even though both instructions are "ADD"!

Now let us consider double-precision multiplication. The product has four components, shown schematically in Fig. 4. Since we need only the leftmost eight bytes, it is convenient to work only with the digits to the left of the vertical line in the diagram, and this means in particular that we need not even compute the product of the two least-significant halves.

Program M (*Double-precision multiplication*). The input and output conventions for this subroutine are the same as for Program A.

```
01  BYTE   EQU  1(4:4)          Byte size
02  QQ     EQU  BYTE*BYTE/2     Excess of double-precision exponent
03  DFMUL  STJ  EXITDF          Double-precision multiplication:
04         STA  TEMP
05         SLAX 2               Remove exponent.
06         STA  ARG             v_m
07         STX  ARGX            v_l
08         LDA  TEMP(EXPD)
09         ADD  ACC(EXPD)
10         STA  EXPO            EXPO ← e_u + e_v.
11         ENT2 -QQ             rI2 ← −QQ.
12         LDA  ACC
13         LDX  ACCX
14         SLAX 2               Remove exponent.
15         STA  ACC             u_m
16         STX  ACCX            u_l
17         MUL  ARGX            u_m × v_l
18         STA  TEMP
19         LDA  ARG(ABS)
20         MUL  ACCX(ABS)       |v_m × u_l|
21         SRA  1               0 x x x x
22         ADD  TEMP(1:4)       (Overflow cannot occur)
23         STA  TEMP
24         LDA  ARG
25         MUL  ACC             v_m × u_m
26         STA  TEMP(SIGN)      Store true sign of result.
27         STA  ACC             Now prepare to add all the
28         STX  ACCX                 partial products together.
```

			$u\ u\ u\ u\ u$	$u\ u\ u\ 0\ 0$	$= u_m + \epsilon u_l$
			$v\ v\ v\ v\ v$	$v\ v\ v\ 0\ 0$	$= v_m + \epsilon v_l$
			$x\ x\ x\ x\ x$	$x\ 0\ 0\ 0\ 0$	$= \epsilon^2 u_l \times v_l$
	$x\ x\ x\ x$	x	$x\ x\ x\ 0\ 0$		$= \epsilon u_m \times v_l$
	$x\ x\ x\ x$	x	$x\ x\ x\ 0\ 0$		$= \epsilon u_l \times v_m$
$x\ x\ x\ x\ x$	$x\ x\ x\ x$	x			$= u_m \times v_m$
$w\ w\ w\ w\ w$	$w\ w\ w\ w$	w	$w\ w\ w\ w\ w$	$w\ 0\ 0\ 0\ 0$	

Fig. 4. Double-precision multiplication of eight-byte fraction parts.

```
         LDA  ACCX(0:4)      0 x x x x
         ADD  TEMP           (Overflow cannot occur)
         SRAX 4
         ADD  ACC            (Overflow cannot occur)
         JMP  DNORM          Normalize and exit.  ▮
```

Note the careful treatment of signs in this program, and note also the fact that the range of exponents makes it impossible to compute the final exponent using an index register. Program M is perhaps too slipshod in accuracy, since it throws away all the information to the right of the vertical line in Fig. 4; this can make the least significant byte as much as 2 in error. A little more accuracy can be achieved as discussed in exercise 4.

Double-precision floating division is the most difficult routine, or at least the most frightening prospect we have encountered so far in this chapter. Actually, it is not terribly complicated, once we see how to do it; let us write the numbers to be divided in the form $(u_m + \epsilon u_l)/(v_m + \epsilon v_l)$, where ϵ is the reciprocal of the word size of the computer, and where v_m is assumed to be normalized. The fraction can now be expanded as follows:

$$\begin{aligned}\frac{u_m + \epsilon u_l}{v_m + \epsilon v_l} &= \frac{u_m + \epsilon u_l}{v_m}\left(\frac{1}{1 + \epsilon(v_l/v_m)}\right)\\ &= \frac{u_m + \epsilon u_l}{v_m}\left(1 - \epsilon\left(\frac{v_l}{v_m}\right) + \epsilon^2\left(\frac{v_l}{v_m}\right)^2 - \cdots\right).\end{aligned} \qquad (2)$$

Since $0 \le |v_l| < 1$ and $1/b \le |v_m| < 1$, we have $|v_l/v_m| < b$, and the error from dropping terms involving ϵ^2 can be disregarded. Our method therefore is to compute $w_m + \epsilon w_l = (u_m + \epsilon u_l)/v_m$, and then to subtract ϵ times $w_m v_l/v_m$ from the result.

In the following program, lines 27–32 do the lower half of a double-precision addition, using another method for forcing the appropriate sign as an alternative to the trick of Program A.

Program D (*Double-precision division*). This program adheres to the same conventions as Programs A and M.

```
01 DFDIV  STJ  EXITDF          Double-precision division:
02        JOV  OFLO            Ensure overflow is off.
03        STA  TEMP
04        SLAX 2               Remove exponent.
05        STA  ARG             v_m
06        STX  ARGX            v_l
07        LDA  ACC(EXPD)
08        SUB  TEMP(EXPD)
09        STA  EXPO            EXPO ← e_u − e_v.
10        ENT2 QQ+1            rI2 ← QQ + 1.
11        LDA  ACC
12        LDX  ACCX
13        SLAX 2               Remove exponent.
14        SRAX 1               (Cf. Algorithm 4.2.1M)
15        DIV  ARG             If overflow, it is detected below.
16        STA  ACC             w_m
17        SLAX 5               Use remainder in further division.
18        DIV  ARG
19        STA  ACCX            ±w_l
20        LDA  ARGX(1:4)
21        ENTX 0
22        DIV  ARG(ABS)        rA ← ⌊|b^4 v_l/v_m|⌋/b^5.
23        JOV  DVZROD          Did division cause overflow?
24        MUL  ACC(ABS)        rAX ← |w_m v_l/b v_m|, approximately.
25        SRAX 4               Multiply by b, and save
26        SLC  5                   the leading byte in rX.
27        SUB  ACCX(ABS)       Subtract |w_l|.
28        DECA 1               Force minus sign.
29        SUB  WM1
30        JOV  *+2             If no overflow, carry one more
31        INCX 1                   to upper half.
32        SLC  5               (Now rA ≤ 0)
33        ADD  ACC(ABS)        rA ← |w_m| − |rA|.
34        STA  ACC(ABS)        (Now rA ≥ 0)
35        LDA  ACC             w_m with correct sign
36        JMP  DNORM           Normalize and exit.
37 DVZROD HLT  30              Unnormalized or zero divisor
38 1H     EQU  1(1:1)
39 WM1    CON  1B-1,BYTE-1(1:1) Word size minus one ∎
```

Here is a table of the approximate average computation times for these double-precision subroutines, compared to the single-precision subroutines that appear in Section 4.2.1:

	Single precision	Double precision
Addition	$45.5u$	$84u$
Subtraction	$49.5u$	$88u$
Multiplication	$48u$	$109u$
Division	$52u$	$126.5u$

For extension of the methods of this section to triple-precision floating point fraction parts, see Y. Ikebe, *CACM* 8 (1965), 175–177.

EXERCISES

1. [*16*] Try the double-precision division technique by hand, with $\epsilon = \frac{1}{1000}$, when dividing 180000 by 314159. (Thus, let $(u_m, u_l) = (.180, .000)$ and $(v_m, v_l) = (.314, .159)$, and find the quotient using the method suggested in the text following (2).)

2. [*20*] Would it be a good idea to insert the instruction "ENTX 0" between lines 30 and 31 of Program M, in order to keep unwanted information left over in register X from interfering with the accuracy of the results?

3. [*M20*] Explain why overflow cannot occur during Program M.

4. [*22*] How should Program M be changed so that extra accuracy is achieved, essentially by moving the vertical line in Fig. 4 over to the right one position? Specify all changes that are required, and determine the difference in execution time caused by these changes.

▸ **5.** [*24*] How should Program A be changed so that extra accuracy is achieved, essentially by working with a nine-byte accumulator instead of an eight-byte accumulator to the right of the decimal point? Specify all changes that are required, and determine the difference in execution time caused by these changes.

6. [*23*] Assume that the double-precision subroutines of this section and the single-precision subroutines of Section 4.2.1 are being used in the same main program. Write a subroutine that converts a single-precision floating point number into double-precision form (1), and write another subroutine that converts a double-precision floating point number into single-precision form (reporting exponent overflow or underflow if the conversion is impossible).

▸ **7.** [*M30*] Estimate the accuracy of the double-precision subroutines in this section, by finding bounds δ_1, δ_2, and δ_3 on the relative errors

$$\left|((u \oplus v) - (u + v))/(u + v)\right|, \qquad \left|((u \otimes v) - (u \times v))/(u \times v)\right|,$$
$$\left|((u \oslash v) - (u/v))/(u/v)\right|.$$

8. [*M28*] Estimate the accuracy of the "improved" double-precision subroutines of exercises 4 and 5, in the sense of exercise 7.

9. [*M42*] T. J. Dekker [*Numer. Math.* **18** (1971), 224–242] has suggested an alternative approach to double precision, based entirely on single-precision floating binary calculations. For example, Theorem 4.2.2C states that $u+v = w+r$, where $w = u \oplus v$ and $r = (u \ominus w) \oplus v$, if $|u| \geq |v|$ and the radix is 2; here $|r| \leq |w|/2^p$, so the pair (w, r) may be considered a double-precision version of $u + v$. To add two such pairs $(u, u') \oplus (v, v')$, where $|u'| \leq |u|/2^p$ and $|v'| \leq |v|/2^p$ and $|u| \geq |v|$, Dekker suggests computing $u+v = w+r$ (exactly), then $s = (r \oplus v') \oplus u'$ (an approximate remainder), and finally returning the value $(w \oplus s, (w \ominus (w \oplus s)) \oplus s)$.

Study the accuracy and efficiency of this approach when it is used recursively to produce quadruple-precision calculations.

Table 1
EMPIRICAL DATA FOR OPERAND ALIGNMENTS BEFORE ADDITION

$\lvert e_u - e_v \rvert$	$b = 2$	$b = 10$	$b = 16$	$b = 64$
0	0.33	0.47	0.47	0.56
1	0.12	0.23	0.26	0.27
2	0.09	0.11	0.10	0.04
3	0.07	0.03	0.02	0.02
4	0.07	0.01	0.01	0.02
5	0.04	0.01	0.02	0.00
over 5	0.28	0.13	0.11	0.09
average	3.1	0.9	0.8	0.5

4.2.4. Distribution of Floating Point Numbers

In order to analyze the average behavior of floating point arithmetic algorithms (and in particular to determine their average running time), we need some statistical information that allows us to determine how often various cases arise. The purpose of this section is to discuss the empirical and theoretical properties of the distribution of floating point numbers.

A. Addition and subtraction routines. The execution time for a floating point addition or subtraction depends largely on the initial difference of exponents, and also on the number of normalization steps required (to the left or to the right). No way is known to give a good theoretical model that tells what characteristics to expect, but extensive empirical investigations have been made by D. W. Sweeney [*IBM Systems J.* 4 (1965), 31–42].

By means of a special tracing routine, Sweeney ran six "typical" large-scale numerical programs, selected from several different computing laboratories, and examined each floating addition or subtraction operation very carefully. Over 250,000 floating point addition-subtractions were involved in gathering this data. About one out of every ten instructions executed by the tested programs was either FADD or FSUB.

Let us consider subtraction to be addition preceded by negating the second operand; therefore we may give all the statistics as if we were merely doing addition. Sweeney's results can be summarized as follows:

One of the two operands to be added was found to be equal to zero about 9 percent of the time, and this was usually the accumulator (ACC). The other 91 percent of the cases split about equally between operands of the same or of opposite signs, and about equally between cases where $|u| \leq |v|$ or $|v| \leq |u|$. The computed answer was zero about 1.4 percent of the time.

The difference between exponents had a behavior approximately given by the probabilities shown in Table 1, for various radices b. (The "over 5" line of that table includes essentially all of the cases when one operand was zero, but the "average" line does not include these cases.)

Table 2
EMPIRICAL DATA FOR NORMALIZATION AFTER ADDITION

	$b = 2$	$b = 10$	$b = 16$	$b = 64$
Shift right 1	0.20	0.07	0.06	0.03
No shift	0.59	0.80	0.82	0.87
Shift left 1	0.07	0.08	0.07	0.06
Shift left 2	0.03	0.02	0.01	0.01
Shift left 3	0.02	0.00	0.01	0.00
Shift left 4	0.02	0.01	0.00	0.01
Shift left >4	0.06	0.02	0.02	0.02

When u and v have the same sign and are normalized, then $u + v$ either requires one shift to the *right* (for fraction overflow), or no normalization shifts whatever. When u and v have opposite signs, we have zero or more *left* shifts during the normalization. Table 2 gives the observed number of shifts required; the last line of that table includes all cases where the result was zero. The average number of left shifts per normalization was about 0.9 when $b = 2$; about 0.2 when $b = 10$ or 16; and about 0.1 when $b = 64$.

B. The fraction parts. Further analysis of floating point routines can be based on the *statistical distribution of the fraction parts* of randomly chosen normalized floating point numbers. In this case the facts are quite surprising, and there is an interesting theory that accounts for the unusual phenomena that are observed.

For convenience let us temporarily assume that we are dealing with floating *decimal* (i.e., radix 10) arithmetic; modifications of the following discussion to any other positive integer base b will be very straightforward. Suppose we are given a "random" positive normalized number $(e, f) = 10^e \cdot f$. Since f is normalized, we know that its leading digit is 1, 2, 3, 4, 5, 6, 7, 8, or 9, and it seems natural to assume that each of these nine possible leading digits will occur about one-ninth of the time. But, in fact, the behavior in practice is quite different. For example, the leading digit tends to be equal to 1 over 30 percent of the time!

One way to test the assertion just made is to take a table of physical constants (e.g., the speed of light, the acceleration of gravity) from some standard reference. If we look at the *Handbook of Mathematical Functions* (U.S. Dept of Commerce, 1964), for example, we find that 8 of the 28 different physical constants given in Table 2.3, roughly 29 percent, have leading digit equal to 1. The decimal values of $n!$ for $1 \le n \le 100$ include exactly 30 entries beginning with 1; so do the decimal values of 2^n and of F_n, for $1 \le n \le 100$. We might also try looking at census reports, or a Farmer's Almanack (but not a telephone directory).

In the days before pocket calculators, the pages in well-used tables of logarithms tended to get quite dirty in the front, while the last pages stayed relatively clean and neat. This phenomenon was apparently first mentioned in print by the American astronomer Simon Newcomb [*Amer. J. Math.* **4** (1881), 39–40], who

gave good grounds for believing that the leading digit d occurs with probability $\log_{10}(1 + 1/d)$. The same distribution was discovered empirically, many years later, by Frank Benford, who reported the results of 20,229 observations taken from different sources [*Proc. Amer. Philosophical Soc.* **78** (1938), 551–572].

In order to account for this leading-digit law, let's take a closer look at the way we write numbers in floating point notation. If we take any positive number u, its leading digits are determined by the value $(\log_{10} u) \bmod 1$: The leading digit is less than d if and only if

$$(\log_{10} u) \bmod 1 < \log_{10} d, \tag{1}$$

since $10f_u = 10^{(\log_{10} u) \bmod 1}$.

Now if we have a "random" positive number U, chosen from some reasonable distribution that might occur in nature, we might expect that $(\log_{10} U) \bmod 1$ would be uniformly distributed between zero and one, at least to a very good approximation. (Similarly, we expect $U \bmod 1$, $U^2 \bmod 1$, $\sqrt{U + \pi} \bmod 1$, etc., to be uniformly distributed. We expect a roulette wheel to be unbiased, for essentially the same reason.) Therefore by (1) the leading digit will be 1 with probability $\log_{10} 2 \approx 30.103$ percent; it will be 2 with probability $\log_{10} 3 - \log_{10} 2 \approx 17.609$ percent; and, in general, if r is any real value between 1 and 10, we ought to have $10f_U \le r$ approximately $\log_{10} r$ of the time.

Another way to explain this law is to say that a random value U should appear at a random point on a slide rule, according to the uniform distribution, since the distance from the left end of a slide rule to the position of U is proportional to $(\log_{10} U) \bmod 1$. The analogy between slide rules and floating point calculation is very close when multiplication and division are being considered.

The fact that leading digits tend to be small is important to keep in mind; it makes the most obvious techniques of "average error" estimation for floating point calculations invalid. The relative error due to rounding is usually a little more than expected.

Of course, it may justly be said that the heuristic argument above does not prove the stated law. It merely shows us a plausible reason why the leading digits behave the way they do. An interesting approach to the analysis of leading digits has been suggested by R. Hamming: Let $p(r)$ be the probability that $10f_U \le r$, where $1 \le r \le 10$ and f_U is the normalized fraction part of a random normalized floating point number U. If we think of random quantities in the real world, we observe that they are measured in terms of arbitrary units; and if we were to change the definition of a meter or a gram, many of the fundamental physical constants would have different values. Suppose then that all of the numbers in the universe are suddenly multiplied by a constant factor c; our universe of random floating point quantities should be essentially unchanged by this transformation, so $p(r)$ should not be affected.

Multiplying everything by c has the effect of transforming $(\log_{10} U) \bmod 1$ into $(\log_{10} U + \log_{10} c) \bmod 1$. It is now time to set up formulas that describe the desired behavior; we may assume that $1 \le c \le 10$. By definition,

$$p(r) = \text{probability that } (\log_{10} U) \bmod 1 \le \log_{10} r.$$

By our assumption, we should also have

$$\begin{aligned} p(r) &= \text{probability that } (\log_{10} U + \log_{10} c) \bmod 1 \le \log_{10} r \\ &= \begin{cases} \text{probability that } (\log_{10} U \bmod 1) \le \log_{10} r - \log_{10} c \\ \qquad \text{or } (\log_{10} U \bmod 1) \ge 1 - \log_{10} c, & \text{if } c \le r; \\ \text{probability that } (\log_{10} U \bmod 1) \le \log_{10} r + 1 - \log_{10} c \\ \qquad \text{and } (\log_{10} U \bmod 1) \ge 1 - \log_{10} c, & \text{if } c \ge r; \end{cases} \\ &= \begin{cases} p(r/c) + 1 - p(10/c), & \text{if } c \le r; \\ p(10r/c) - p(10/c), & \text{if } c \ge r. \end{cases} \end{aligned} \tag{2}$$

Let us now extend the function $p(r)$ to values outside the range $1 \le r \le 10$, by defining $p(10^n r) = p(r) + n$; then if we replace $10/c$ by d, the last equation of (2) may be written

$$p(rd) = p(r) + p(d). \tag{3}$$

If our assumption about invariance of the distribution under multiplication by a constant factor is valid, then Eq. (3) must hold for all $r > 0$ and $1 \le d \le 10$. The facts that $p(1) = 0$, $p(10) = 1$ now imply that

$$1 = p(10) = p\big((\sqrt[n]{10})^n\big) = p(\sqrt[n]{10}) + p\big((\sqrt[n]{10})^{n-1}\big) = \cdots = np(\sqrt[n]{10});$$

hence we deduce that $p(10^{m/n}) = m/n$ for all positive integers m and n. If we now decide to require that p is continuous, we are forced to conclude that $p(r) = \log_{10} r$, and this is the desired law.

Although this argument may be more convincing than the first one, it doesn't really hold up under scrutiny if we stick to conventional notions of probability. The traditional way to make the above argument rigorous is to assume that there is some underlying distribution of numbers $F(u)$ such that a given positive number U is $\le u$ with probability $F(u)$; then the probability of concern to us is

$$p(r) = \sum_m \big(F(10^m r) - F(10^m)\big), \tag{4}$$

summed over all values $-\infty < m < \infty$. Our assumptions about scale invariance and continuity have led us to conclude that

$$p(r) = \log_{10} r.$$

Using the same argument, we could "prove" that

$$\sum_m \big(F(b^m r) - F(b^m)\big) = \log_b r, \tag{5}$$

for each integer $b \ge 2$, when $1 \le r \le b$. But there *is* no distribution function F that satisfies this equation for all such b and r! (See exercise 7.)

One way out of the difficulty is to regard the logarithm law $p(r) = \log_{10} r$ as only a very close *approximation* to the true distribution. The true distribution itself may perhaps be changing as the universe expands, becoming a better and better approximation as time goes on; and if we replace 10 by an arbitrary base b, the approximation might be less accurate (at any given time) as b gets larger. Another rather appealing way to resolve the dilemma, by abandoning the traditional idea of a distribution function, has been suggested by R. A. Raimi, *AMM* **76** (1969), 342–348.

The hedging in the last paragraph is probably a very unsatisfactory explanation, and so the following further calculation (which sticks to rigorous mathematics and avoids any intuitive, yet paradoxical, notions of probability) should be welcome. Let us consider the distribution of the leading digits of the *positive integers*, instead of the distribution for some imagined set of real numbers. The investigation of this topic is quite interesting, not only because it sheds some light on the probability distributions of floating point data, but also because it makes a particularly instructive example of how to combine the methods of discrete mathematics with the methods of infinitesimal calculus.

In the following discussion, let r be a fixed real number, $1 \le r \le 10$; we will attempt to make a reasonable definition of $p(r)$, the "probability" that the representation $10^{e_N} \cdot f_N$ of a "random" positive integer N has $10f_N < r$, assuming infinite precision.

To start, let us try to find the probability using a limiting method like the definition of "Pr" in Section 3.5. One nice way to rephrase that definition is to define

$$P_0(n) = \begin{cases} 1, & \text{if } n = 10^e \cdot f \text{ where } 10f < r, \\ & \quad \text{i.e., if } (\log_{10} n) \bmod 1 < \log_{10} r; \\ 0, & \text{otherwise.} \end{cases} \tag{6}$$

Now $P_0(1)$, $P_0(2)$, ... is an infinite sequence of zeros and ones, with ones to represent the cases that contribute to the probability we are seeking. We can try to "average out" this sequence, by defining

$$P_1(n) = \frac{1}{n} \sum_{1 \le k \le n} P_0(k). \tag{7}$$

Thus if we generate a random integer between 1 and n using the techniques of Chapter 3, and convert it to floating decimal form (e, f), the probability that $10f < r$ is exactly $P_1(n)$. It is natural to let $\lim_{n\to\infty} P_1(n)$ be the "probability" $p(r)$ we are after, and that is just what we did in Section 3.5.

But in this case the limit does not exist: For example, let us consider the subsequence

$$P_1(s),\ P_1(10s),\ P_1(100s),\ \ldots,\ P_1(10^n s),\ \ldots,$$

where s is a real number, $1 \le s \le 10$. If $s \le r$, we find that

$$\begin{aligned} P_1(10^n s) &= \frac{1}{10^n s}\big(\lceil r\rceil - 1 + \lceil 10r\rceil - 10 + \cdots + \lceil 10^{n-1}r\rceil \\ &\qquad - 10^{n-1} + \lfloor 10^n s\rfloor + 1 - 10^n\big) \\ &= \frac{1}{10^n s}\big(r(1 + 10 + \cdots + 10^{n-1}) + O(n) \\ &\qquad + \lfloor 10^n s\rfloor - 1 - 10 - \cdots - 10^n\big) \\ &= \frac{1}{10^n s}\big(\tfrac{1}{9}(10^n r - 10^{n+1}) + \lfloor 10^n s\rfloor + O(n)\big). \end{aligned} \tag{8}$$

As $n \to \infty$, $P_1(10^n s)$ therefore approaches the limiting value $1 + (r - 10)/9s$. The above calculation for the case $s \le r$ can be modified so that it is valid for $s > r$ if we replace $\lfloor 10^n s\rfloor + 1$ by $\lceil 10^n r\rceil$; when $s \ge r$, we therefore obtain the limiting value $10(r - 1)/9s$. [See J. Franel, *Naturforschende Gesellschaft, Vierteljahrsschrift* **62** (Zürich, 1917), 286–295.]

In other words, the sequence $\langle P_1(n)\rangle$ has subsequences $\langle P_1(10^n s)\rangle$ whose limit goes from $(r - 1)/9$ up to $10(r - 1)/9r$ and down again to $(r - 1)/9$, as s goes from 1 to r to 10. We see that $P_1(n)$ has no limit as $n \to \infty$; and the values of $P_1(n)$ for large n are not particularly good approximations to our conjectured limit $\log_{10} r$ either!

Since $P_1(n)$ doesn't approach a limit, we can try to use the same idea as (7) once again, to "average out" the anomalous behavior. In general, let

$$P_{m+1}(n) = \frac{1}{n}\sum_{1 \le k \le n} P_m(k). \tag{9}$$

Then $P_{m+1}(n)$ will tend to be a more well-behaved sequence than $P_m(n)$. Let us try to confirm this with quantitative calculations; our experience with the special case $m = 0$ indicates that it might be worthwhile to consider the subsequence $P_{m+1}(10^n s)$. The following results can, in fact, be derived:

Lemma Q. *For any integer $m \ge 1$ and any real number $\epsilon > 0$, there are functions $Q_m(s)$, $R_m(s)$ and an integer $N_m(\epsilon)$, such that whenever $n > N_m(\epsilon)$ and $1 \le s \le 10$, we have*

$$\begin{aligned} |P_m(10^n s) - Q_m(s)| &< \epsilon, \qquad \text{if } s \le r; \\ |P_m(10^n s) - \big(Q_m(s) + R_m(s)\big)| &< \epsilon, \qquad \text{if } s > r. \end{aligned} \tag{10}$$

Furthermore the functions $Q_m(s)$ and $R_m(s)$ satisfy the relations

$$\begin{aligned} Q_m(s) &= \frac{1}{s}\Big(\frac{1}{9}\int_1^{10} Q_{m-1}(t)\,dt + \int_1^s Q_{m-1}(t)\,dt + \frac{1}{9}\int_r^{10} R_{m-1}(t)\,dt\Big); \\ R_m(s) &= \frac{1}{s}\int_r^s R_{m-1}(t)\,dt; \\ Q_0(s) &= 1, \qquad R_0(s) = -1. \end{aligned} \tag{11}$$

Proof. Consider the functions $Q_m(s)$ and $R_m(s)$ defined by (11), and let

$$S_m(t) = \begin{cases} Q_m(t), & t \le r. \\ Q_m(t) + R_m(t), & t > r. \end{cases} \tag{12}$$

We will prove the lemma by induction on m.

First note that $Q_1(s) = \big(1 + (s-1) + (r-10)/9\big)/s = 1 + (r-10)/9s$, and $R_1(s) = (r-s)/s$. From (8) we find that $|P_1(10^n s) - S_1(s)| = O(n)/10^n$; this establishes the lemma when $m = 1$.

Now for $m > 1$, we have

$$P_m(10^n s) = \frac{1}{s}\Bigg(\sum_{0\le j<n} \frac{1}{10^{n-j}} \sum_{10^j \le k < 10^{j+1}} \frac{1}{10^j} P_{m-1}(k) + \sum_{10^n \le k \le 10^n s} \frac{1}{10^n} P_{m-1}(k)\Bigg),$$

and we want to approximate this quantity. By induction, the difference

$$\left| \sum_{10^j \le k \le 10^j q} \frac{1}{10^j} P_{m-1}(k) - \sum_{10^j \le k \le 10^j q} \frac{1}{10^j} S_{m-1}\Big(\frac{k}{10^j}\Big) \right| \tag{13}$$

is less than $q\epsilon$ when $1 \le q \le 10$ and $j > N_{m-1}(\epsilon)$. Since $S_{m-1}(t)$ is continuous, it is a Riemann-integrable function; and the difference

$$\left| \sum_{10^j \le k \le 10^j q} \frac{1}{10^j} S_{m-1}\Big(\frac{k}{10^j}\Big) - \int_1^q S_{m-1}(t)\,dt \right| \tag{14}$$

is less than ϵ for all j greater than some number N, independent of q, by the definition of integration. We may choose N to be $> N_{m-1}(\epsilon)$. Therefore for $n > N$, the difference

$$\left| P_m(10^n s) - \frac{1}{s}\Bigg(\sum_{0\le j<n} \frac{1}{10^{n-j}} \int_1^{10} S_{m-1}(t)\,dt + \int_1^s S_{m-1}(t)\,dt\Bigg) \right| \tag{15}$$

is bounded by $\sum_{0\le j\le N}(M/10^{n-j}) + \sum_{N<j<n}(11\epsilon/10^{n-j}) + 11\epsilon$, if M is an upper bound for (13) + (14) that is valid for all positive integers j. Finally, the sum $\sum_{0\le j<n}(1/10^{n-j})$, which appears in (15), is equal to $(1 - 1/10^n)/9$; so

$$\left| P_m(10^n s) - \frac{1}{s}\Big(\frac{1}{9}\int_1^{10} S_{m-1}(t)\,dt + \int_1^s S_{m-1}(t)\,dt\Big) \right|$$

can be made smaller than, say, 20ϵ, if n is taken large enough. Comparing this with (10) and (11) completes the proof. ∎

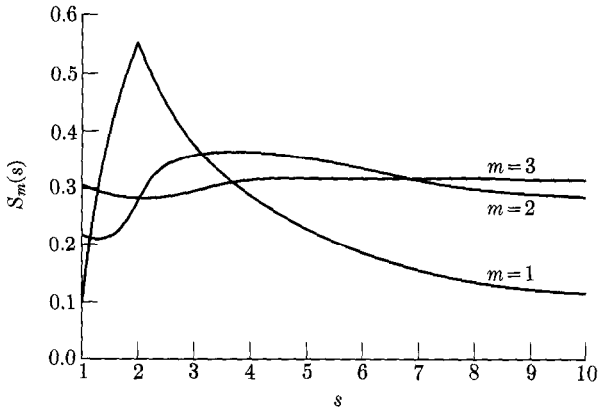


Fig. 5. The probability that the leading digit is 1.

The gist of Lemma Q is that we have the limiting relationship

$$\lim_{n\to\infty} P_m(10^n s) = S_m(s). \tag{16}$$

Also, since $S_m(s)$ is not constant as s varies, the limit

$$\lim_{n\to\infty} P_m(n)$$

(which would be our desired "probability") does not exist for any m. The situation is shown in Fig. 5, which shows the values of $S_m(s)$ when m is small and $r = 2$.

Even though $S_m(s)$ is not a constant, so that we do not have a definite limit for $P_m(n)$, note that already for $m = 3$ in Fig. 5 the value of $S_m(s)$ stays very close to $\log_{10} 2 = 0.30103\ldots$. Therefore we have good reason to suspect that $S_m(s)$ is very close to $\log_{10} r$ for all large m, and, in fact, that the sequence of functions $\langle S_m(s)\rangle$ converges uniformly to the constant function $\log_{10} r$.

It is interesting to prove this conjecture by explicitly calculating $Q_m(s)$ and $R_m(s)$ for all m, as in the proof of the following theorem:

Theorem F. *Let $S_m(s)$ be the limit defined in* (16). *For all $\epsilon > 0$, there exists a number $N(\epsilon)$ such that*

$$|S_m(s) - \log_{10} r| < \epsilon, \qquad \text{for } 1 \le s \le 10, \tag{17}$$

whenever $m > N(\epsilon)$.

Proof. In view of Lemma Q, we can prove this result if we can show that there is a number M depending on ϵ such that, for $1 \le s \le 10$ and for all $m > M$, we have

$$|Q_m(s) - \log_{10} r| < \epsilon \qquad \text{and} \qquad |R_m(s)| < \epsilon. \tag{18}$$

It is not difficult to solve the recurrence formula (11) for R_m: We have $R_0(s) = -1$, $R_1(s) = -1 + r/s$, $R_2(s) = -1 + (r/s)(1 + \ln(s/r))$, and in general

$$R_m(s) = -1 + \frac{r}{s}\left(1 + \frac{1}{1!}\ln\left(\frac{s}{r}\right) + \cdots + \frac{1}{(m-1)!}\left(\ln\left(\frac{s}{r}\right)\right)^{m-1}\right). \tag{19}$$

For the stated range of s, this converges uniformly to

$$-1 + (r/s)\exp(\ln(s/r)) = 0.$$

The recurrence (11) for Q_m takes the form

$$Q_m(s) = \frac{1}{s}\left(c_m + 1 + \int_1^s Q_{m-1}(t)\,dt\right), \tag{20}$$

where

$$c_m = \frac{1}{9}\left(\int_1^{10} Q_{m-1}(t)\,dt + \int_r^{10} R_{m-1}(t)\,dt\right) - 1. \tag{21}$$

The solution to recurrence (20) is easily found by trying out the first few cases and guessing at a formula that can be proved by induction; we find that

$$Q_m(s) = 1 + \frac{1}{s}\left(c_m + \frac{1}{1!}c_{m-1}\ln s + \cdots + \frac{1}{(m-1)!}c_1(\ln s)^{m-1}\right). \tag{22}$$

It remains for us to calculate the coefficients c_m, which by (19), (21), and (22) satisfy the relations

$$\begin{aligned} c_1 &= (r-10)/9; \\ c_{m+1} &= \frac{1}{9}\left(c_m\ln 10 + \frac{1}{2!}c_{m-1}(\ln 10)^2 + \cdots + \frac{1}{m!}c_1(\ln 10)^m \right. \\ &\qquad \left. + r\left(1 + \frac{1}{1!}\ln\frac{10}{r} + \cdots + \frac{1}{m!}\left(\ln\frac{10}{r}\right)^m\right) - 10\right). \end{aligned} \tag{23}$$

This sequence appears at first to be very complicated, but actually we can analyze it without difficulty with the help of generating functions. Let

$$C(z) = c_1 z + c_2 z^2 + c_3 z^3 + \cdots;$$

then since $10^z = 1 + z\ln 10 + (1/2!)(z\ln 10)^2 + \cdots$, we deduce that

$$\begin{aligned} c_{m+1} &= \frac{1}{10}c_{m+1} + \frac{9}{10}c_{m+1} \\ &= \frac{1}{10}\left(c_{m+1} + c_m\ln 10 + \cdots + \frac{1}{m!}c_1(\ln 10)^m\right) \\ &\qquad + \frac{r}{10}\left(1 + \cdots + \frac{1}{m!}\left(\ln\frac{10}{r}\right)^m\right) - 1 \end{aligned}$$

is the coefficient of z^{m+1} in the function

$$\frac{1}{10}C(z)10^z + \frac{r}{10}\left(\frac{10}{r}\right)^z\left(\frac{z}{1-z}\right) - \frac{z}{1-z}. \tag{24}$$

This condition holds for all values of m, so (24) must equal $C(z)$, and we obtain the explicit formula

$$C(z) = \frac{-z}{1-z}\left(\frac{(10/r)^{z-1}-1}{10^{z-1}-1}\right). \tag{25}$$

We want to study asymptotic properties of the coefficients of $C(z)$, to complete our analysis. The large parenthesized factor in (25) approaches $\ln(10/r)/\ln 10 = 1 - \log_{10} r$ as $z \to 1$, so we see that

$$C(z) + \frac{1-\log_{10} r}{1-z} = R(z) \tag{26}$$

is an analytic function of the complex variable z in the circle

$$|z| < \left|1 + \frac{2\pi i}{\ln 10}\right|.$$

In particular, $R(z)$ converges for $z = 1$, so its coefficients approach zero. This proves that the coefficients of $C(z)$ behave like those of $(\log_{10} r - 1)/(1-z)$, that is,

$$\lim_{m\to\infty} c_m = \log_{10} r - 1.$$

Finally, we may combine this with (22), to show that $Q_m(s)$ approaches

$$1 + \frac{\log_{10} r - 1}{s}\left(1 + \ln s + \frac{1}{2!}(\ln s)^2 + \cdots\right) = \log_{10} r$$

uniformly for $1 \le s \le 10$. ∎

Therefore we have established the logarithmic law for integers by direct calculation, at the same time seeing that it is an extremely good approximation to the average behavior although it is never precisely achieved.

The above proofs of Lemma Q and Theorem F are slight simplifications and amplifications of methods due to B. J. Flehinger, *AMM* **73** (1966), 1056–1061. Many authors have written about the distribution of initial digits, showing that the logarithmic law is a good approximation for many underlying distributions; see the survey by Ralph A. Raimi, *AMM* **83** (1976), 521–538, for a comprehensive review of the literature. Another interesting (and different) treatment of floating point distribution has been given by Alan G. Konheim, *Math. Comp.* **19** (1965), 143–144.

Exercise 17 discusses an approach to the definition of probability under which the logarithmic law holds exactly, over the integers. Furthermore, exercise 18 demonstrates that *any* reasonable definition of probability over the integers must lead to the logarithmic law, if it assigns a value to the probability of leading digits.

EXERCISES

1. [*13*] Given that u and v are nonzero floating decimal numbers *with the same sign*, what is the approximate probability that fraction overflow occurs during the calculation of $u \oplus v$, according to Tables 1 and 2?

2. [*42*] Make further tests of floating point addition and subtraction, to confirm or improve on the accuracy of Tables 1 and 2.

3. [*15*] What is the probability that the two leading digits of a floating decimal number are "23", according to the logarithmic law?

4. [*M18*] The text points out that the front pages of a well-used table of logarithms get dirtier than the back pages do. What if we had an *antilogarithm* table instead, i.e., a table giving the value of x when $\log_{10} x$ is given; which pages of such a table would be the dirtiest?

▶ **5.** [*M20*] Let U be a random real number that is uniformly distributed in the interval $0 < U < 1$. What is the distribution of the leading digits of U?

6. [*23*] If we have binary computer words containing $n + 1$ bits, we might use p bits for the fraction part of floating binary numbers, one bit for the sign, and $n - p$ bits for the exponent. This means that the range of values representable, i.e., the ratio of the largest positive normalized value to the smallest, is essentially $2^{2^{n-p}}$. The same computer word could be used to represent floating *hexadecimal* numbers, i.e., floating point numbers with radix 16, with $p + 2$ bits for the fraction part ($(p + 2)/4$ hexadecimal digits) and $n - p - 2$ bits for the exponent; then the range of values would be $16^{2^{n-p-2}} = 2^{2^{n-p}}$, the same as before, and with more bits in the fraction part. This may sound as if we are getting something for nothing, but the normalization condition for base 16 is weaker in that there may be up to three leading zero bits in the fraction part; thus not all of the $p + 2$ bits are "significant."

On the basis of the logarithmic law, what are the probabilities that the fraction part of a positive normalized radix 16 floating point number has exactly 0, 1, 2, and 3 leading zero bits? Discuss the desirability of hexadecimal versus binary.

7. [*HM28*] Prove that there is no distribution function $F(u)$ that satisfies (5) for each integer $b \geq 2$, and for all real values r in the range $1 \leq r \leq b$.

8. [*HM23*] Does (10) hold when $m = 0$ for suitable $N_0(\epsilon)$?

9. [*HM24*] (P. Diaconis.) Let $P_1(n)$, $P_2(n)$, ... be any sequence of functions defined by repeatedly averaging a given function $P_0(n)$ according to Eq. (9). Prove that $\lim_{m\to\infty} P_m(n) = P_0(1)$ for all fixed n.

▶ **10.** [*HM28*] The text shows that $c_m = \log_{10} r - 1 + \epsilon_m$, where ϵ_m approaches zero as $m \to \infty$. Obtain the next term in the asymptotic expansion of c_m.

11. [*M15*] Given that U is a random variable distributed according to the logarithmic law, prove that $1/U$ is also.

12. [*HM25*] (R. W. Hamming.) The purpose of this exercise is to show that the result of floating point multiplication tends to obey the logarithmic law more perfectly than the operands do. Let U and V be random, normalized, positive floating point numbers, whose fraction parts are independently distributed with the respective density functions $f(x)$ and $g(x)$. Thus, $f_u \leq r$ and $f_v \leq s$ with probability $\int_{1/b}^{r}\int_{1/b}^{s} f(x)g(y)\,dy\,dx$, for $1/b \leq r, s \leq 1$. Let $h(x)$ be the density function of the fraction part of $U \times V$ (unrounded). Define the *abnormality* $A(f)$ of a density function f to be the maximum

relative error,

$$A(f) = \max_{1/b \le x \le 1} \left| \frac{f(x) - l(x)}{l(x)} \right|,$$

where $l(x) = 1/(x \ln b)$ is the density of the logarithmic distribution.

Prove that $A(h) \le \min(A(f), A(g))$. (In particular, if either factor has logarithmic distribution the product does also.)

▶ **13.** [*M20*] The floating point multiplication routine, Algorithm 4.2.1M, requires zero or one left shifts during normalization, depending on whether $f_u f_v \ge 1/b$ or not. Assuming that the input operands are independently distributed according to the logarithmic law, what is the probability that no left shift is needed for normalization of the result?

▶ **14.** [*HM30*] Let U and V be random, normalized, positive floating point numbers whose fraction parts are independently distributed according to the logarithmic law, and let p_k be the probability that the difference in their exponents is k. Assuming that the distribution of the exponents is independent of the fraction parts, give an equation for the probability that "fraction overflow" occurs during the floating point addition of $U \oplus V$, in terms of the base b and the quantities $p_0, p_1, p_2, \ldots$. Compare this result with exercise 1. (Ignore rounding.)

15. [*HM28*] Let U, V, p_0, p_1, ... be as in exercise 14, and assume that radix 10 arithmetic is being used. Show that regardless of the values of $p_0, p_1, p_2, \ldots$, the sum $U \oplus V$ will *not* obey the logarithmic law exactly, and in fact the probability that $U \oplus V$ has leading digit 1 is always strictly *less* than $\log_{10} 2$.

16. [*HM28*] (P. Diaconis.) Let $P_0(n)$ be 0 or 1 for each n, and define "probabilities" $P_{m+1}(n)$ by repeated averaging, as in (9). Show that if $\lim_{n\to\infty} P_1(n)$ does not exist, neither does $\lim_{n\to\infty} P_m(n)$ for any m. [*Hint:* Prove that $a_n \to 0$ whenever we have $(a_1 + \cdots + a_n)/n \to 0$ and $a_{n+1} \le a_n + M/n$, for some fixed constant $M > 0$.]

▶ **17.** [*HM25*] (R. L. Duncan.) Another way to define the value of $\Pr(S(n))$ is to evaluate the quantity $\lim_{n\to\infty}((\sum_{S(k) \text{ and } 1\le k\le n} 1/k)/H_n)$; it can be shown that this "harmonic probability" exists and is equal to $\Pr(S(n))$, whenever the latter exists according to Definition 3.5A. Prove that the harmonic probability of the statement "$(\log_{10} n) \bmod 1 < r$" exists and equals r. (Thus, initial digits of integers *exactly* satisfy the logarithmic law in this sense.)

▶ **18.** [*HM30*] Let $P(S)$ be any real-valued function defined on sets S of positive integers, but not necessarily on all such sets, satisfying the following rather weak axioms:

i) If $P(S)$ and $P(T)$ are defined and $S \cap T = \emptyset$, then $P(S \cup T) = P(S) + P(T)$.
ii) If $P(S)$ is defined, then $P(S + 1) = P(S)$, where $S + 1 = \{ n + 1 \mid n \in S \}$.
iii) If $P(S)$ is defined, then $P(2S) = \frac{1}{2}P(S)$, where $2S = \{ 2n \mid n \in S \}$.
iv) If S is the set of *all* positive integers, then $P(S) = 1$.
v) If $P(S)$ is defined, then $P(S) \ge 0$.

Assume furthermore that $P(L_a)$ is defined for all positive integers a, where L_a is the set of all integers whose decimal representation begins with a:

$$L_a = \{ n \mid 10^m a \le n < 10^m (a + 1) \text{ for some integer } m \}.$$

(In this definition, m may be negative; for example, 1 is an element of L_{10}, but not of L_{11}.) Prove that $P(L_a) = \log_{10}(1 + 1/a)$ for all integers $a \ge 1$.

4.3. MULTIPLE-PRECISION ARITHMETIC

LET US NOW consider operations on numbers that have arbitrarily high precision. For simplicity in exposition, we shall assume that we are working with integers, instead of with numbers that have an embedded radix point.

4.3.1. The Classical Algorithms

In this section we shall discuss algorithms for

a) addition or subtraction of n-place integers, giving an n-place answer and a carry;

b) multiplication of an n-place integer by an m-place integer, giving an $(m+n)$-place answer;

c) division of an $(m+n)$-place integer by an n-place integer, giving an $(m+1)$-place quotient and an n-place remainder.

These may be called "the classical algorithms," since the word "algorithm" was used only in connection with these processes for several centuries. The term "n-place integer" means any integer less than b^n, where b is the radix of ordinary positional notation in which the numbers are expressed; such numbers can be written using at most n "places" in this notation.

It is a straightforward matter to apply the classical algorithms for integers to numbers with embedded radix points or to extended-precision floating point numbers, in the same way that arithmetic operations defined for integers in MIX are applied to these more general problems.

In this section we shall study algorithms that do operations (a), (b), and (c) above for integers expressed in radix b notation, where b is any given integer ≥ 2. Thus the algorithms are quite general definitions of arithmetic processes, and as such they are unrelated to any particular computer. But the discussion in this section will also be somewhat machine-oriented, since we are chiefly concerned with efficient methods for doing high-precision calculations by computer. Although our examples are based on the mythical MIX, essentially the same considerations apply to nearly every other machine. For convenience, we shall assume first that we have a computer (like MIX) that uses the signed-magnitude representation for numbers; suitable modifications for complement notations are discussed near the end of this section.

The most important fact to understand about extended-precision numbers is that they may be regarded as numbers written in radix w notation, where w is the computer's word size. For example, an integer that fills 10 words on a computer whose word size is $w = 10^{10}$ has 100 decimal digits; but we will consider it to be a 10-place number to the base 10^{10}. This viewpoint is justified for the same reason that we may convert, say, from binary to octal notation, simply by grouping the bits together. (See Eq. 4.1–5.)

In these terms, we are given the following primitive operations to work with:

a_0) addition or subtraction of one-place integers, giving a one-place answer and a carry;

b_0) multiplication of a one-place integer by another one-place integer, giving a two-place answer;

c_0) division of a two-place integer by a one-place integer, provided that the quotient is a one-place integer, and yielding also a one-place remainder.

By adjusting the word size, if necessary, nearly all computers will have these three operations available; so we will construct algorithms (a), (b), and (c) mentioned above in terms of the primitive operations (a_0), (b_0), and (c_0).

Since we are visualizing extended-precision integers as base b numbers, it is sometimes helpful to think of the situation when $b = 10$, and to imagine that we are doing the arithmetic by hand. Then operation (a_0) is analogous to memorizing the addition table; (b_0) is analogous to memorizing the multiplication table; and (c_0) is essentially memorizing the multiplication table in reverse. The more complicated operations (a), (b), (c) on high-precision numbers can now be done using the simple addition, subtraction, multiplication, and long-division procedures we are taught in elementary school. In fact, most of the algorithms we shall discuss in this section are essentially nothing more than mechanizations of familiar pencil-and-paper operations. Of course, we must state the algorithms much more precisely than they have ever been stated in the fifth grade, and we should also attempt to minimize computer memory and running time requirements.

To avoid a tedious discussion and cumbersome notations, let us assume that all numbers we deal with are *nonnegative*. The additional work of computing the signs, etc., is quite straightforward, and the reader will find it easy to fill in any details of this sort.

First comes addition, which of course is very simple, but it is worth studying since the same ideas occur in the other algorithms also:

Algorithm A (*Addition of nonnegative integers*). Given nonnegative n-place integers $(u_1u_2\ldots u_n)_b$ and $(v_1v_2\ldots v_n)_b$, this algorithm forms their radix-b sum, $(w_0w_1w_2\ldots w_n)_b$. (Here w_0 is the "carry," and it will always be equal to 0 or 1.)

A1. [Initialize.] Set $j \leftarrow n$, $k \leftarrow 0$. (The variable j will run through the various digit positions, and the variable k keeps track of carries at each step.)

A2. [Add digits.] Set $w_j \leftarrow (u_j + v_j + k) \bmod b$, and $k \leftarrow \lfloor (u_j + v_j + k)/b \rfloor$. (In other words, k is set to 1 or 0, depending on whether a "carry" occurs or not, i.e., whether $u_j + v_j + k \geq b$ or not. At most one carry is possible during the two additions, since we always have

$$u_j + v_j + k \leq (b-1) + (b-1) + 1 < 2b,$$

by induction on the computation.)

A3. [Loop on j.] Decrease j by one. Now if $j > 0$, go back to step A2; otherwise set $w_0 \leftarrow k$ and terminate the algorithm. ∎

For a formal proof that Algorithm A is a valid, see exercise 4.

A MIX program for this addition process might take the following form:

Program A (*Addition of nonnegative integers*). Let $\texttt{LOC}(u_j) \equiv \texttt{U}+j$, $\texttt{LOC}(v_j) \equiv \texttt{V}+j$, $\texttt{LOC}(w_j) \equiv \texttt{W}+j$, rI1 $\equiv j$, rA $\equiv k$, word size $\equiv b$, N $\equiv n$.

```
01        ENT1  N          1          A1. Initialize.  j ← n.
02        JOV   OFLO       1          Ensure overflow is off.
03   1H   ENTA  0        N+1-K        k ← 0.
04        J1Z   3F       N+1-K        To A3 if j = 0.
05   2H   ADD   U,1          N        A2. Add digits.
06        ADD   V,1          N
07        STA   W,1          N
08        DEC1  1            N        A3. Loop on j.
09        JNOV  1B           N        If no overflow, set k ← 0.
10        ENTA  1            K        Otherwise, set k ← 1.
11        J1P   2B           K        To A2 if j ≠ 0.
12   3H   STA   W            1        Store final carry in w0.  ∎
```

The running time for this program is $10N+6$ cycles, independent of the number of carries, K. The quantity K is analyzed in detail at the close of this section.

Many modifications of Algorithm A are possible, and only a few of these are mentioned in the exercises below. A chapter on generalizations of this algorithm might be entitled "How to design addition circuits for a digital computer."

The problem of subtraction is similar to addition, but the differences are worth noting:

Algorithm S (*Subtraction of nonnegative integers*). Given nonnegative n-place integers $(u_1u_2\ldots u_n)_b \geq (v_1v_2\ldots v_n)_b$, this algorithm forms their nonnegative radix-b difference, $(w_1w_2\ldots w_n)_b$.

S1. [Initialize.] Set $j \leftarrow n$, $k \leftarrow 0$.

S2. [Subtract digits.] Set $w_j \leftarrow (u_j - v_j + k) \bmod b$, and $k \leftarrow \lfloor (u_j - v_j + k)/b \rfloor$. (In other words, k is set to -1 or 0, depending on whether a "borrow" occurs or not, i.e., whether $u_j - v_j + k < 0$ or not. In the calculation of w_j, note that we must have $-b = 0 - (b-1) + (-1) \leq u_j - v_j + k \leq (b-1) - 0 + 0 < b$; hence $0 \leq u_j - v_j + k + b < 2b$, and this suggests the method of computer implementation explained below.)

S3. [Loop on j.] Decrease j by one. Now if $j > 0$, go back to step S2; otherwise terminate the algorithm. (When the algorithm terminates, we should have $k = 0$; the condition $k = -1$ will occur if and only if $v_1\ldots v_n > u_1\ldots u_n$, and this is contrary to the given assumptions. See exercise 12.) ∎

In a MIX program to implement subtraction, it is most convenient to retain the value $1+k$ instead of k throughout the algorithm, so that we can calculate $u_j - v_j + (1+k) + (b-1)$ in step S2. (Recall that b is the word size.) This is illustrated in the following code.

Program S (*Subtraction of nonnegative integers*). This program is analogous to the code in Program A; we have rI1 $\equiv j$, rA $\equiv 1 + k$. Here, as in other programs of this section, location WM1 contains the constant $b - 1$, the largest possible value that can be stored in a MIX word; cf. Program 4.2.3D, lines 38–39.

```
01       ENT1 N       1      S1. Initialize.  j ← n.
02       JOV  OFLO    1      Ensure overflow is off.
03   1H  J1Z  DONE    K+1    Terminate if j = 0.
04       ENTA 1       K      Set k ← 0.
05   2H  ADD  U,1     N      S2. Subtract digits.
06       SUB  V,1     N      Compute u_j − v_j + k + b.
07       ADD  WM1     N
08       STA  W,1     N      (May be minus zero)
09       DEC1 1       N      S3. Loop on j.
10       JOV  1B      N      If overflow, set k ← 0.
11       ENTA 0       N−K    Otherwise, set k ← −1.
12       J1P  2B      N−K    Back to S2.
13       HLT  5              (Error, v > u)  ∎
```

The running time for this program is $12N + 3$ cycles, slightly longer than the corresponding amount for Program A.

The reader may wonder if it would not be worthwhile to have a combined addition-subtraction routine in place of the two algorithms A and S. But an examination of the computer programs shows that it is generally better to use two different routines, so that the inner loops of the computations can be performed as rapidly as possible, since the programs are so short.

Our next problem is multiplication, and here we carry the ideas used in Algorithm A a little further:

Algorithm M (*Multiplication of nonnegative integers*). Given nonnegative integers $(u_1 u_2 \ldots u_n)_b$ and $(v_1 v_2 \ldots v_m)_b$, this algorithm forms their radix-b product $(w_1 w_2 \ldots w_{m+n})_b$. (The conventional pencil-and-paper method is based on forming the partial products $(u_1 u_2 \ldots u_n) \times v_j$ first, for $1 \le j \le m$, and then adding these products together with appropriate scale factors; but in a computer it is best to do the addition concurrently with the multiplication, as described in this algorithm.)

M1. [Initialize.] Set $w_{m+1}, w_{m+2}, \ldots, w_{m+n}$ all to zero. Set $j \leftarrow m$. (If $w_{m+1}, \ldots, w_{m+n}$ were not cleared to zero in this step, it turns out that the steps below would set

$$(w_1 \ldots w_{m+n})_b \leftarrow (u_1 \ldots u_n)_b \times (v_1 \ldots v_m)_b + (w_{m+1} \ldots w_{m+n})_b.$$

This more general operation is sometimes useful.)

M2. [Zero multiplier?] If $v_j = 0$, set $w_j \leftarrow 0$ and go to step M6. (This test saves a good deal of time if there is a reasonable chance that v_j is zero, but otherwise it may be omitted without affecting the validity of the algorithm.)

Table 1

MULTIPLICATION OF 914 BY 84.

Step	i	j	u_i	v_j	t	w_1	w_2	w_3	w_4	w_5
M5	3	2	4	4	16	x	x	0	0	6
M5	2	2	1	4	05	x	x	0	5	6
M5	1	2	9	4	36	x	x	6	5	6
M6	0	2	x	4	36	x	3	6	5	6
M5	3	1	4	8	37	x	3	6	7	6
M5	2	1	1	8	17	x	3	7	7	6
M5	1	1	9	8	76	x	6	7	7	6
M6	0	1	x	8	76	7	6	7	7	6

M3. [Initialize i.] Set $i \leftarrow n$, $k \leftarrow 0$.

M4. [Multiply and add.] Set $t \leftarrow u_i \times v_j + w_{i+j} + k$; then set $w_{i+j} \leftarrow t \bmod b$ and $k \leftarrow \lfloor t/b \rfloor$. (Here the "carry" k will always be in the range $0 \leq k < b$; see below.)

M5. [Loop on i.] Decrease i by one. Now if $i > 0$, go back to step M4; otherwise set $w_j \leftarrow k$.

M6. [Loop on j.] Decrease j by one. Now if $j > 0$, go back to step M2; otherwise the algorithm terminates. ∎

Algorithm M is illustrated in Table 1, assuming that $b = 10$, by showing the states of the computation at the beginning of steps M5 and M6. A proof of Algorithm M appears in the answer to exercise 14.

The two inequalities

$$0 \leq t < b^2, \qquad 0 \leq k < b \tag{1}$$

are crucial for an efficient implementation of this algorithm, since they point out how large a register is needed for the computations. These inequalities may be proved by induction as the algorithm proceeds, for if we have $k < b$ at the start of step M4, we have

$$u_i \times v_j + w_{i+j} + k \leq (b-1) \times (b-1) + (b-1) + (b-1) = b^2 - 1 < b^2.$$

The following MIX program shows the considerations that are necessary when Algorithm M is implemented on a computer. The coding for step M4 would be a little simpler if our computer had a "multiply-and-add" instruction, or if it had a double-length accumulator for addition.

Program M. (*Multiplication of nonnegative integers*). This program is analogous to Program A. rI1 $\equiv i$, rI2 $\equiv j$, rI3 $\equiv i + j$, CONTENTS(CARRY) $\equiv k$.

```
01    ENT1  N          1     M1. Initialize.
02    JOV   OFLO       1     Ensure overflow is off.
03    STZ   W+M,1      N     w_{m+i} ← 0.
04    DEC1  1          N
05    J1P   *-2        N     Repeat for n ≥ i > 0.
```

```
06      ENT2 M          1       j ← m.
07  1H  LDX  V,2        M       M2. Zero multiplier?
08      JXZ  8F         M       If v_j = 0, set w_j ← 0 and go to M6.
09      ENT1 N          M − Z   M3. Initialize i.
10      ENT3 N,2        M − Z   i ← n, (i + j) ← (n + j).
11      ENTX 0          M − Z   k ← 0.
12  2H  STX  CARRY      (M − Z)N M4. Multiply and add.
13      LDA  U,1        (M − Z)N
14      MUL  V,2        (M − Z)N rAX ← u_i × v_j.
15      SLC  5          (M − Z)N Interchange rA ↔ rX.
16      ADD  W,3        (M − Z)N Add w_{i+j} to lower half.
17      JNOV *+2        (M − Z)N Did overflow occur?
18      INCX 1          K       If so, carry 1 into upper half.
19      ADD  CARRY      (M − Z)N Add k to lower half.
20      JNOV *+2        (M − Z)N Did overflow occur?
21      INCX 1          K'      If so, carry 1 into upper half.
22      STA  W,3        (M − Z)N w_{i+j} ← t mod b.
23      DEC1 1          (M − Z)N M5. Loop on i.
24      DEC3 1          (M − Z)N Decrease i and (i + j) by 1.
25      J1P  2B         (M − Z)N Back to M4 if i > 0; rX = ⌊t/b⌋.
26  8H  STX  W,2        M       Set w_j ← k.
27      DEC2 1          M       M6. Loop on j.
28      J2P  1B         M       Repeat until j = 0. ∎
```

The execution time of Program M depends on the number of places, M, in the multiplier v; the number of places, N, in the multiplicand u; the number of zeros, Z, in the multiplier; and the number of carries, K and K', that occur during the addition to the lower half of the product in the computation of t. If we approximate both K and K' by the reasonable (although somewhat pessimistic) values $\frac{1}{2}(M - Z)N$, we find that the total running time comes to $28MN + 10M + 4N + 3 - Z(28N + 3)$ cycles. If step M2 were deleted, the running time would be $28MN + 7M + 4N + 3$ cycles, so this step is not advantageous unless the density of zero positions within the multiplier is $Z/M > 3/(28N + 3)$. If the multiplier is chosen completely at random, this ratio Z/M is expected to be only about $1/b$, which is extremely small; so step M2 is usually *not* worthwhile, unless b is small.

Algorithm M is not the fastest way to multiply when m and n are large, although it has the advantage of simplicity. Speedier methods are discussed in Section 4.3.3; even when $m = n = 4$, it is possible to multiply numbers in a little less time than is required by Algorithm M.

The final algorithm of concern to us in this section is long division, in which we want to divide $(n + m)$-place integers by n-place integers. Here the ordinary pencil-and-paper method involves a certain amount of guesswork and ingenuity on the part of the person doing the division; we must either eliminate this guesswork from the algorithm or develop some theory to explain it more carefully.

A moment's reflection about the ordinary process of long division shows that the general problem breaks down into simpler steps, each of which is the division

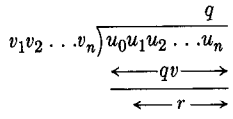


Fig. 6. Wanted: a way to determine q rapidly.

of an $(n+1)$-place number u by the n-place divisor v, where $0 \le u/v < b$; the remainder r after each step is less than v, so we may use the quantity $rb+$(next place of dividend) as the new u in the succeeding step. For example, if we are asked to divide 3142 by 47, we first divide 314 by 47, getting 6 and a remainder of 32; then we divide 322 by 47, getting 6 and a remainder of 40; thus we have a quotient of 66 and a remainder of 40. It is clear that this same idea works in general, and so our search for an appropriate division algorithm reduces to the following problem (Fig. 6):

Let $u = (u_0u_1\ldots u_n)_b$ and $v = (v_1v_2\ldots v_n)_b$ be nonnegative integers in radix-b notation, such that $u/v < b$. Find an algorithm to determine $q = \lfloor u/v \rfloor$.

We may observe that the condition $u/v < b$ is equivalent to the condition that $u/b < v$; i.e., $\lfloor u/b \rfloor < v$. This is simply the condition that $(u_0u_1\ldots u_{n-1})_b < (v_1v_2\ldots v_n)_b$. Furthermore, if we write $r = u - qv$, then q is the unique integer such that $0 \le r < v$.

The most obvious approach to this problem is to make a guess about q, based on the most significant digits of u and v. It isn't obvious that such a method will be reliable enough, but it is worth investigating; let us therefore set

$$\hat{q} = \min\left(\left\lfloor \frac{u_0b + u_1}{v_1} \right\rfloor, b-1\right). \tag{2}$$

This formula says $\hat{q}$ is obtained by dividing the two leading digits of u by the leading digit of v; and if the result is b or more we can replace it by $(b-1)$.

It is a remarkable fact, which we will now investigate, that this value $\hat{q}$ is always a very good approximation to the desired answer q, so long as v_1 is reasonably large. In order to analyze how close $\hat{q}$ comes to q, we will first prove that $\hat{q}$ is never too small.

Theorem A. *In the notation above, $\hat{q} \ge q$.*

Proof. Since $q \le b-1$, the theorem is certainly true if $\hat{q} = b-1$. Otherwise we have $\hat{q} = \lfloor (u_0b + u_1)/v_1 \rfloor$, hence $\hat{q}v_1 \ge u_0b + u_1 - v_1 + 1$. It follows that

$$\begin{aligned} u - \hat{q}v \le u - \hat{q}v_1b^{n-1} &\le u_0b^n + \cdots + u_n \\ &\quad - (u_0b^n + u_1b^{n-1} - v_1b^{n-1} + b^{n-1}) \\ &= u_2b^{n-2} + \cdots + u_n - b^{n-1} + v_1b^{n-1} < v_1b^{n-1} \le v. \end{aligned}$$

Since $u - \hat{q}v < v$, we must have $\hat{q} \ge q$. ∎

We will now prove that $\hat{q}$ cannot be much larger than q in practical situations. Assume that $\hat{q} \geq q + 3$. We have

$$\hat{q} \leq \frac{u_0 b + u_1}{v_1} = \frac{u_0 b^n + u_1 b^{n-1}}{v_1 b^{n-1}} \leq \frac{u}{v_1 b^{n-1}} < \frac{u}{v - b^{n-1}}.$$

(The case $v = b^{n-1}$ is impossible, for if $v = (100\ldots0)_b$ then $q = \hat{q}$.) Furthermore, the relation $q > (u/v) - 1$ implies that

$$3 \leq \hat{q} - q < \frac{u}{v - b^{n-1}} - \frac{u}{v} + 1 = \frac{u}{v}\left(\frac{b^{n-1}}{v - b^{n-1}}\right) + 1.$$

Therefore

$$\frac{u}{v} > 2\left(\frac{v - b^{n-1}}{b^{n-1}}\right) \geq 2(v_1 - 1).$$

Finally, since $b - 4 \geq \hat{q} - 3 \geq q = \lfloor u/v \rfloor \geq 2(v_1 - 1)$, we have $v_1 < \lfloor b/2 \rfloor$. This proves the result we seek:

Theorem B. *If $v_1 \geq \lfloor b/2 \rfloor$, then $\hat{q} - 2 \leq q \leq \hat{q}$.* ∎

The most important part of this theorem is that *the conclusion is independent of b*; no matter how large b is, the trial quotient $\hat{q}$ will never be more than 2 in error.

The condition that $v_1 \geq \lfloor b/2 \rfloor$ is very much like a normalization requirement; in fact, it is exactly the condition of floating-binary normalization in a binary computer. One simple way to ensure that v_1 is sufficiently large is to multiply both u and v by $\lfloor b/(v_1 + 1) \rfloor$; this does not change the value of u/v, nor does it increase the number of places in v, and exercise 23 proves that it will always make the new value of v_1 large enough. (*Note:* Another way to normalize the divisor is discussed in exercise 28.)

Now that we have armed ourselves with all of these facts, we are in a position to write the desired long-division algorithm. This algorithm uses a slightly improved choice of $\hat{q}$ in step D3, which guarantees that $q = \hat{q}$ or $\hat{q} - 1$; in fact, the improved choice of $\hat{q}$ made here is almost always accurate.

Algorithm D (*Division of nonnegative integers*). Given nonnegative integers $u = (u_1 u_2 \ldots u_{m+n})_b$ and $v = (v_1 v_2 \ldots v_n)_b$, where $v_1 \neq 0$ and $n > 1$, we form the radix-b quotient $\lfloor u/v \rfloor = (q_0 q_1 \ldots q_m)_b$ and the remainder $u \bmod v = (r_1 r_2 \ldots r_n)_b$. (This notation is slightly different from that used in the above proofs. When $n = 1$, the simpler algorithm of exercise 16 should be used.)

D1. [Normalize.] Set $d \leftarrow \lfloor b/(v_1 + 1) \rfloor$. Then set $(u_0 u_1 u_2 \ldots u_{m+n})_b$ equal to $(u_1 u_2 \ldots u_{m+n})_b$ times d, and set $(v_1 v_2 \ldots v_n)_b$ equal to $(v_1 v_2 \ldots v_n)_b$ times d. (Note the introduction of a new digit position u_0 at the left of u_1; if $d = 1$, all we need to do in this step is to set $u_0 \leftarrow 0$. On a binary computer it may be preferable to choose d to be a power of 2 instead of using the value suggested here; any value of d that results in $v_1 \geq \lfloor b/2 \rfloor$ will suffice. See also exercise 37.)

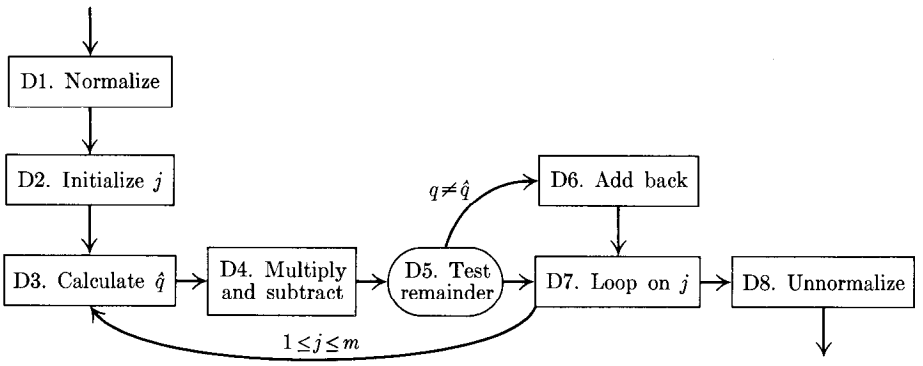


Fig. 7. Long division.

D2. [Initialize j.] Set $j \leftarrow 0$. (The loop on j, steps D2 through D7, will be essentially a division of $(u_j u_{j+1} \dots u_{j+n})_b$ by $(v_1 v_2 \dots v_n)_b$ to get a single quotient digit q_j; cf. Fig. 6.)

D3. [Calculate $\hat{q}$.] If $u_j = v_1$, set $\hat{q} \leftarrow b-1$; otherwise set $\hat{q} \leftarrow \lfloor (u_j b + u_{j+1})/v_1 \rfloor$. Now test if $v_2 \hat{q} > (u_j b + u_{j+1} - \hat{q} v_1) b + u_{j+2}$; if so, decrease $\hat{q}$ by 1 and repeat this test. (The latter test determines at high speed most of the cases in which the trial value $\hat{q}$ is one too large, and it eliminates *all* cases where $\hat{q}$ is two too large; see exercises 19, 20, 21.)

D4. [Multiply and subtract.] Replace $(u_j u_{j+1} \dots u_{j+n})_b$ by $(u_j u_{j+1} \dots u_{j+n})_b$ minus $\hat{q}$ times $(v_1 v_2 \dots v_n)_b$. This step (analogous to steps M3, M4, and M5 of Algorithm M) consists of a simple multiplication by a one-place number, combined with a subtraction. The digits $(u_j, u_{j+1}, \dots, u_{j+n})$ should be kept positive; if the result of this step is actually negative, $(u_j u_{j+1} \dots u_{j+n})_b$ should be left as the true value plus b^{n+1}, i.e., as the b's complement of the true value, and a "borrow" to the left should be remembered.

D5. [Test remainder.] Set $q_j \leftarrow \hat{q}$. If the result of step D4 was negative, go to step D6; otherwise go on to step D7.

D6. [Add back.] (The probability that this step is necessary is very small, on the order of only $2/b$, as shown in exercise 21; test data that activates this step should therefore be specifically contrived when debugging.) Decrease q_j by 1, and add $(0 v_1 v_2 \dots v_n)_b$ to $(u_j u_{j+1} u_{j+2} \dots u_{j+n})_b$. (A carry will occur to the left of u_j, and it should be ignored since it cancels with the "borrow" that occurred in D4.)

D7. [Loop on j.] Increase j by one. Now if $j \leq m$, go back to D3.

D8. [Unnormalize.] Now $(q_0 q_1 \dots q_m)_b$ is the desired quotient, and the desired remainder may be obtained by dividing $(u_{m+1} \dots u_{m+n})_b$ by d. ∎

The representation of Algorithm D as a MIX program has several points of interest:

Program D (*Division of nonnegative integers*). The conventions of this program are analogous to Program A; rI1 $\equiv i$, rI2 $\equiv j - m$, rI3 $\equiv i + j$.

```
001  D1 JOV   OFLO          1          D1. Normalize.
...                                    (See exercise 25)
039  D2 ENN2  M             1          D2. Initialize j.
040     STZ   V             1          Set v0 ← 0, for convenience in D4.
041  D3 LDA   U+M,2(1:5)    M+1        D3. Calculate q̂.
042     LDX   U+M+1,2       M+1        rAX ← uj b + uj+1.
043     DIV   V+1           M+1        rA ← ⌊rAX/v1⌋.
044     JOV   1F            M+1        Jump if quotient = b.
045     STA   QHAT          M+1        q̂ ← rA.
046     STX   RHAT          M+1        r̂ ← uj b + uj+1 − q̂v1
047     JMP   2F            M+1              = (uj b + uj+1) mod v1.
048  1H LDX   WM1                      rX ← b − 1.
049     LDA   U+M+1,2                  rA ← uj+1. (Here uj = v1.)
050     JMP   4F
051  3H LDX   QHAT          E
052     DECX  1             E          Decrease q̂ by one.
053     LDA   RHAT          E          Adjust r̂ accordingly:
054  4H STX   QHAT          E          q̂ ← rX.
055     ADD   V+1           E          rA ← rA + v1.
056     JOV   D4            E          (If r̂ will be ≥ b, v2q̂ will be < r̂b.)
057     STA   RHAT          E          r̂ ← rA.
058     LDA   QHAT          E
059  2H MUL   V+2           M+E+1
060     CMPA  RHAT          M+E+1      Test if v2q̂ ≤ r̂b + uj+2.
061     JL    D4            M+E+1
062     JG    3B            E
063     CMPX  U+M+2,2
064     JG    3B                       If not, q̂ is too large.
065  D4 ENTX  1             M+1        D4. Multiply and subtract.
066     ENT1  N             M+1        i ← n.
067     ENT3  M+N,2         M+1        (i + j) ← (n + j).
068  2H STX   CARRY         (M+1)(N+1) (Here 1 − b < rX ≤ +1.)
069     LDAN  V,1           (M+1)(N+1)
070     MUL   QHAT          (M+1)(N+1) rAX ← −q̂vi.
071     SLC   5             (M+1)(N+1) Interchange rA ↔ rX.
072     ADD   CARRY         (M+1)(N+1) Add the contribution from the
073     JNOV  *+2           (M+1)(N+1)       digit to the right, plus 1.
074     DECX  1             K          If sum is ≤ −b, carry −1.
075     ADD   U,3           (M+1)(N+1) Add ui+j.
076     ADD   WM1           (M+1)(N+1) Add b − 1 to force + sign.
077     JNOV  *+2           (M+1)(N+1) If no overflow, carry −1.
078     INCX  1             K′         rX ≡ carry + 1.
079     STA   U,3           (M+1)(N+1) ui+j ← rA (may be minus zero).
080     DEC1  1             (M+1)(N+1)
081     DEC3  1             (M+1)(N+1)
082     J1NN  2B            (M+1)(N+1) Repeat for n ≥ i ≥ 0.
```

```
083  D5 LDA  QHAT      M+1   D5. Test remainder.
084     STA  Q+M,2     M+1   Set q_j ← q̂.
085     JXP  D7        M+1   (Here rX = 0 or 1, since v_0 = 0.)
086  D6 DECA 1               D6. Add back.
087     STA  Q+M,2           Set q_j ← q̂ − 1.
088     ENT1 N               i ← n.
089     ENT3 M+N,2           (i + j) ← (n + j).
090  1H ENTA 0               (This is essentially Program A.)
091  2H ADD  U,3
092     ADD  V,1
093     STA  U,3
094     DEC1 1
095     DEC3 1
096     JNOV 1B
097     ENTA 1
098     J1NN 2B
099  D7 INC2 1       M+1   D7. Loop on j.
100     J2NP D3      M+1   Repeat for 0 ≤ j ≤ m.
101  D8 ···                (See exercise 26) ∎
```

Note how easily the rather complex-appearing calculations and decisions of step D3 can be handled inside the machine. Note also that the program for step D4 is analogous to Program M, except that the ideas of Program S have also been incorporated.

The running time for Program D can be estimated by considering the quantities M, N, E, K, and K' shown in the program. (These quantities ignore several situations that occur only with very low probability; for example, we may assume that lines 048–050, 063–064, and step D6 are never executed.) Here $M+1$ is the number of words in the quotient; N is the number of words in the divisor; E is the number of times $\hat{q}$ is adjusted downwards in step D3; K and K' are the number of times certain "carry" adjustments are made during the multiply-subtract loop. If we assume that $K+K'$ is approximately $(N+1)(M+1)$, and that E is approximately $\frac{1}{2}M$, we get a total running time of approximately

$$30MN + 30N + 89M + 111$$

cycles, plus $67N+235M+4$ more if $d > 1$. (The program segments of exercises 25 and 26 are included in these totals.) When M and N are large, this is only about seven percent longer than the time Program M takes to multiply the quotient by the divisor.

Further commentary on Algorithm D appears in the exercises at the close of this section.

It is possible to debug programs for multiple-precision arithmetic by using the multiplication and addition routines to check the result of the division routine, etc. The following type of test data is occasionally useful:

$$(t^m - 1)(t^n - 1) = t^{m+n} - t^n - t^m + 1.$$

If $m < n$, this number has the radix-t expansion

$$\underbrace{(t-1)\ \ldots\ (t-1)}_{m-1\text{ places}}\ (t-2)\ \underbrace{(t-1)\ \ldots\ (t-1)}_{n-m\text{ places}}\ \underbrace{0\ \ldots\ 0}_{m-1\text{ places}}\ 1;$$

for example, $(10^3 - 1)(10^5 - 1) = 99899001$. In the case of Program D, it is also necessary to find some test cases that cause the rarely executed parts of the program to be exercised; some portions of that program would probably never get tested even if a million random test cases were tried.

Now that we have seen how to operate with signed-magnitude numbers, let us consider what approach should be taken to the same problems when a computer with complement notation is being used. For two's complement and ones' complement notations, it is usually best to let the radix b be *one half* of the word size; thus for a 32-bit computer word we would use $b = 2^{31}$ in the above algorithms. The sign bit of all but the most significant word of a multiple-precision number will be zero, so that no anomalous sign correction takes place during the computer's multiplication and division operations. In fact, the basic meaning of complement notation requires that we consider all but the most significant word to be nonnegative. For example, assuming an 8-bit word, the two's complement number

$$11011111\quad 1111110\quad 1101011$$

(where the sign bit is shown only in the most significant word) is properly thought of as

$$-2^{21} + (1011111)_2 \cdot 2^{14} + (1111110)_2 \cdot 2^7 + (1101011)_2.$$

Addition of signed numbers is slightly easier when complement notations are being used, since the routine for adding n-place nonnegative integers can be used for arbitrary n-place integers; the sign appears only in the first word, so the less significant words may be added together irrespective of the actual sign. (Special attention must be given to the leftmost carry when ones' complement notation is being used, however; it must be added into the least significant word, and possibly propagated further to the left.) Similarly, we find that subtraction of signed numbers is slightly simpler with complement notation. On the other hand, multiplication and division seem to be done most easily by working with nonnegative quantities and doing suitable complementation operations beforehand to make sure that both operands are nonnegative; it may be possible to avoid this complementation by devising some tricks for working directly with negative numbers in a complement notation, and it is not hard to see how this could be done in double-precision multiplication, but care should be taken not to slow down the inner loops of the subroutines when high precision is required. Note that the product of two m-place numbers in two's complement notation may require $2m + 1$ places: the square of $(-b^m)$ is b^{2m}.

Let us now turn to an analysis of the quantity K that arises in Program A, i.e., the number of carries that occur when two n-place numbers are being added together. Although K has no effect on the total running time of Program A, it does affect the running time of the Program A's counterparts that deal with complement notations, and its analysis is interesting in itself as a significant application of generating functions.

Suppose that u and v are independent random n-place integers, uniformly distributed in the range $0 \le u, v < b^n$. Let p_{nk} be the probability that exactly k carries occur in the addition of u to v, *and* that one of these carries occurs in the most significant position (so that $u + v \ge b^n$). Similarly, let q_{nk} be the probability that exactly k carries occur, but that there is no carry in the most significant position. Then it is not hard to see that

$$\begin{aligned} p_{0k} &= 0, \qquad q_{0k} = \delta_{0k}, \qquad \text{for all } k; \\ p_{(n+1)(k+1)} &= \frac{b+1}{2b} p_{nk} + \frac{b-1}{2b} q_{nk}, \\ q_{(n+1)k} &= \frac{b-1}{2b} p_{nk} + \frac{b+1}{2b} q_{nk}; \end{aligned} \tag{3}$$

this happens because $(b-1)/2b$ is the probability that $u_1 + v_1 \ge b$ and $(b+1)/2b$ is the probability that $u_1 + v_1 + 1 \ge b$, when u_1 and v_1 are independently and uniformly distributed integers in the range $0 \le u_1, v_1 < b$.

To obtain further information about these quantities p_{nk} and q_{nk}, we may set up the generating functions

$$P(z,t) = \sum_{k,n} p_{nk}\, z^k t^n, \qquad Q(z,t) = \sum_{k,n} q_{nk}\, z^k t^n. \tag{4}$$

From (3) we have the basic relations

$$\begin{aligned} P(z,t) &= zt\left(\frac{b+1}{2b} P(z,t) + \frac{b-1}{2b} Q(z,t)\right), \\ Q(z,t) &= 1 + t\left(\frac{b-1}{2b} P(z,t) + \frac{b+1}{2b} Q(z,t)\right). \end{aligned}$$

These two equations are readily solved for $P(z,t)$ and $Q(z,t)$; and if we let

$$G(z,t) = P(z,t) + Q(z,t) = \sum_n G_n(z) t^n,$$

where $G_n(z)$ is the generating function for the total number of carries when n-place numbers are added, we find that

$$G(z,t) = (b - zt)/p(z,t), \quad \text{where } p(z,t) = b - \tfrac{1}{2}(1+b)(1+z)t + zt^2. \tag{5}$$

Note that $G(1,t) = 1/(1-t)$, and this checks with the fact that $G_n(1)$ must equal 1 (it is the sum of all the possible probabilities). Taking partial derivatives

of (5) with respect to z, we find that

$$\frac{\partial G}{\partial z} = \sum_n G_n'(z)t^n = \frac{-t}{p(z,t)} + \frac{t(b-zt)(b+1-2t)}{2p(z,t)^2};$$
$$\frac{\partial^2 G}{\partial z^2} = \sum_n G_n''(z)t^n = \frac{-t^2(b+1-2t)}{p(z,t)^2} + \frac{t^2(b-zt)(b+1-2t)}{p(z,t)^3}.$$

Now let us put $z = 1$ and expand in partial fractions:

$$\sum_n G_n'(1)t^n = \frac{t}{2}\left(\frac{1}{(1-t)^2} - \frac{1}{(b-1)(1-t)} + \frac{1}{(b-1)(b-t)}\right),$$
$$\sum_n G_n''(1)t^n = \frac{t^2}{2}\left(\frac{1}{(1-t)^3} - \frac{1}{(b-1)^2(1-t)} + \frac{1}{(b-1)^2(b-t)} + \frac{1}{(b-1)(b-t)^2}\right).$$

It follows that the average number of carries, i.e., the mean value of K, is

$$G_n'(1) = \frac{1}{2}\left(n - \frac{1}{b-1}\left(1 - \left(\frac{1}{b}\right)^n\right)\right); \tag{6}$$

the variance is

$$G_n''(1) + G_n'(1) - G_n'(1)^2 = \frac{1}{4}\left(n + \frac{2n}{b-1} - \frac{2b+1}{(b-1)^2} + \frac{2b+2}{(b-1)^2}\left(\frac{1}{b}\right)^n - \frac{1}{(b-1)^2}\left(\frac{1}{b}\right)^{2n}\right). \tag{7}$$

So the number of carries is just slightly less than $\frac{1}{2}n$ under these assumptions.

History and bibliography. The early history of the classical algorithms described in this section is left as an interesting project for the reader, and only the history of their implementation on computers will be traced here.

The use of 10^n as an assumed radix when multiplying large numbers on a desk calculator was discussed by D. N. Lehmer and J. P. Ballantine, *AMM* **30** (1923), 67–69.

Double-precision arithmetic on digital computers was first treated by J. von Neumann and H. H. Goldstine in their introductory notes on programming, originally published in 1947 [J. von Neumann, *Collected Works* **5**, 142–151]. Theorems A and B above are due to D. A. Pope and M. L. Stein [*CACM* **3** (1960), 652–654], whose paper also contains a bibliography of earlier work on double-precision routines. Other ways of choosing the trial quotient $\hat{q}$ have been discussed by A. G. Cox and H. A. Luther, *CACM* **4** (1961), 353 [divide by $v_1 + 1$ instead of v_1], and by M. L. Stein, *CACM* **7** (1964), 472–474 [divide by v_1 or

$v_1 + 1$ according to the magnitude of v_2]; E. V. Krishnamurthy [*CACM* **8** (1965), 179–181] showed that examination of the single-precision remainder in the latter method leads to an improvement over Theorem B. Krishnamurthy and Nandi [*CACM* **10** (1967), 809–813] suggested a way to replace the normalization and unnormalization operations of Algorithm D by a calculation of $\hat{q}$ based on several leading digits of the operands. G. E. Collins and D. R. Musser have carried out an interesting analysis of the original Pope and Stein algorithm [*Inf. Proc. Letters* **6** (1977), 151–155].

Several alternative methods for division have also been suggested:

1) "Fourier division" [J. Fourier, *Analyse des équations déterminées* (Paris, 1831), §2.21]. This method, which was often used on desk calculators, essentially obtains each new quotient digit by increasing the precision of the divisor and the dividend at each step. Some rather extensive tests by the author have indicated that such a method is inferior to the divide-and-correct technique above, but there may be some applications in which Fourier division is practical. See D. H. Lehmer, *AMM* **33** (1926), 198–206; J. V. Uspensky, *Theory of Equations* (New York: McGraw-Hill, 1948), 159–164.

2) "Newton's method" for evaluating the reciprocal of a number was extensively used in early computers when there was no single-precision division instruction. The idea is to find some initial approximation x_0 to the number $1/v$, then to let $x_{n+1} = 2x_n - vx_n^2$. This method converges rapidly to $1/v$, since $x_n = (1-\epsilon)/v$ implies that $x_{n+1} = (1-\epsilon^2)/v$. Convergence to third order, i.e., with ϵ replaced by $O(\epsilon^3)$ at each step, can be obtained using the formula

$$\begin{aligned} x_{n+1} &= x_n + x_n(1 - vx_n) + x_n(1 - vx_n)^2 \\ &= x_n\big(1 + (1 - vx_n)(1 + (1 - vx_n))\big), \end{aligned}$$

and similar formulas hold for fourth-order convergence, etc.; see P. Rabinowitz, *CACM* **4** (1961), 98. For calculations on extremely large numbers, Newton's second-order method and subsequent multiplication by u can actually be considerably faster than Algorithm D, if we increase the precision of x_n at each step and if we also use the fast multiplication routines of Section 4.3.3. (See Algorithm 4.3.3D for details.) Some related iterative schemes have been discussed by E. V. Krishnamurthy, *IEEE Trans.* **C–19** (1970), 227–231.

3) Division methods have also been based on the evaluation of

$$\frac{u}{v+\epsilon} = \frac{u}{v}\left(1 - \left(\frac{\epsilon}{v}\right) + \left(\frac{\epsilon}{v}\right)^2 - \left(\frac{\epsilon}{v}\right)^3 + \cdots\right).$$

See H. H. Laughlin, *AMM* **37** (1930), 287–293. We have used this idea in the double-precision case (Eq. 4.2.3–3).

Besides the references just cited, the following early articles concerning multiple-precision arithmetic are of interest: High-precision routines for floating point calculations using ones' complement arithmetic are described by A. H.

Stroud and D. Secrest, *Comp. J.* **6** (1963), 62–66. Extended-precision subroutines for use in FORTRAN programs are described by B. I. Blum, *CACM* **8** (1965), 318–320; and for use in ALGOL by M. Tienari and V. Suokonautio, *BIT* **6** (1966), 332–338. Arithmetic on integers with *unlimited* precision, making use of linked memory allocation techniques, has been elegantly described by G. E. Collins, *CACM* **9** (1966), 578–589. For a much larger repertoire of operations, including logarithms and trigonometric functions, see R. P. Brent, *ACM Trans. Math. Software* **4** (1978), 57–81.

We have restricted our discussion in this section to arithmetic techniques for use in computer programming. There are many algorithms for *hardware* implementation of arithmetic operations that are very interesting, but they appear to be inapplicable to computer programs for high-precision numbers; for example, see G. W. Reitwiesner, "Binary Arithmetic," *Advances in Computers* **1** (New York: Academic Press, 1960), 231–308; O. L. MacSorley, *Proc. IRE* **49** (1961), 67–91; G. Metze, *IRE Trans.* **EC–11** (1962), 761–764; H. L. Garner, "Number Systems and Arithmetic," *Advances in Computers* **6** (New York: Academic Press, 1965), 131–194. The minimum achievable execution time for hardware addition and multiplication operations has been investigated by S. Winograd, *JACM* **12** (1965), 277–285, **14** (1967), 793–802; by R. P. Brent, *IEEE Trans.* **C–19** (1970), 758–759; and by R. W. Floyd, *IEEE Symp. Found. Comp. Sci.* **16** (1975), 3–5.

EXERCISES

1. [*42*] Study the early history of the classical algorithms for arithmetic by looking up the writings of, say, Sun Tsŭ, al-Khwârizmî, al-Uqlîdisî, Fibonacci, and Robert Recorde, and by translating their methods as faithfully as possible into precise algorithmic notation.

2. [*15*] Generalize Algorithm A so that it does "column addition," i.e., obtains the sum of m nonnegative n-place integers. (Assume that $m \leq b$.)

3. [*21*] Write a MIX program for the algorithm of exercise 2, and estimate its running time as a function of m and n.

4. [*M21*] Give a formal proof of the validity of Algorithm A, using the method of "inductive assertions" explained in Section 1.2.1.

5. [*21*] Algorithm A adds the two inputs by going from right to left, but sometimes the data is more readily accessible from left to right. Design an algorithm that produces the same answer as Algorithm A, but that generates the digits of the answer from left to right, going back to change previous values if a carry occurs to make a previous value incorrect. [*Note:* Early Hindu and Arabic manuscripts dealt with addition from left to right in this way, probably because it was customary to work from left to right on an abacus; the right-to-left addition algorithm was a refinement due to al-Uqlîdisî, perhaps because Arabic is written from right to left.]

▸ **6.** [*22*] Design an algorithm that adds from left to right (as in exercise 5), but your algorithm should not store a digit of the answer until this digit cannot possibly be affected by future carries; there is to be no changing of any answer digit once it has been stored. [*Hint:* Keep track of the number of consecutive $(b-1)$'s that have not yet been stored in the answer.] This sort of algorithm would be appropriate, for example, in a situation where the input and output numbers are to be read and written from left to right on magnetic tapes, or if they appear in straight linear lists.

7. [*M26*] Determine the average number of times the algorithm of exercise 5 will find that a carry makes it necessary to go back and change k digits of the partial answer, for $k = 1, 2, \ldots, n$. (Assume that both inputs are independently and uniformly distributed between 0 and $b^n - 1$.)

8. [*M26*] Write a MIX program for the algorithm of exercise 5, and determine its average running time based on the expected number of carries as computed in the text.

▸ **9.** [*21*] Generalize Algorithm A to obtain an algorithm that adds two n-place numbers in a *mixed-radix* number system, with bases $b_0, b_1, \ldots$ (from right to left). Thus the least significant digits lie between 0 and $b_0 - 1$, the next digits lie between 0 and $b_1 - 1$, etc.; cf. Eq. 4.1–9.

10. [*18*] Would Program S work properly if the instructions on lines 06 and 07 were interchanged? If the instructions on lines 05 and 06 were interchanged?

11. [*10*] Design an algorithm that compares two nonnegative n-place integers $u = (u_1 u_2 \ldots u_n)_b$ and $v = (v_1 v_2 \ldots v_n)_b$, to determine whether $u < v$, $u = v$, or $u > v$.

12. [*16*] Algorithm S assumes that we know which of the two input operands is the larger; if this information is not known, we could go ahead and perform the subtraction anyway, and we would find that an extra "borrow" is still present at the end of the algorithm. Design another algorithm that could be used (if there is a "borrow" present at the end of Algorithm S) to complement $(w_1 w_2 \ldots w_n)_b$ and therefore to obtain the absolute value of the difference of u and v.

13. [*21*] Write a MIX program that multiplies $(u_1 u_2 \ldots u_n)_b$ by v, where v is a single-precision number (i.e., $0 \le v < b$), producing the answer $(w_0 w_1 \ldots w_n)_b$. How much running time is required?

▸ **14.** [*M24*] Give a formal proof of the validity of Algorithm M, using the method of "inductive assertions" explained in Section 1.2.1.

15. [*M20*] If we wish to form the product of two n-place fractions, $(.u_1 u_2 \ldots u_n)_b \times (.v_1 v_2 \ldots v_n)_b$, and to obtain only an n-place approximation $(.w_1 w_2 \ldots w_n)_b$ to the result, Algorithm M could be used to obtain a $2n$-place answer that is subsequently rounded to the desired approximation. But this involves about twice as much work as is necessary for reasonable accuracy, since the products $u_i v_j$ for $i + j > n + 2$ contribute very little to the answer.

Give an estimate of the maximum error that can occur, if these products $u_i v_j$ for $i + j > n + 2$ are not computed during the multiplication, but are assumed to be zero.

▸ **16.** [*20*] Design an algorithm that divides a nonnegative n-place integer $(u_1 u_2 \ldots u_n)_b$ by v, where v is a single-precision number (i.e., $0 < v < b$), producing the quotient $(w_1 w_2 \ldots w_n)_b$ and remainder r.

17. [*M20*] In the notation of Fig. 6, assume that $v_1 \ge \lfloor b/2 \rfloor$; show that if $u_0 = v_1$, we must have $q = b - 1$ or $b - 2$.

18. [*M20*] In the notation of Fig. 6, show that if $q' = \lfloor (u_0 b + u_1)/(v_1 + 1) \rfloor$, then $q' \le q$.

▶ **19.** [*M21*] In the notation of Fig. 6, let $\hat{q}$ be an approximation to q, and let $\hat{r} = u_0 b + u_1 - \hat{q} v_1$. Assume that $v_1 > 0$. Show that if $v_2 \hat{q} > b\hat{r} + u_2$, then $q < \hat{q}$. [*Hint:* Strengthen the proof of Theorem A by examining the influence of v_2.]

20. [*M22*] Using the notation and assumptions of exercise 19, show that if $v_2 \hat{q} \le b\hat{r} + u_2$, then $\hat{q} = q$ or $q = \hat{q} - 1$.

▶ **21.** [*M23*] Show that if $v_1 \ge \lfloor b/2 \rfloor$, and if $v_2 \hat{q} \le b\hat{r} + u_2$ but $\hat{q} \ne q$ in the notation of exercises 19 and 20, then $u \bmod v \ge (1 - 2/b)v$. (The latter event occurs with approximate probability $2/b$, so that when b is the word size of a computer we must have $q_j = \hat{q}$ in Algorithm D except in very rare circumstances.)

▶ **22.** [*24*] Find an example of a four-digit number divided by a three-digit number for which step D6 is necessary in Algorithm D, when the radix b is 10.

23. [*M23*] Given that v and b are integers, and that $1 \le v < b$, prove that we always have $\lfloor b/2 \rfloor \le v\lfloor b/(v+1) \rfloor < (v+1)\lfloor b/(v+1) \rfloor \le b$.

24. [*M20*] Using the law of the distribution of leading digits explained in Section 4.2.4, give an approximate formula for the probability that $d = 1$ in Algorithm D. (When $d = 1$, it is, of course, possible to omit most of the calculation in steps D1 and D8.)

25. [*26*] Write a MIX routine for step D1, which is needed to complete Program D.

26. [*21*] Write a MIX routine for step D8, which is needed to complete Program D.

27. [*M20*] Prove that at the beginning of step D8 in Algorithm D, the unnormalized remainder $(u_{m+1} u_{m+2} \ldots u_{m+n})_b$ is always an exact multiple of d.

28. [*M30*] (A. Svoboda, *Stroje na Zpracování Informací* **9** (1963), 25–32.) Let $v = (v_1 v_2 \ldots v_n)_b$ be any radix b integer, where $v_1 \ne 0$. Perform the following operations:

N1. If $v_1 < b/2$, multiply v by $\lfloor (b+1)/(v_1+1) \rfloor$. Let the result of this step be $(v_0 v_1 v_2 \ldots v_n)_b$.

N2. If $v_0 = 0$, set $v \leftarrow v + (1/b)\lfloor b(b - v_1)/(v_1 + 1) \rfloor v$; let the result of this step be $(v_0 v_1 v_2 \ldots v_n . v_{n+1} \ldots)_b$. Repeat step N2 until $v_0 \ne 0$.

Prove that step N2 will be performed at most three times, and that we must always have $v_0 = 1$, $v_1 = 0$ at the end of the calculations.

[*Note:* If u and v are both multiplied by the above constants, we do not change the value of the quotient u/v, and the divisor has been converted into the form $(10 v_2 \ldots v_n . v_{n+1} v_{n+2} v_{n+3})_b$. This form of the divisor is very convenient because, in the notation of Algorithm D, we may simply take $\hat{q} = u_j$ as a trial divisor at the beginning of step D3, or $\hat{q} = b - 1$ when $(u_{j-1}, u_j) = (1, 0)$.]

29. [*15*] Prove or disprove: At the beginning of step D7 of Algorithm D, we always have $u_j = 0$.

▶ **30.** [*22*] If memory space is limited, it may be desirable to use the same storage locations for both input and output during the performance of some of the algorithms in this section. Is it possible to have $w_1, \ldots, w_n$ stored in the same respective locations as $u_1, \ldots, u_n$ or $v_1, \ldots, v_n$ during Algorithm A or S? Is it possible to have the quotient $q_0, \ldots, q_m$ occupy the same locations as $u_0, \ldots, u_m$ in Algorithm D? Is there any permissible overlap of memory locations between input and output in Algorithm M?

31. [28] Assume that $b = 3$ and that $u = (u_1 \ldots u_{m+n})_3$, $v = (v_1 \ldots v_n)_3$ are integers in *balanced ternary* notation (cf. Section 4.1), $v_1 \neq 0$. Design a long-division algorithm that divides u by v, obtaining a remainder whose absolute value does not exceed $\frac{1}{2}|v|$. Try to find an algorithm that would be efficient if incorporated into the arithmetic circuitry of a balanced ternary computer.

32. [M40] Assume that $b = 2i$ and that u and v are complex numbers expressed in the quater-imaginary number system. Design algorithms that divide u by v, perhaps obtaining a suitable remainder of some sort, and compare their efficiency. [*References:* M. Nadler, *CACM* **4** (1961), 192–193; Z. Pawlak and A. Wakulicz, *Bull. de l'Acad. Polonaise des Sciences*, Classe III, **5** (1957), 233–236 (see also pp. 803–804); and exercise 4.1–15.]

33. [M40] Design an algorithm for taking square roots, analogous to Algorithm D and to the traditional pencil-and-paper method for extracting square roots.

34. [40] Develop a set of computer subroutines for doing the four arithmetic operations on arbitrary integers, putting no constraint on the size of the integers except for the implicit assumption that the total memory capacity of the computer should not be exceeded. (Use linked memory allocation, so that no time is wasted in finding room to put the results.)

35. [40] Develop a set of computer subroutines for "decuple-precision floating point" arithmetic, using excess 0, base b, nine-place floating point number representation, where b is the computer word size, and allowing a full word for the exponent. (Thus each floating point number is represented in 10 words of memory, and all scaling is done by moving full words instead of by shifting within the words.)

36. [M42] Compute the values of the fundamental constants listed in Appendix B to much higher precision than the 40-place values listed there. [*Note:* The first 100,000 digits of the decimal expansion of π were published by D. Shanks and J. W. Wrench, Jr., in *Math. Comp.* **16** (1962), 76–99. One million digits of π were computed by Jean Guilloud and Martine Bouyer of the French Atomic Energy Commission in 1974.]

▶ **37.** [20] (E. Salamin.) Explain how to avoid the normalization and unnormalization steps of Algorithm D, when d is a power of 2 on a binary computer, without changing the sequence of trial quotient digits computed by that algorithm. (How can $\hat{q}$ be computed in step D3 if the normalization of step D1 hasn't been done?)

*4.3.2. Modular Arithmetic

Another interesting alternative is available for doing arithmetic on large integer numbers, based on some simple principles of number theory. The idea is to have several "moduli" $m_1, m_2, \ldots, m_r$ that contain no common factors, and to work indirectly with "residues" $u \bmod m_1, u \bmod m_2, \ldots, u \bmod m_r$ instead of directly with the number u.

For convenience in notation throughout this section, let

$$u_1 = u \bmod m_1, \qquad u_2 = u \bmod m_2, \qquad \ldots, \qquad u_r = u \bmod m_r. \tag{1}$$

It is easy to compute $(u_1, u_2, \ldots, u_r)$ from an integer number u by means of division; and it is important to note that no information is lost in this process,

since we can recompute u from $(u_1, u_2, \ldots, u_r)$ provided that u is not too large. For example, if $0 \le u < v \le 1000$, it is impossible to have $(u \bmod 7, u \bmod 11, u \bmod 13)$ equal to $(v \bmod 7, v \bmod 11, v \bmod 13)$. This is a consequence of the "Chinese remainder theorem" stated below.

We may therefore regard $(u_1, u_2, \ldots, u_r)$ as a new type of internal computer representation, a "modular representation," of the integer u.

The advantages of a modular representation are that addition, subtraction, and multiplication are very simple:

$$(u_1, \ldots, u_r) + (v_1, \ldots, v_r) = ((u_1 + v_1) \bmod m_1, \ldots, (u_r + v_r) \bmod m_r), \quad (2)$$
$$(u_1, \ldots, u_r) - (v_1, \ldots, v_r) = ((u_1 - v_1) \bmod m_1, \ldots, (u_r - v_r) \bmod m_r), \quad (3)$$
$$(u_1, \ldots, u_r) \times (v_1, \ldots, v_r) = ((u_1 \times v_1) \bmod m_1, \ldots, (u_r \times v_r) \bmod m_r). \quad (4)$$

To derive (4), for example, we need to show that

$$uv \bmod m_j = (u \bmod m_j)(v \bmod m_j) \bmod m_j$$

for each modulus m_j. But this is a basic fact of elementary number theory: $x \bmod m_j = y \bmod m_j$ if and only if $x \equiv y$ (modulo m_j); furthermore if $x \equiv x'$ and $y \equiv y'$, then $xy \equiv x'y'$ (modulo m_j); hence $(u \bmod m_j)(v \bmod m_j) \equiv uv$ (modulo m_j).

The disadvantages of a modular representation are that it is comparatively difficult to test whether a number is positive or negative or to test whether or not $(u_1, \ldots, u_r)$ is greater than $(v_1, \ldots, v_r)$. It is also difficult to test whether or not overflow has occurred as the result of an addition, subtraction, or multiplication, and it is even more difficult to perform division. When these operations are required frequently in conjunction with addition, subtraction, and multiplication, the use of modular arithmetic can be justified only if fast means of conversion into and out of the modular representation are available. Therefore conversion between modular and positional notation is one of the principal topics of interest to us in this section.

The processes of addition, subtraction, and multiplication using (2), (3), and (4) are called residue arithmetic or *modular arithmetic*. The range of numbers that can be handled by modular arithmetic is equal to $m = m_1 m_2 \ldots m_r$, the product of the moduli. Therefore we see that the amount of time required to add, subtract, or multiply n-digit numbers using modular arithmetic is essentially proportional to n (not counting the time to convert in and out of modular representation). This is no advantage at all when addition and subtraction are considered, but it can be a considerable advantage with respect to multiplication since the conventional method of the preceding section requires an execution time proportional to n^2.

Moreover, on a computer that allows many operations to take place simultaneously, modular arithmetic can be a significant advantage even for addition and subtraction; the operations with respect to different moduli can all be done at the same time, so we obtain a substantial increase in speed. The same kind of

decrease in execution time could not be achieved by the conventional techniques discussed in the previous section, since carry propagation must be considered. Perhaps some day highly parallel computers will make simultaneous operations commonplace, so that modular arithmetic will be of significant importance in "real-time" calculations when a quick answer to a single problem requiring high precision is needed. (With highly parallel computers, it is often preferable to run k *separate* programs simultaneously, instead of running a *single* program k times as fast, since the latter alternative is more complicated but does not utilize the machine any more efficiently. "Real-time" calculations are exceptions that make the inherent parallelism of modular arithmetic more significant.)

Now let us examine the basic fact that underlies the modular representation of numbers:

Theorem C (*Chinese Remainder Theorem*). *Let m_1, m_2, ..., m_r be positive integers that are relatively prime in pairs, i.e.,*

$$\gcd(m_j, m_k) = 1 \qquad \text{when } j \neq k. \tag{5}$$

Let $m = m_1 m_2 \ldots m_r$, and let a, u_1, u_2, ..., u_r be integers. Then there is exactly one integer u that satisfies the conditions

$$a \leq u < a + m, \qquad \text{and} \qquad u \equiv u_j \text{ (modulo } m_j) \text{ for } 1 \leq j \leq r. \tag{6}$$

Proof. If $u \equiv v$ (modulo m_j) for $1 \leq j \leq r$, then $u - v$ is a multiple of m_j for all j, so (5) implies that $u - v$ is a multiple of $m = m_1 m_2 \ldots m_r$. This argument shows that there is *at most* one solution of (6). To complete the proof we must now show the existence of *at least* one solution, and this can be done in two simple ways:

METHOD 1 ("Nonconstructive" proof). As u runs through the m distinct values $a \leq u < a + m$, the r-tuples $(u \bmod m_1, \ldots, u \bmod m_r)$ must also run through m distinct values, since (6) has at most one solution. But there are exactly $m_1 m_2 \ldots m_r$ possible r-tuples $(v_1, \ldots, v_r)$ such that $0 \leq v_j < m_j$. Therefore each r-tuple must occur exactly once, and there must be some value of u for which $(u \bmod m_1, \ldots, u \bmod m_r) = (u_1, \ldots, u_r)$.

METHOD 2 ("Constructive" proof). We can find numbers M_j for $1 \leq j \leq r$ such that

$$M_j \equiv 1 \text{ (modulo } m_j) \qquad \text{and} \qquad M_j \equiv 0 \text{ (modulo } m_k) \text{ for } k \neq j. \tag{7}$$

This follows because (5) implies that m_j and m/m_j are relatively prime, so we may take

$$M_j = (m/m_j)^{\varphi(m_j)} \tag{8}$$

by Euler's theorem (exercise 1.2.4–28). Now the number

$$u = a + \big((u_1 M_1 + u_2 M_2 + \cdots + u_r M_r - a) \bmod m\big) \tag{9}$$

satisfies all the conditions of (6). ∎

A very special case of this theorem was stated by the Chinese mathematician Sun Tsŭ, who gave a rule called tái-yen ("great generalization"). The date of his writing is very uncertain; it is thought to be between 280 and 473 A.D. [See Joseph Needham, *Science and Civilization in China* **3** (Cambridge University Press, 1959), 33–34, 119–120, for an interesting discussion.] Theorem C was apparently first stated and proved in its proper generality by Chhin Chiu Shao in his *Shu Shu Chiu Chang* (1247). Numerous early contributions to this theory have been summarized by L. E. Dickson in his *History of the Theory of Numbers* **2** (New York: Chelsea, 1952), 57–64.

As a consequence of Theorem C, we may use modular representation for numbers in any consecutive interval of $m = m_1 m_2 \dots m_r$ integers. For example, we could take $a = 0$ in (6), and work only with nonnegative integers u less than m. On the other hand, when addition and subtraction are being done, as well as multiplication, it is usually most convenient to assume that all the moduli $m_1, m_2, \dots, m_r$ are odd numbers, so that $m = m_1 m_2 \dots m_r$ is odd, and to work with integers in the range

$$-\frac{m}{2} < u < \frac{m}{2}, \tag{10}$$

which is completely symmetrical about zero.

To perform the basic operations listed in (2), (3), and (4), we need to compute $(u_j + v_j) \bmod m_j$, $(u_j - v_j) \bmod m_j$, and $u_j v_j \bmod m_j$, when $0 \le u_j, v_j < m_j$. If m_j is a single-precision number, it is most convenient to form $u_j v_j \bmod m_j$ by doing a multiplication and then a division operation. For addition and subtraction, the situation is a little simpler, since no division is necessary; the following formulas may conveniently be used:

$$(u_j + v_j) \bmod m_j = \begin{cases} u_j + v_j, & \text{if } u_j + v_j < m_j; \\ u_j + v_j - m_j, & \text{if } u_j + v_j \ge m_j. \end{cases} \tag{11}$$

$$(u_j - v_j) \bmod m_j = \begin{cases} u_j - v_j, & \text{if } u_j - v_j \ge 0; \\ u_j - v_j + m_j, & \text{if } u_j - v_j < 0. \end{cases} \tag{12}$$

(Cf. Section 3.2.1.1.) In this case, since we want m to be as large as possible, it is easiest to let m_1 be the largest odd number that fits in a computer word, to let m_2 be the largest odd number $< m_1$ that is relatively prime to m_1, to let m_3 be the largest odd number $< m_2$ that is relatively prime to both m_1 and m_2, and so on until enough m_j's have been found to give the desired range m. Efficient ways to determine whether or not two integers are relatively prime are discussed in Section 4.5.2.

As a simple example, suppose that we have a decimal computer whose words hold only two digits, so that the word size is 100. Then the procedure described in the previous paragraph would give

$$m_1 = 99, \quad m_2 = 97, \quad m_3 = 95, \quad m_4 = 91, \quad m_5 = 89, \quad m_6 = 83, \tag{13}$$

and so on.

On binary computers it is sometimes desirable to choose the m_j in a different way, by selecting

$$m_j = 2^{e_j} - 1. \tag{14}$$

In other words, each modulus is one less than a power of 2. Such a choice of m_j often makes the basic arithmetic operations simpler, because it is relatively easy to work modulo $2^{e_j} - 1$, as in ones' complement arithmetic. When the moduli are chosen according to this strategy, it is helpful to relax the condition $0 \le u_j < m_j$ slightly, so that we require only

$$0 \le u_j < 2^{e_j}, \qquad u_j \equiv u \text{ (modulo } 2^{e_j} - 1). \tag{15}$$

Thus, the value $u_j = m_j = 2^{e_j} - 1$ is allowed as an optional alternative to $u_j = 0$, since this does not affect the validity of Theorem C, and it means we are allowing u_j to be any e_j-bit binary number. Under this assumption, the operations of addition and multiplication modulo m_j become the following:

$$u_j \oplus v_j = \begin{cases} u_j + v_j, & \text{if } u_j + v_j < 2^{e_j}; \\ ((u_j + v_j) \bmod 2^{e_j}) + 1, & \text{if } u_j + v_j \ge 2^{e_j}. \end{cases} \tag{16}$$

$$u_j \otimes v_j = (u_j v_j \bmod 2^{e_j}) \oplus \lfloor u_j v_j / 2^{e_j} \rfloor. \tag{17}$$

(Here $\oplus$ and $\otimes$ refer to the operations done on the individual components of $(u_1, \ldots, u_r)$ and $(v_1, \ldots, v_r)$ when adding or multiplying, respectively, using the convention (15).) Equation (12) may be used for subtraction. These operations can be performed efficiently even when m_j is larger than the computer's word size, since it is a simple matter to compute the remainder of a positive number modulo a power of 2, or to divide a number by a power of 2. In (17) we have the sum of the "upper half" and the "lower half" of the product, as discussed in exercise 3.2.1.1–8.

If moduli of the form $2^{e_j} - 1$ are to be used, we must know under what conditions the number $2^e - 1$ is relatively prime to the number $2^f - 1$. Fortunately, there is a very simple rule,

$$\gcd(2^e - 1, 2^f - 1) = 2^{\gcd(e,f)} - 1, \tag{18}$$

which states in particular that $2^e - 1$ *and* $2^f - 1$ *are relatively prime if and only if* e *and* f *are relatively prime.* Equation (18) follows from Euclid's algorithm and the identity

$$(2^e - 1) \bmod (2^f - 1) = 2^{e \bmod f} - 1. \tag{19}$$

(See exercise 6.) Thus we could choose for example $m_1 = 2^{35} - 1$, $m_2 = 2^{34} - 1$, $m_3 = 2^{33} - 1$, $m_4 = 2^{31} - 1$, $m_5 = 2^{29} - 1$, if we had a computer with word size 2^{35}; this would permit efficient addition, subtraction, and multiplication of integers in a range of size $m_1 m_2 m_3 m_4 m_5 > 2^{161}$.

As we have already observed, the operations of conversion to and from modular representation are very important. If we are given a number u, its modular representation $(u_1, \ldots, u_r)$ may be obtained by simply dividing u by $m_1, \ldots, m_r$ and saving the remainders. A possibly more attractive procedure, if $u = (v_m v_{m-1} \ldots v_0)_b$, is to evaluate the polynomial

$$(\ldots(v_m b + v_{m-1})b + \cdots)b + v_0$$

using modular arithmetic. When $b = 2$ and when the modulus m_j has the special form $2^{e_j} - 1$, both of these methods reduce to quite a simple procedure: Consider the binary representation of u with blocks of e_j bits grouped together,

$$u = a_t A^t + a_{t-1}A^{t-1} + \cdots + a_1 A + a_0, \tag{20}$$

where $A = 2^{e_j}$ and $0 \le a_k < 2^{e_j}$ for $0 \le k \le t$. Then

$$u \equiv a_t + a_{t-1} + \cdots + a_1 + a_0 \text{ (modulo } 2^{e_j} - 1), \tag{21}$$

since $A \equiv 1$, so we obtain u_j by adding the e_j-bit numbers $a_t \oplus \cdots \oplus a_1 \oplus a_0$, using (16). This process is similar to the familiar device of "casting out nines" that determines $u \bmod 9$ when u is expressed in the decimal system.

Conversion back from modular form to positional notation is somewhat more difficult. It is interesting in this regard to make a few side remarks about the way computers make us change our viewpoint towards mathematical proofs: Theorem C tells us that the conversion from $(u_1, \ldots, u_r)$ to u is possible, and two proofs are given. The first proof we considered is a classical one that relies only on very simple concepts, namely the facts that

i) any number that is a multiple of m_1, of m_2, ..., and of m_r, must be a multiple of $m_1 m_2 \ldots m_r$ when the m_j's are pairwise relatively prime; and

ii) if m things are put into m boxes with no two things in the same box, then there must be one in each box.

By traditional notions of mathematical aesthetics, this is no doubt the nicest proof of Theorem C; but from a computational standpoint it is completely worthless. It amounts to saying, "Try $u = a$, $a + 1$, ... until you find a value for which $u \equiv u_1$ (modulo m_1), ..., $u \equiv u_r$ (modulo m_r)."

The second proof of Theorem C is more explicit; it shows how to compute r new constants $M_1, \ldots, M_r$, and to get the solution in terms of these constants by formula (9). This proof uses more complicated concepts (for example, Euler's theorem), but it is much more satisfactory from a computational standpoint, since the constants $M_1, \ldots, M_r$ need to be determined only once. On the other hand, the determination of M_j by Eq. (8) is certainly not trivial, since the evaluation of Euler's φ-function requires, in general, the factorization of m_j into prime powers. Furthermore, M_j is likely to be a terribly large number, even if we compute only the quantity $M_j \bmod m$ (which will work just as well as M_j in (9)).

Since $M_j \bmod m$ is uniquely determined if (7) is to be satisfied (because of the Chinese remainder theorem), we can see that, in any event, Eq. (9) requires a lot of high-precision calculation, and such calculation is just what we wished to avoid by modular arithmetic in the first place.

So we need an even *better* proof of Theorem C if we are going to have a really usable method of conversion from $(u_1, \ldots, u_r)$ to u. Such a method was suggested by H. L. Garner in 1958; it can be carried out using $\binom{r}{2}$ constants c_{ij} for $1 \le i < j \le r$, where

$$c_{ij}\, m_i \equiv 1 \quad (\text{modulo } m_j). \tag{22}$$

These constants c_{ij} are readily computed using Euclid's algorithm, since for any given i and j Algorithm 4.5.2X will determine a and b such that $am_i + bm_j = \gcd(m_i, m_j) = 1$, and we may take $c_{ij} = a$. When the moduli have the special form $2^{e_j} - 1$, a simple method of determining c_{ij} is given in exercise 6.

Once the c_{ij} have been determined satisfying (22), we can set

$$\begin{aligned}
v_1 &\leftarrow u_1 \bmod m_1,\\
v_2 &\leftarrow (u_2 - v_1)\, c_{12} \bmod m_2,\\
v_3 &\leftarrow \big((u_3 - v_1)\, c_{13} - v_2\big)\, c_{23} \bmod m_3,\\
&\;\;\vdots\\
v_r &\leftarrow \big(\ldots((u_r - v_1)\, c_{1r} - v_2)\, c_{2r} - \cdots - v_{r-1}\big)\, c_{(r-1)r} \bmod m_r.
\end{aligned} \tag{23}$$

Then

$$u = v_r m_{r-1} \ldots m_2 m_1 + \cdots + v_3 m_2 m_1 + v_2 m_1 + v_1 \tag{24}$$

is a number satisfying the conditions

$$0 \le u < m, \qquad u \equiv u_j \ (\text{modulo } m_j) \quad \text{for } 1 \le j \le r. \tag{25}$$

(See exercise 8; another way of rewriting (23) that does not involve as many auxiliary constants is given in exercise 7.) Equation (24) is a *mixed-radix representation* of u, which may be converted to binary or decimal notation using the methods of Section 4.4. If $0 \le u < m$ is not the desired range, an appropriate multiple of m can be added or subtracted after the conversion process.

The advantage of the computation shown in (23) is that the calculation of v_j can be done using only arithmetic mod m_j, which is already built into the modular arithmetic algorithms. Furthermore, (23) allows parallel computation: We can start with $(v_1, \ldots, v_r) \leftarrow (u_1 \bmod m_1, \ldots, u_r \bmod m_r)$, then at time j for $1 \le j < r$ we simultaneously set $v_k \leftarrow (v_k - v_j)\, c_{jk} \bmod m_k$ for $j < k \le r$. An alternative way to compute the mixed-radix representation, allowing similar possibilities for parallelism, has been discussed by A. S. Fraenkel, *Proc. ACM Nat. Conf.* **19** (Philadelphia, 1965), E1.4.

It is important to observe that the mixed-radix representation (24) is sufficient to compare the magnitudes of two modular numbers. For if we know that

$0 \le u < m$ and $0 \le u' < m$, then we can tell if $u < u'$ by first doing the conversion to $(v_1, \ldots, v_r)$ and $(v'_1, \ldots, v'_r)$, then testing if $v_r < v'_r$, or if $v_r = v'_r$ and $v_{r-1} < v'_{r-1}$, etc. It is not necessary to convert all the way to binary or decimal notation if we only want to know whether $(u_1, \ldots, u_r)$ is less than $(u'_1, \ldots, u'_r)$.

The operation of comparing two numbers, or of deciding if a modular number is negative, is intuitively very simple, so we would expect to have a much easier method for making this test than the conversion to mixed-radix form. But the following theorem shows that there is little hope of finding a substantially better method, since the range of a modular number depends essentially on all bits of all the residues $(u_1, \ldots, u_r)$:

Theorem S (Nicholas Szabó, 1961). *In terms of the notation above, assume that* $m_1 < \sqrt{m}$, *and let* L *be any value in the range*

$$m_1 \le L \le m - m_1. \tag{26}$$

Let g *be any function such that the set* $\{g(0), g(1), \ldots, g(m_1 - 1)\}$ *contains fewer than* m_1 *values. Then there are numbers* u *and* v *such that*

$$g(u \bmod m_1) = g(v \bmod m_1), \qquad u \bmod m_j = v \bmod m_j \quad \text{for } 2 \le j \le r; \tag{27}$$

$$0 \le u < L \le v < m. \tag{28}$$

Proof. By hypothesis, there must exist numbers $u \neq v$ satisfying (27), since g must take on the same value for two different residues. Let (u, v) be a pair of values with $0 \le u < v < m$ satisfying (27), for which u is a minimum. Since $u' = u - m_1$ and $v' = v - m_1$ also satisfy (27), we must have $u' < 0$ by the minimality of u. Hence $u < m_1 \le L$; and if (28) does not hold, we must have $v < L$. But $v > u$, and $v - u$ is a multiple of $m_2 \ldots m_r = m/m_1$, so $v \ge v - u \ge m/m_1 > m_1$. Therefore, if (28) does not hold for (u, v), it will be satisfied for the pair $(u'', v'') = (v - m_1, u + m - m_1)$. ∎

Of course, a similar result can be proved for any m_j in place of m_1; and we could also replace (28) by the condition "$a \le u < a + L \le v < a + m$" with only minor changes in the proof. Therefore Theorem S shows that many simple functions cannot be used to determine the range of a modular number.

Let us now reiterate the main points of the discussion in this section: Modular arithmetic can be a significant advantage for applications in which the predominant calculations involve exact multiplication (or raising to a power) of large integers, combined with addition and subtraction, but where there is very little need to divide or compare numbers, *or to test whether intermediate results "overflow" out of range.* (It is important not to forget the latter restriction; methods are available to test for overflow, as in exercise 12, but they are in general so complicated that they nullify the advantages of modular arithmetic.) Several applications of modular computations have been discussed by H. Takahasi and Y. Ishibashi, *Information Proc. in Japan* **1** (1961), 28–42.

An example of such an application is the exact solution of linear equations with rational coefficients. For various reasons it is desirable in this case to assume that the moduli $m_1, m_2, \ldots, m_r$ are all large prime numbers; the linear equations can be solved independently modulo each m_j. A detailed discussion of this procedure has been given by I. Borosh and A. S. Fraenkel [*Math. Comp.* **20** (1966), 107–112]. By means of their method, the nine independent solutions of a system of 111 linear equations in 120 unknowns were obtained exactly in less than one hour's running time on a CDC 1604 computer. The same procedure is worthwhile also for solving simultaneous linear equations with floating point coefficients, when the matrix of coefficients is ill-conditioned. The modular technique (treating the given floating point coefficients as exact rational numbers) gives a method for obtaining the *true* answers in less time than conventional methods can produce reliable *approximate* answers! [See M. T. McClellan, *JACM* **20** (1973), 563–588, for further developments of this approach; and see also E. H. Bareiss, *J. Inst. Math. and Appl.* **10** (1972), 68–104, for a discussion of its limitations.]

The published literature concerning modular arithmetic is mostly oriented towards hardware design, since the carry-free properties of modular arithmetic make it attractive from the standpoint of high-speed operation. The idea was first published by A. Svoboda and M. Valach in the Czechoslovakian journal *Stroje na Zpracování Informací* **3** (1955), 247–295; then independently by H. L. Garner [*IRE Trans.* **EC–8** (1959), 140–147]. The use of moduli of the form $2^{e_j} - 1$ was suggested by A. S. Fraenkel [*JACM* **8** (1961), 87–96], and several advantages of such moduli were demonstrated by A. Schönhage [*Computing* **1** (1966), 182–196]. See the book *Residue Arithmetic and its Applications to Computer Technology* by N. S. Szabó and R. I. Tanaka (New York: McGraw-Hill, 1967), for additional information and a comprehensive bibliography of the subject. A Russian book published in 1968 by I. Ĭa. Akushskiĭ and D. I. Ĭuditskiĭ includes a chapter about complex moduli [cf. *Rev. Romaine des Math.* **15** (1970), 159–160].

Further discussion of modular arithmetic can be found in Section 4.3.3B.

EXERCISES

1. [*20*] Find all integers u that satisfy all of the following conditions: $u \bmod 7 = 1$, $u \bmod 11 = 6$, $u \bmod 13 = 5$, $0 \le u < 1000$.

2. [*M20*] Would Theorem C still be true if we allowed a, u_1, u_2, ..., u_r and u to be arbitrary real numbers (not just integers)?

▶ **3.** [*M26*] (*Generalized Chinese Remainder Theorem.*) Let $m_1, m_2, \ldots, m_r$ be positive integers. Let m be the least common multiple of $m_1, m_2, \ldots, m_r$, and let a, $u_1, u_2, \ldots, u_r$ be any integers. Prove that there is exactly one integer u that satisfies the conditions

$$a \le u < a + m, \qquad u \equiv u_j \text{ (modulo } m_j), \qquad 1 \le j \le r,$$

provided that

$$u_i \equiv u_j \pmod{\gcd(m_i, m_j)}, \qquad 1 \le i < j \le r;$$

and there is no such integer u when the latter condition fails to hold.

4. [*20*] Continue the process shown in (13); what would m_7, m_8, m_9, ... be?

▶ **5.** [*M23*] Suppose that the method of (13) is continued until no more m_j can be chosen; does this method give the largest attainable value $m_1 m_2 \ldots m_r$ such that the m_j are odd positive integers less than 100 that are relatively prime in pairs?

6. [*M22*] Let e, f, g be nonnegative integers. (a) Show that $2^e \equiv 2^f \pmod{2^g - 1}$ if and only if $e \equiv f \pmod{g}$. (b) Given that $e \bmod f = d$ and $ce \bmod f = 1$, prove that

$$\left((1 + 2^d + \cdots + 2^{(c-1)d}) \cdot (2^e - 1)\right) \bmod (2^f - 1) = 1.$$

(Thus, we have a comparatively simple formula for the inverse of $2^e - 1$, modulo $2^f - 1$, as required in (22).)

▶ **7.** [*M21*] Show that (23) can be rewritten as follows:

$$\begin{aligned}
v_1 &\leftarrow u_1 \bmod m_1,\\
v_2 &\leftarrow (u_2 - v_1)\, c_{12} \bmod m_2,\\
v_3 &\leftarrow (u_3 - (v_1 + m_1 v_2))\, c_{13} c_{23} \bmod m_3,\\
&\vdots\\
v_r &\leftarrow (u_r - (v_1 + m_1(v_2 + m_2(v_3 + \cdots + m_{r-2} v_{r-1}) \ldots)))\, c_{1r} \ldots c_{(r-1)r} \bmod m_r.
\end{aligned}$$

If the formulas are rewritten in this way, we see that only $r - 1$ constants $C_j = c_{1j} \ldots c_{(j-1)j} \bmod m_j$ are needed instead of $r(r-1)/2$ constants c_{ij} as in (23). Discuss the relative merits of this version of the formula as compared to (23), from the standpoint of computer calculation.

8. [*M21*] Prove that the number u defined by (23) and (24) satisfies (25).

9. [*M20*] Show how to go from the values $v_1, \ldots, v_r$ of the mixed-radix notation (24) back to the original residues $u_1, \ldots, u_r$, using only arithmetic mod m_j to compute the value of u_j.

10. [*M25*] An integer u that lies in the symmetrical range (10) might be represented by finding the numbers $u_1, \ldots, u_r$ such that $u \equiv u_j \pmod{m_j}$ and $-m_j/2 < u_j < m_j/2$, instead of insisting that $0 \le u_j < m_j$ as in the text. Discuss the modular arithmetic procedures that would be appropriate in connection with such a symmetrical representation (including the conversion process, (23)).

11. [*M23*] Assume that all the m_j are odd, and that $u = (u_1, \ldots, u_r)$ is known to be even, where $0 \le u < m$. Find a reasonably fast method to compute $u/2$ using modular arithmetic.

12. [*M10*] Prove that, if $0 \le u, v < m$, the modular addition of u and v causes overflow (i.e., is outside the range allowed by the modular representation) if and only if the sum is less than u. (Thus the overflow detection problem is equivalent to the comparison problem.)

▸ **13.** [*M25*] (*Automorphic numbers.*) An n-place decimal number $x > 1$ is called an "automorph" by recreational mathematicians if the last n digits of x^2 are equal to x. For example, 9376 is a 4-place automorph, since $9376^2 = 87909376$. [See *Scientific American* **218** (January 1968), 125.]

a) Prove that an n-place number $x > 1$ is an automorph if and only if $x \bmod 5^n = 0$ or 1 and $x \bmod 2^n = 1$ or 0, respectively. (Thus, if $m_1 = 2^n$ and $m_2 = 5^n$, the only two n-place automorphs are the numbers M_1 and M_2 in (7).)

b) Prove that if x is an n-place automorph, then $(3x^2 - 2x^3) \bmod 10^{2n}$ is a $2n$-place automorph.

c) Given that $cx \equiv 1$ (modulo y), find a simple formula for a number c' depending on c and x but not on y, such that $c'x^2 \equiv 1$ (modulo y^2).

*4.3.3. How Fast Can We Multiply?

The conventional method for multiplication in positional number systems, Algorithm 4.3.1M, requires approximately cmn operations to multiply an m-digit number by an n-digit number, where c is a constant. In this section, let us assume for convenience that $m = n$, and let us consider the following question: *Does every general computer algorithm for multiplying two n-digit numbers require an execution time proportional to n^2, as n increases?*

(In this question, a "general" algorithm means one that accepts, as input, the number n and two arbitrary n-digit numbers in positional notation, and whose output is their product in positional form. Certainly if we were allowed to choose a different algorithm for each value of n, the question would be of no interest, since multiplication could be done for any specific value of n by a "table-lookup" operation in some huge table. The term "computer algorithm" is meant to imply an algorithm that is suitable for implementation on a digital computer such as MIX, and the execution time is to be the time it takes to perform the algorithm on such a computer.)

A. Digital methods. The answer to the above question is, rather surprisingly, "No," and, in fact, it is not very difficult to see why. For convenience, let us assume throughout this section that we are working with integers expressed in binary notation. If we have two $2n$-bit numbers $u = (u_{2n-1} \ldots u_1 u_0)_2$ and $v = (v_{2n-1} \ldots v_1 v_0)_2$, we can write

$$u = 2^n U_1 + U_0, \qquad v = 2^n V_1 + V_0, \tag{1}$$

where $U_1 = (u_{2n-1} \ldots u_n)_2$ is the "most significant half" of the number u and $U_0 = (u_{n-1} \ldots u_0)_2$ is the "least significant half"; similarly $V_1 = (v_{2n-1} \ldots v_n)_2$ and $V_0 = (v_{n-1} \ldots v_0)_2$. Now we have

$$uv = (2^{2n} + 2^n)U_1V_1 + 2^n(U_1 - U_0)(V_0 - V_1) + (2^n + 1)U_0V_0. \tag{2}$$

This formula reduces the problem of multiplying $2n$-bit numbers to three multiplications of n-bit numbers, namely U_1V_1, $(U_1 - U_0)(V_0 - V_1)$, and U_0V_0, plus some simple shifting and adding operations.

Formula (2) can be used for double-precision multiplication when we want a quadruple-precision result, and it will be just a little faster than the traditional method on many machines. But the main advantage of (2) is that we can use it to define a recursive process for multiplication that is significantly faster than the familiar order-n^2 method when n is large: If $T(n)$ is the time required to perform multiplication of n-bit numbers, we have

$$T(2n) \le 3T(n) + cn \tag{3}$$

for some constant c, since the right-hand side of (2) uses just three multiplications plus some additions and shifts. Relation (3) implies by induction that

$$T(2^k) \le c(3^k - 2^k), \qquad k \ge 1, \tag{4}$$

if we choose c to be large enough so that this inequality is valid when $k = 1$; therefore we have

$$\begin{aligned} T(n) \le T(2^{\lceil \lg n \rceil}) &\le c(3^{\lceil \lg n \rceil} - 2^{\lceil \lg n \rceil}) \\ &< 3c \cdot 3^{\lg n} = 3cn^{\lg 3}. \end{aligned} \tag{5}$$

Relation (5) shows that the running time for multiplication can be reduced from order n^2 to order $n^{\lg 3} \approx n^{1.585}$, so the recursive method is much faster than the traditional method when n is large.

(A similar but more complicated method for doing multiplication with running time of order $n^{\lg 3}$ was apparently first suggested by A. Karatsuba in *Doklady Akad. Nauk SSSR* **145** (1962), 293–294 [English translation in *Soviet Physics–Doklady* **7** (1963), 595–596]. Curiously, this idea does not seem to have been discovered before 1962; none of the "calculating prodigies" who have become famous for their ability to multiply large numbers mentally have been reported to use any such method, although formula (2) adapted to decimal notation would seem to lead to a reasonably easy way to multiply eight-digit numbers in one's head.)

The running time can be reduced still further, in the limit as n approaches infinity, if we observe that the method just used is essentially the special case $r = 1$ of a more general method that yields

$$T\big((r+1)n\big) \le (2r+1)T(n) + cn \tag{6}$$

for any fixed r. This more general method can be obtained as follows: Let

$$u = (u_{(r+1)n-1} \ldots u_1 u_0)_2 \qquad \text{and} \qquad v = (v_{(r+1)n-1} \ldots v_1 v_0)_2$$

be broken into $r+1$ pieces,

$$u = U_r 2^{rn} + \cdots + U_1 2^n + U_0, \qquad v = V_r 2^{rn} + \cdots + V_1 2^n + V_0, \tag{7}$$

where each U_j and each V_j is an n-bit number. Consider the polynomials

$$U(x) = U_r x^r + \cdots + U_1 x + U_0, \qquad V(x) = V_r x^r + \cdots + V_1 x + V_0, \tag{8}$$

and let

$$W(x) = U(x)V(x) = W_{2r} x^{2r} + \cdots + W_1 x + W_0. \tag{9}$$

Since $u = U(2^n)$ and $v = V(2^n)$, we have $uv = W(2^n)$, so we can easily compute uv if we know the coefficients of $W(x)$. The problem is to find a good way to compute the coefficients of $W(x)$ by using only $2r+1$ multiplications of n-bit numbers plus some further operations that involve only an execution time proportional to n. This can be done by computing

$$U(0)V(0) = W(0), \quad U(1)V(1) = W(1), \quad \ldots, \quad U(2r)V(2r) = W(2r). \tag{10}$$

The coefficients of a polynomial of degree $2r$ can be written as a linear combination of the values of that polynomial at $2r+1$ distinct points; computing such a linear combination requires an execution time at most proportional to n. (Actually, the products $U(j)V(j)$ are not strictly products of n-bit numbers, but they are products of at most $(n+t)$-bit numbers, where t is a fixed value depending on r. It is easy to design a multiplication routine for $(n+t)$-bit numbers that requires only $T(n)+c_1 n$ operations, where $T(n)$ is the number of operations needed for n-bit multiplications, since two products of t-bit by n-bit numbers can be done in $c_2 n$ operations when t is fixed.) Therefore we obtain a method of multiplication satisfying (6).

Relation (6) implies that $T(n) \le c_3 n^{\log_{r+1}(2r+1)} < c_3 n^{1+\log_{r+1} 2}$, if we argue as in the derivation of (5), so we have now proved the following result:

Theorem A. *Given $\epsilon > 0$, there exists a multiplication algorithm such that the number of elementary operations $T(n)$ needed to multiply two n-bit numbers satisfies*

$$T(n) < c(\epsilon) n^{1+\epsilon}, \tag{11}$$

for some constant $c(\epsilon)$ independent of n. ∎

This theorem is still not the result we are after. It is unsatisfactory for practical purposes in that the method becomes much more complicated as $\epsilon \to 0$ (and therefore as $r \to \infty$), causing $c(\epsilon)$ to grow so rapidly that extremely huge values of n are needed before we have any significant improvement over (5). And it is unsatisfactory for theoretical purposes because it does not make use of the full power of the polynomial method on which it is based. We can obtain a better result if we let r *vary* with n, choosing larger and larger values of r as n increases. This idea is due to A. L. Toom [*Doklady Akad. Nauk SSSR* **150** (1963), 496–498, English translation in *Soviet Mathematics* **3** (1963), 714–716], who used it to show that computer circuitry for multiplication of n-bit numbers can be constructed involving a fairly small number of components as n grows. S. A. Cook [*On the minimum computation time of functions* (Thesis, Harvard

University, 1966), 51–77] later showed how Toom's method can be adapted to fast computer programs.

Before we discuss the Toom–Cook algorithm any further, let us study a small example of the transition from $U(x)$ and $V(x)$ to the coefficients of $W(x)$. This example will not demonstrate the efficiency of the method, since the numbers are too small, but it points out some useful simplifications that we can make in the general case. Suppose that we want to multiply $u = 1234$ times $v = 2341$; in binary notation this is $u = (0100\ 1101\ 0010)_2$ times $v = (1001\ 0010\ 0101)_2$. Let $r = 2$; the polynomials $U(x)$, $V(x)$ in (8) are

$$U(x) = 4x^2 + 13x + 2, \qquad V(x) = 9x^2 + 2x + 5.$$

Hence we find, for $W(x) = U(x)V(x)$,

$$\begin{array}{lllll} U(0) = 2, & U(1) = 19, & U(2) = 44, & U(3) = 77, & U(4) = 118; \\ V(0) = 5, & V(1) = 16, & V(2) = 45, & V(3) = 92, & V(4) = 157; \\ W(0) = 10, & W(1) = 304, & W(2) = 1980, & W(3) = 7084, & W(4) = 18526. \end{array} \tag{12}$$

Our job now is to compute the five coefficients of $W(x)$ from the latter five values.

There is an attractive little algorithm that can be used to compute the coefficients of a polynomial $W(x) = W_{m-1}x^{m-1} + \cdots + W_1x + W_0$ when the values $W(0), W(1), \ldots, W(m-1)$ are given: Let us first write

$$W(x) = \theta_{m-1}\,x^{\underline{m-1}} + \theta_{m-2}\,x^{\underline{m-2}} + \cdots + \theta_1 x^{\underline{1}} + \theta_0, \tag{13}$$

where $x^{\underline{k}} = x(x-1)\ldots(x-k+1)$, and where the coefficients θ_j are unknown. The falling factorial powers have the important property that

$$W(x+1) - W(x) = (m-1)\theta_{m-1}\,x^{\underline{m-2}} + (m-2)\theta_{m-2}\,x^{\underline{m-3}} + \cdots + \theta_1;$$

hence by induction we find that, for all $k \geq 0$,

$$\frac{1}{k!}\left(W(x+k) - \binom{k}{1}W(x+k-1) + \binom{k}{2}W(x+k-2) - \cdots + (-1)^k W(x)\right)$$
$$= \binom{m-1}{k}\theta_{m-1}\,x^{\underline{m-1-k}} + \binom{m-2}{k}\theta_{m-2}\,x^{\underline{m-2-k}} + \cdots + \binom{k}{k}\theta_k. \tag{14}$$

Denoting the left-hand side of (14) by $(1/k!)\,\Delta^k W(x)$, we see that

$$\frac{1}{k!}\Delta^k W(x) = \frac{1}{k}\left(\frac{1}{(k-1)!}\Delta^{k-1}W(x+1) - \frac{1}{(k-1)!}\Delta^{k-1}W(x)\right)$$

and $(1/k!)\,\Delta^k W(0) = \theta_k$. So the coefficients θ_j can be evaluated using a very simple method, illustrated here for the polynomial $W(x)$ in (12):

$$\begin{array}{lllll} 10 & & & & \\ & 294 & & & \\ 304 & & 1382/2 = 691 & & \\ & 1676 & & 1023/3 = 341 & \\ 1980 & & 3428/2 = 1714 & & 144/4 = 36 \\ & 5104 & & 1455/3 = 485 & \\ 7084 & & 6338/2 = 3169 & & \\ & 11442 & & & \\ 18526 & & & & \end{array} \tag{15}$$

The leftmost column of this tableau is a listing of the given values of $W(0)$, $W(1)$, $\ldots$, $W(4)$; the kth succeeding column is obtained by computing the difference between successive values of the preceding column and dividing by k. The coefficients θ_j appear at the top of the columns, so that $\theta_0 = 10$, $\theta_1 = 294$, $\ldots$, $\theta_4 = 36$, and we have

$$\begin{aligned} W(x) &= 36x^{\underline{4}} + 341x^{\underline{3}} + 691x^{\underline{2}} + 294x^{\underline{1}} + 10 \\ &= \big(((36(x-3)+341)(x-2)+691)(x-1)+294\big)x + 10. \end{aligned} \tag{16}$$

In general, we can write

$$W(x) = \big(\ldots((\theta_{m-1}(x-m+2)+\theta_{m-2})(x-m+3)+\theta_{m-3})(x-m+4) + \cdots + \theta_1\big)x + \theta_0,$$

and this formula shows how the coefficients $W_{m-1}, \ldots, W_1, W_0$ can be obtained from the θ's:

36	341			
	$-3\cdot 36$			
36	233	691		
	$-2\cdot 36$	$-2\cdot 233$		
36	161	225	294	
	$-1\cdot 36$	$-1\cdot 161$	$-1\cdot 225$	
36	125	64	69	10

(17)

Here the numbers below the horizontal lines successively show the coefficients of the polynomials

$$\theta_{m-1}, \qquad \theta_{m-1}(x+m+2)+\theta_{m-2}, \qquad \big(\theta_{m-1}(x-m+2)+\theta_{m-2}\big)(x-m+3)+\theta_{m-3}, \qquad \text{etc.}$$

From this tableau we have

$$W(x) = 36x^4 + 125x^3 + 64x^2 + 69x + 10,$$

so the answer to our original problem is $1234 \cdot 2341 = W(16)$, where $W(16)$ is obtained by adding and shifting. A generalization of this method for obtaining coefficients is discussed in Section 4.6.4.

The basic Stirling number identity,

$$x^n = \left\{ {n \atop n} \right\} x^{\underline{n}} + \cdots + \left\{ {n \atop 1} \right\} x^{\underline{1}} + \left\{ {n \atop 0} \right\},$$

Eq. 1.2.6–41, shows that if the coefficients of $W(x)$ are nonnegative, so are the numbers θ_j, and in such a case *all of the intermediate results in the above computation are nonnegative.* This further simplifies the Toom–Cook multiplication algorithm, which we will now consider in detail.

Algorithm C (*High-precision multiplication of binary numbers*). Given a positive integer n and two nonnegative n-bit integers u and v, this algorithm forms their $2n$-bit product, w. Four auxiliary stacks are used to hold the long numbers that are manipulated during the procedure:

Stacks U, V:	Temporary storage of $U(j)$ and $V(j)$ in step C4.
Stack C:	Numbers to be multiplied, and control codes.
Stack W:	Storage of $W(j)$.

These stacks may contain either binary numbers or special control symbols called code-1, code-2, and code-3. The algorithm also constructs an auxiliary table of numbers q_k, r_k; this table is maintained in such a manner that it may be stored as a linear list, where a single pointer that traverses the list (moving back and forth) may be used to access the current table entry of interest.

(Stacks C and W are used to control the recursive mechanism of this multiplication algorithm in a reasonably straightforward manner that is a special case of general procedures discussed in Chapter 8.)

C1. [Compute q, r tables.] Set stacks U, V, C, and W empty. Set

$$k \leftarrow 1, \quad q_0 \leftarrow q_1 \leftarrow 16, \quad r_0 \leftarrow r_1 \leftarrow 4, \quad Q \leftarrow 4, \quad R \leftarrow 2.$$

Now if $q_{k-1} + q_k < n$, set

$$k \leftarrow k+1, \quad Q \leftarrow Q + R, \quad R \leftarrow \lfloor \sqrt{Q} \rfloor, \quad q_k \leftarrow 2^Q, \quad r_k \leftarrow 2^R,$$

and repeat this operation until $q_{k-1} + q_k \geq n$. (*Note:* The calculation of $R \leftarrow \lfloor \sqrt{Q} \rfloor$ does not require a square root to be taken, since we may simply set $R \leftarrow R+1$ if $(R+1)^2 \leq Q$ and leave R unchanged if $(R+1)^2 > Q$; see exercise 2. In this step we build the sequences

k	$=$	0	1	2	3	4	5	6	$\ldots$
q_k	$=$	2^4	2^4	2^6	2^8	2^{10}	2^{13}	2^{16}	$\ldots$
r_k	$=$	2^2	2^2	2^2	2^2	2^3	2^3	2^4	$\ldots$

The multiplication of 70000-bit numbers would cause this step to terminate with $k = 6$, since $70000 < 2^{13} + 2^{16}$.)

C2. [Put u, v on stack.] Put code-1 on stack C, then place u and v onto stack C as numbers of exactly $q_{k-1} + q_k$ bits each.

C3. [Check recursion level.] Decrease k by 1. If $k = 0$, the top of stack C now contains two 32-bit numbers, u and v; remove them, set $w \leftarrow uv$ using a built-in routine for multiplying 32-bit numbers, and go to step C10. If $k > 0$, set $r \leftarrow r_k$, $q \leftarrow q_k$, $p \leftarrow q_{k-1} + q_k$, and go on to step C4.

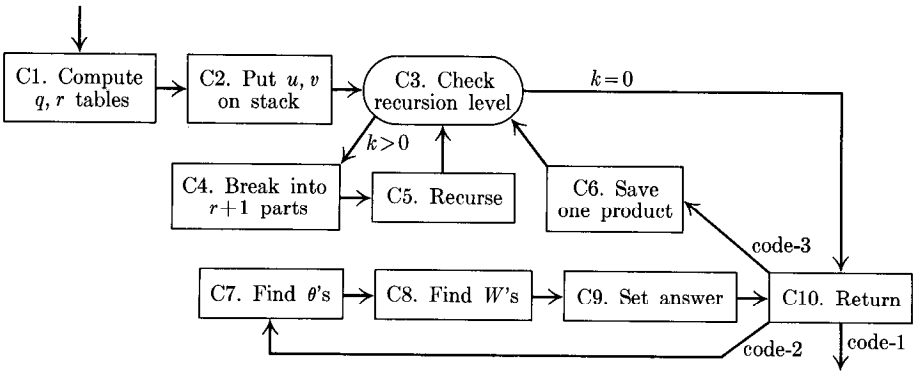


Fig. 8. Toom–Cook algorithm for high-precision multiplication.

C4. [Break into $r+1$ parts.] Let the number at the top of stack C be regarded as a list of $r+1$ numbers with q bits each, $(U_r \ldots U_1 U_0)_{2^q}$. (The top of stack C now contains an $(r+1)q = (q_k + q_{k+1})$-bit number.) For $j = 0$, $1, \ldots, 2r$, compute the p-bit numbers

$$\big(\ldots(U_r j + U_{r-1})j + \cdots + U_1\big)j + U_0 = U(j)$$

and successively put these values onto stack U. (The bottom of stack U now contains $U(0)$, then comes $U(1)$, etc., with $U(2r)$ on top. Note that

$$U(j) \le U(2r) < 2^q\big((2r)^r + (2r)^{r-1} + \cdots + 1\big) < 2^{q+1}(2r)^r \le 2^p,$$

by exercise 3.) Then remove $U_r \ldots U_1 U_0$ from stack C.

Now the top of stack C contains another list of $r+1$ q-bit numbers, $V_r \ldots V_1 V_0$, and the p-bit numbers

$$\big(\ldots(V_r j + V_{r-1})j + \cdots + V_1\big)j + V_0 = V(j)$$

should be put onto stack V in the same way. After this has been done, remove $V_r \ldots V_1 V_0$ from stack C.

C5. [Recurse.] Successively put the following items onto stack C, at the same time emptying stacks U and V:

$$\text{code-2, } V(2r), U(2r), \text{ code-3, } V(2r-1), U(2r-1), \ldots,$$
$$\text{code-3, } V(1), U(1), \text{ code-3, } V(0), U(0).$$

Go back to step C3.

C6. [Save one product.] (At this point the multiplication algorithm has set w to one of the products $W(j) = U(j)V(j)$.) Put w onto stack W. (This number w contains $2(q_k + q_{k-1})$ bits.) Go back to step C3.

C7. [Find θ's.] Set $r \leftarrow r_k$, $q \leftarrow q_k$, $p \leftarrow q_{k-1} + q_k$. (At this point stack W contains a sequence of numbers ending with $W(0)$, $W(1)$, $\ldots$, $W(2r)$ from bottom to top, where each $W(j)$ is a $2p$-bit number.)

Now for $j = 1, 2, 3, \ldots, 2r$, perform the following loop: For $t = 2r$, $2r-1$, $2r-2$, $\ldots$, j, set $W(t) \leftarrow \big(W(t) - W(t-1)\big)/j$. (Here j must increase and t must decrease. The quantity $\big(W(t) - W(t-1)\big)/j$ will always be a nonnegative integer that fits in $2p$ bits; cf. (15).)

C8. [Find W's.] For $j = 2r-1, 2r-2, \ldots, 1$, perform the following loop: For $t = j, j+1, \ldots, 2r-1$, set $W(t) \leftarrow W(t) - jW(t+1)$. (Here j must decrease and t must increase. The result of this operation will again be a nonnegative $2p$-bit integer; cf. (17).)

C9. [Set answer.] Set w to the $2(q_k + q_{k+1})$-bit integer

$$\big(\ldots(W(2r)2^q + W(2r-1))2^q + \cdots + W(1)\big)2^q + W(0).$$

Remove $W(2r)$, $\ldots$, $W(0)$ from stack W.

C10. [Return.] Set $k \leftarrow k+1$. Remove the top of stack C. If it is code-3, go to step C6. If it is code-2, put w onto stack W and go to step C7. And if it is code-1, terminate the algorithm (w is the answer). ∎

Let us now estimate the running time, $T(n)$, for Algorithm C, in terms of some things we shall call "cycles," i.e., elementary machine operations. Step C1 takes $O(q_k)$ cycles, even if we represent the number q_k internally as a long string of q_k bits followed by some delimiter, since $q_k + q_{k-1} + \cdots + q_0$ will be $O(q_k)$. Step C2 obviously takes $O(q_k)$ cycles.

Now let t_k denote the amount of computation required to get from step C3 to step C10 for a particular value of k (after k has been decreased at the beginning of step C3). Step C3 requires $O(q)$ cycles at most. Step C4 involves r multiplications of p-bit numbers by $(\lg 2r)$-bit numbers, and r additions of p-bit numbers, all repeated $4r+2$ times. Thus we need a total of $O(r^2 q \log r)$ cycles. Step C5 requires moving $4r+2$ p-bit numbers, so it involves $O(rq)$ cycles. Step C6 requires $O(q)$ cycles, and it is done $2r+1$ times per iteration. The recursion involved when the algorithm essentially invokes itself (by returning to step C3) requires t_{k-1} cycles, $2r+1$ times. Step C7 requires $O(r^2)$ subtractions of p-bit numbers and divisions of $2p$-bit by $(\lg 2r)$-bit numbers, so it requires $O(r^2 q \log r)$ cycles. Similarly, step C8 requires $O(r^2 q \log r)$ cycles. Step C9 involves $O(rq)$ cycles, and C10 takes hardly any time at all.

Summing up, we have $T(n) = O(q_k) + O(q_k) + t_{k-1}$, where (if $q = q_k$ and $r = r_k$) the main contribution to the running time satisfies

$$\begin{aligned} t_k &= O(q) + O(r^2 q \log r) + O(rq) + (2r+1)O(q) + O(r^2 q \log r) \\ &\qquad + O(r^2 q \log r) + O(rq) + O(q) + (2r+1)t_{k-1} \\ &= O(r^2 q \log r) + (2r+1)t_{k-1}. \end{aligned}$$

Thus there is a constant c such that

$$t_k \le cr_k^2 q_k \lg r_k + (2r_k + 1)t_{k-1}.$$

To complete the estimation of t_k we can prove by brute force that

$$t_k \le Cq_{k+1}2^{2.5\sqrt{\lg q_{k+1}}} \tag{18}$$

for some constant C. Let us choose $C > 20c$, and let us also take C large enough so that (18) is valid for $k \le k_0$, where k_0 will be specified below. Then when $k > k_0$, let $Q_k = \lg q_k$, $R_k = \lg r_k$; we have by induction

$$t_k \le cq_k r_k^2 \lg r_k + (2r_k + 1)Cq_k 2^{2.5\sqrt{Q_k}} = Cq_{k+1}2^{2.5\sqrt{\lg q_{k+1}}}(\eta_1 + \eta_2),$$

where

$$\eta_1 = \frac{c}{C}R_k 2^{R_k - 2.5\sqrt{Q_{k+1}}} < \frac{1}{20}R_k 2^{-R_k} < 0.05,$$

$$\eta_2 = \left(2 + \frac{1}{r_k}\right)2^{2.5(\sqrt{Q_k} - \sqrt{Q_{k+1}})} \to 2^{-1/4} < 0.85,$$

since

$$\sqrt{Q_{k+1}} - \sqrt{Q_k} = \sqrt{Q_k + \lfloor\sqrt{Q_k}\rfloor} - \sqrt{Q_k} \to \tfrac{1}{2}$$

as $k \to \infty$. It follows that we can find k_0 such that $\eta_2 < 0.95$ for all $k > k_0$, and this completes the proof of (18) by induction.

Finally, therefore, we may compute $T(n)$. Since $n > q_{k-1} + q_{k-2}$, we have $q_{k-1} < n$; hence

$$r_{k-1} = 2^{\lfloor\sqrt{\lg q_{k-1}}\rfloor} < 2^{\sqrt{\lg n}}, \qquad \text{and} \qquad q_k = r_{k-1}q_{k-1} < n2^{\sqrt{\lg n}}.$$

Thus

$$t_{k-1} \le Cq_k 2^{2.5\sqrt{\lg q_k}} < Cn2^{\sqrt{\lg n} + 2.5(\sqrt{\lg n} + 1)},$$

and, since $T(n) = O(q_k) + t_{k-1}$, we have finally derived the following theorem:

Theorem C. *There is a constant c_0 such that the execution time of Algorithm C is less than $c_0 n 2^{3.5\sqrt{\lg n}}$ cycles.* ∎

Since $n2^{3.5\sqrt{\lg n}} = n^{1+3.5/\sqrt{\lg n}}$, this result is noticeably stronger than Theorem A. By adding a few complications to the algorithm, pushing the ideas to their apparent limits (see exercise 5), we can improve the estimated execution time to

$$T(n) = O(n2^{\sqrt{2\lg n}}\log n). \tag{19}$$

B. A modular method. There is another way to multiply large numbers very rapidly, based on the ideas of modular arithmetic as presented in Section 4.3.2. It is very hard to believe at first that this method can be of advantage, since a multiplication algorithm based on modular arithmetic must include the choice of moduli and the conversion of numbers into and out of modular representation, besides the actual multiplication operation itself. In spite of these formidable difficulties, A. Schönhage discovered that all of these operations can be carried out quite rapidly.

In order to understand the essential mechanism of Schönhage's method, we shall look at a special case. Consider the sequence defined by the rules

$$q_0 = 1, \qquad q_{k+1} = 3q_k - 1, \tag{20}$$

so that $q_k = 3^k - 3^{k-1} - \cdots - 1 = \frac{1}{2}(3^k + 1)$. We will study a procedure that multiplies $(18q_k + 8)$-bit numbers, in terms of a method for multiplying $(18q_{k-1} + 8)$-bit numbers. Thus, if we know how to multiply numbers having $(18q_0 + 8) = 26$ bits, the procedure to be described will show us how to multiply numbers of $(18q_1 + 8) = 44$ bits, then 98 bits, then 260 bits, etc., eventually increasing the number of bits by almost a factor of 3 at each step.

Let $p_k = 18q_k + 8$. When multiplying p_k-bit numbers, the idea is to use the six moduli

$$\begin{aligned} m_1 &= 2^{6q_k-1} - 1, \quad & m_2 &= 2^{6q_k+1} - 1, \quad & m_3 &= 2^{6q_k+2} - 1, \\ m_4 &= 2^{6q_k+3} - 1, \quad & m_5 &= 2^{6q_k+5} - 1, \quad & m_6 &= 2^{6q_k+7} - 1. \end{aligned} \tag{21}$$

These moduli are relatively prime, by Eq. 4.3.2–18, since the exponents

$$6q_k - 1, \quad 6q_k + 1, \quad 6q_k + 2, \quad 6q_k + 3, \quad 6q_k + 5, \quad 6q_k + 7 \tag{22}$$

are always relatively prime (see exercise 6). The six moduli in (21) are capable of representing numbers up to $m = m_1m_2m_3m_4m_5m_6 > 2^{36q_k+16} = 2^{2p_k}$, so there is no chance of overflow in the multiplication of p_k-bit numbers u and v. Thus we may use the following method, when $k > 0$:

a) Compute $u_1 = u \bmod m_1$, ..., $u_6 = u \bmod m_6$; and $v_1 = v \bmod m_1$, ..., $v_6 = v \bmod m_6$.

b) Multiply u_1 by v_1, u_2 by v_2, ..., u_6 by v_6. These are numbers of at most $6q_k + 7 = 18q_{k-1} + 1 < p_{k-1}$ bits, so the multiplications can be performed by using the assumed p_{k-1}-bit multiplication procedure.

c) Compute $w_1 = u_1v_1 \bmod m_1$, $w_2 = u_2v_2 \bmod m_2$, ..., $w_6 = u_6v_6 \bmod m_6$.

d) Compute w such that $0 \le w < m$, $w \bmod m_1 = w_1$, ..., $w \bmod m_6 = w_6$.

Let t_k be the amount of time needed for this process. It is not hard to see that operation (a) takes $O(p_k)$ cycles, since the determination of $u \bmod (2^e - 1)$ is quite simple (like "casting out nines"), as shown in Section 4.3.2. Similarly, operation (c) takes $O(p_k)$ cycles. Operation (b) requires essentially $6t_{k-1}$ cycles.

This leaves us with operation (d), which seems to be quite a difficult computation; but Schönhage has found an ingenious way to perform step (d) in $O(p_k \log p_k)$ cycles, and this is the crux of the method. As a consequence, we have

$$t_k = 6t_{k-1} + O(p_k \log p_k).$$

Since $p_k = 3^{k+2} + 17$, we can show that the time for n-bit multiplication is

$$T(n) = O(N^{\log_3 6}) = O(N^{1.63}). \tag{23}$$

(See exercise 7.)

Although the modular method is more complicated than the $O(n^{\lg 3})$ procedure discussed at the beginning of this section, Eq. (23) shows that it does, in fact, lead to an execution time substantially better than $O(n^2)$ for the multiplication of n-bit numbers. Thus we can improve on the classical method by using either of two completely different approaches.

Let us now analyze operation (d) above. Assume that we are given a set of positive integers $e_1 < e_2 < \cdots < e_r$, relatively prime in pairs; let

$$m_1 = 2^{e_1} - 1, \qquad m_2 = 2^{e_2} - 1, \qquad \ldots, \qquad m_r = 2^{e_r} - 1. \tag{24}$$

We are also given numbers $w_1, \ldots, w_r$ such that $0 \le w_j \le m_j$. Our job is *to determine the binary representation of the number w that satisfies the conditions*

$$\begin{aligned} 0 \le w < m_1 m_2 \ldots m_r, \\ w \equiv w_1 \text{ (modulo } m_1), \qquad \ldots, \qquad w \equiv w_r \text{ (modulo } m_r). \end{aligned} \tag{25}$$

The method is based on (23) and (24) of Section 4.3.2. First we compute

$$w'_j = \big(\ldots((w_j - w'_1)c_{1j} - w'_2)c_{2j} - \cdots - w'_{j-1}\big)c_{(j-1)j} \bmod m_j, \tag{26}$$

for $j = 2, \ldots, r$, where $w'_1 = w_1 \bmod m_1$; then we compute

$$w = \big(\ldots(w'_r m_{r-1} + w'_{r-1})m_{r-2} + \cdots + w'_2\big)m_1 + w'_1. \tag{27}$$

Here c_{ij} is a number such that $c_{ij}m_i \equiv 1$ (modulo m_j); these numbers c_{ij} are not given, they must be determined from the e_j's.

The calculation of (26) for all j involves $\binom{r}{2}$ additions modulo m_j, each of which takes $O(e_r)$ cycles, plus $\binom{r}{2}$ multiplications by c_{ij}, modulo m_j. The calculation of w by formula (27) involves r additions and r multiplications by m_j; it is easy to multiply by m_j, since this is just adding, shifting, and subtracting, so it is clear that the evaluation of Eq. (27) takes $O(r^2 e_r)$ cycles. We will soon see that each of the multiplications by c_{ij}, modulo m_j, requires only $O(e_r \log e_r)$ cycles, and therefore *it is possible to complete the entire job of conversion in* $O(r^2 e_r \log e_r)$ *cycles.*

The above observations leave us with the following problem to solve: Given positive integers $e < f$ and a nonnegative integer $u < 2^f$, compute the value of $(cu) \bmod (2^f - 1)$, where c is the number such that $(2^e - 1)c \equiv 1$ (modulo $2^f - 1$); and the computation must be done in $O(f \log f)$ cycles. The result of exercise 4.3.2–6 gives a formula for c that suggests a suitable procedure. First we find the least positive integer b such that

$$be \equiv 1 \text{ (modulo } f). \tag{28}$$

This can be done using Euclid's algorithm in $O((\log f)^3)$ cycles, since Euclid's algorithm applied to e and f requires $O(\log f)$ iterations, and each iteration requires $O((\log f)^2)$ cycles; alternatively, we could be very sloppy here without violating the total time constraint, by simply trying $b = 1$, 2, etc., until (28) is satisfied, since such a process would take $O(f \log f)$ cycles in all. Once b has been found, exercise 4.3.2–6 tells us that

$$c = c[b] = \Bigg(\sum_{0 \le j < b} 2^{je} \Bigg) \bmod (2^f - 1). \tag{29}$$

A brute-force multiplication of $(cu) \bmod (2^f - 1)$ would not be good enough to solve the problem, since we do not know how to multiply general f-bit numbers in $O(f \log f)$ cycles. But the special form of c provides a clue: The binary representation of c is composed of bits in a regular pattern, and Eq. (29) shows that the number $c[2b]$ can be obtained in a simple way from $c[b]$. This suggests that we can rapidly multiply a number u by $c[b]$ if we build $c[b]u$ up in $\lg b$ steps in a suitably clever manner, such as the following: Let the binary notation for b be

$$b = (b_s \ldots b_2 b_1 b_0)_2;$$

we may calculate the sequences a_k, d_k, u_k, v_k defined by the rules

$$\begin{aligned} a_0 &= e, & a_k &= 2a_{k-1} \bmod f; \\ d_0 &= b_0 e, & d_k &= (d_{k-1} + b_k a_k) \bmod f; \\ u_0 &= u, & u_k &= (u_{k-1} + 2^{a_{k-1}} u_{k-1}) \bmod (2^f - 1); \\ v_0 &= b_0 u, & v_k &= (v_{k-1} + b_k 2^{d_{k-1}} u_k) \bmod (2^f - 1). \end{aligned} \tag{30}$$

It is easy to prove by induction on k that

$$\begin{aligned} a_k &= (2^k e) \bmod f; & u_k &= (c[2^k]u) \bmod (2^f - 1); \\ d_k &= \big((b_k \ldots b_1 b_0)_2\, e\big) \bmod f; & v_k &= \big(c[(b_k \ldots b_1 b_0)_2]u\big) \bmod (2^f - 1). \end{aligned} \tag{31}$$

Hence the desired result, $(c[b]u) \bmod (2^f - 1)$, is v_s. The calculation of a_k, d_k, u_k, v_k from a_{k-1}, d_{k-1}, u_{k-1}, v_{k-1} takes $O(\log f) + O(\log f) + O(f) + O(f) = O(f)$ cycles, and therefore the entire calculation can be done in $s\,O(f) = O(f \log f)$ cycles as desired.

The reader will find it instructive to study the ingenious method represented by (30) and (31) very carefully. Similar techniques are discussed in Section 4.6.3.

Schönhage's paper [*Computing* **1** (1966), 182–196] shows that these ideas can be extended to the multiplication of n-bit numbers using $r \approx 2^{\sqrt{2 \lg n}}$ moduli, obtaining a method analogous to Algorithm C. We shall not dwell on the details here, since Algorithm C is always superior; in fact, an even better method is next on our agenda.

C. Use of discrete Fourier transforms. The critical problem in high-precision multiplication is the determination of "convolution products" such as

$$u_r v_0 + u_{r-1} v_1 + \cdots + u_0 v_r,$$

and there is an intimate relation between convolutions and an important mathematical concept called "Fourier transformation." If $\omega = \exp(2\pi i/K)$ is a Kth root of unity, the one-dimensional Fourier transform of the sequence of complex numbers $(u_0, u_1, \ldots, u_{K-1})$ is defined to be the sequence $(\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{K-1})$, where

$$\hat{u}_s = \sum_{0 \le t < K} \omega^{st} u_t, \qquad 0 \le s < K. \tag{32}$$

Letting $(\hat{v}_0, \hat{v}_1, \ldots, \hat{v}_{K-1})$ be defined in the same way, as the Fourier transform of $(v_0, v_1, \ldots, v_{K-1})$, it is not difficult to see that $(\hat{u}_0 \hat{v}_0, \hat{u}_1 \hat{v}_1, \ldots, \hat{u}_{K-1} \hat{v}_{K-1})$ is the transform of $(w_0, w_1, \ldots, w_{K-1})$, where

$$\begin{aligned} w_r &= u_r v_0 + u_{r-1} v_1 + \cdots + u_0 v_r + u_{K-1} v_{r+1} + \cdots + u_{r+1} v_{K-1} \\ &= \sum_{i+j \equiv r \,(\text{modulo } K)} u_i v_j. \end{aligned}$$

When $K \ge 2n - 1$ and $u_n = u_{n+1} = \cdots = u_{K-1} = v_n = v_{n+1} = \cdots = v_{K-1} = 0$, the w's are just what we need for multiplication, since the terms $u_{K-1} v_{r+1} + \cdots + u_{r+1} v_{K-1}$ vanish when $0 \le r \le 2n-2$. In other words, *the transform of a convolution product is the ordinary product of the transforms.* This idea is actually a special case of Toom's use of polynomials (cf. (10)), with x replaced by roots of unity.

If K is a power of 2, the discrete Fourier transform (32) can be obtained quite rapidly when the computations are arranged in a certain way, and so can the inverse transform (determining the w's from the $\hat{w}$'s). This property of Fourier transforms was exploited by V. Strassen in 1968, who discovered how to multiply large numbers faster than was possible under all previously known schemes. He and A. Schönhage later refined the method and published improved procedures in *Computing* **7** (1971), 281–292. In order to understand their approach to the problem, let us first take a look at the mechanism of fast Fourier transforms.

Given a sequence of $K = 2^k$ complex numbers $(u_0, \ldots, u_{K-1})$, and given the complex number

$$\omega = \exp(2\pi i/K),$$

the sequence $(\hat{u}_0, \ldots, \hat{u}_{K-1})$ defined in (32) can be calculated rapidly by carrying out the following scheme. (In these formulas the parameters s_j and t_j are either 0 or 1, so that each "pass" represents 2^k computations.)

Pass 0. Let $A^{[0]}(t_{k-1},\ldots,t_0) = u_t$, where $t = (t_{k-1}\ldots t_0)_2$.

Pass 1. Set $A^{[1]}(s_{k-1},t_{k-2},\ldots,t_0) \leftarrow$
$$A^{[0]}(0,t_{k-2},\ldots,t_0) + \omega^{(s_{k-1}0\ldots0)_2}\cdot A^{[0]}(1,t_{k-2},\ldots,t_0).$$

Pass 2. Set $A^{[2]}(s_{k-1},s_{k-2},t_{k-3},\ldots,t_0) \leftarrow$
$$A^{[1]}(s_{k-1},0,t_{k-3},\ldots,t_0) + \omega^{(s_{k-2}s_{k-1}0\ldots0)_2}\cdot A^{[1]}(s_{k-1},1,t_{k-3},\ldots,t_0).$$

...

Pass k. Set $A^{[k]}(s_{k-1},\ldots,s_1,s_0) \leftarrow$
$$A^{[k-1]}(s_{k-1},\ldots,s_1,0) + \omega^{(s_0s_1\ldots s_{k-1})_2}\cdot A^{[k-1]}(s_{k-1},\ldots,s_1,1).$$

It is fairly easy to prove by induction that we have

$$A^{[j]}(s_{k-1},\ldots,s_{k-j},t_{k-j-1},\ldots,t_0) = \sum_{0\le t_{k-1},\ldots,t_{k-j}\le 1} \omega^{(s_0s_1\ldots s_{k-1})_2\cdot(t_{k-1}\ldots t_{k-j}0\ldots0)_2}\,u_t, \tag{33}$$

so that

$$A^{[k]}(s_{k-1},\ldots,s_1,s_0) = \hat{u}_s, \qquad \text{where } s = (s_0s_1\ldots s_{k-1})_2. \tag{34}$$

(Note the reversed order of the binary digits in s. For further discussion of transforms such as this, see Section 4.6.4.)

To get the inverse Fourier transform $(u_0,\ldots,u_{K-1})$ from the values of $(\hat{u}_0,\ldots,\hat{u}_{K-1})$, we may note that the "double transform" is

$$\hat{\hat{u}}_r = \sum_{0\le s<K}\omega^{rs}\hat{u}_s = \sum_{0\le s,t<K}\omega^{rs}\omega^{st}u_t = \sum_{0\le t<K}u_t\left(\sum_{0\le s<K}\omega^{s(t+r)}\right) = Ku_{(-r)\bmod K},$$

since the geometric series $\sum_{0\le s<K}\omega^{sj}$ sums to zero unless j is a multiple of K. Therefore the inverse transform can be computed in the same way as the transform itself, except that the final results must be divided by K and shuffled slightly.

Applying this to the problem of integer multiplication, suppose we wish to compute the product of two n-bit integers u and v. As in Algorithm C we shall work with groups of bits; let

$$2n \le 2^k l < 4n, \qquad K = 2^k, \qquad L = 2^l, \tag{35}$$

and write

$$u = (U_{K-1}\ldots U_1U_0)_L, \qquad v = (V_{K-1}\ldots V_1V_0)_L, \tag{36}$$

regarding u and v as K-place numbers in radix L so that each digit U_j or V_j is an l-bit integer. Actually the leading digits U_j and V_j are zero for all $j \ge K/2$,

because $2^{k-1}l \geq n$. We will select appropriate values for k and l later; at the moment our goal is to see what happens in general, so that we can choose k and l intelligently when all the facts are before us.

The next step of the multiplication procedure is to compute the Fourier transforms $(\hat{u}_0, \ldots, \hat{u}_{K-1})$ and $(\hat{v}_0, \ldots, \hat{v}_{K-1})$ of the sequences $(u_0, \ldots, u_{K-1})$ and $(v_0, \ldots, v_{K-1})$, where we define

$$u_t = U_t/2^{k+l}, \qquad v_t = V_t/2^{k+l}. \tag{37}$$

This scaling is done for convenience so that the absolute values $|u_t|$ and $|v_t|$ are less than 2^{-k}, ensuring that $|\hat{u}_s|$ and $|\hat{v}_s|$ will be less than 1 for all s.

An obvious problem arises here, since the complex number ω can't be represented exactly in binary notation. How are we going to compute a reliable Fourier transform? By a stroke of good luck, it turns out that everything will work properly if we do the calculations with only a modest amount of precision. For the moment let us bypass this question and assume that infinite-precision calculations are being performed; we shall analyze later how much accuracy is actually needed.

Once the $\hat{u}_s$ and $\hat{v}_s$ have been found, we let $\hat{w}_s = \hat{u}_s\hat{v}_s$ for $0 \leq s < K$ and determine the inverse Fourier transform $(w_0, \ldots, w_{K-1})$. As explained above, we now have

$$w_r = \sum_{i+j=r} u_i v_j = \sum_{i+j=r} U_i V_j / 2^{2k+2l},$$

so the integers $W_r = 2^{2k+2l} w_r$ are the coefficients in the desired product

$$u \cdot v = W_{K-2} L^{K-2} + \cdots + W_1 L + W_0. \tag{38}$$

Since $0 \leq W_r < (r+1)L^2 < KL^2$, each W_r has at most $k + 2l$ bits, so it will not be difficult to compute the binary representation when the W's are known unless k is large compared to l.

Let us try to estimate how much time this method takes, if m-bit fixed point arithmetic is used in calculating the Fourier transforms. Exercise 10 shows that all of the quantities $A^{[j]}$ during all the passes of the transform calculations will be less than 1 in magnitude because of the scaling (37), hence it suffices to deal with m-bit fractions $(.a_{-1} \ldots a_{-m})_2$ for the real and imaginary parts of all the intermediate quantities. Simplifications are possible because the inputs u_t and v_t are real-valued; only K real values instead of $2K$ need to be carried in each step (see exercise 4.6.4–14). We will ignore such refinements in order to keep complications to a minimum.

The first job is to compute ω and its powers. For simplicity we shall make a table of the values $\omega^0, \ldots, \omega^{K-1}$. Let

$$\omega_r = \exp(2\pi i/2^r),$$

so that $\omega_1 = -1$, $\omega_2 = i$, $\omega_3 = (1+i)/\sqrt{2}$, $\ldots$, $\omega_k = \omega$. If $\omega_r = x_r + iy_r$, we have

$$\omega_{r+1} = \sqrt{\frac{1+x_r}{2}} + i\sqrt{\frac{1-x_r}{2}}. \tag{39}$$

The calculation of $\omega_1, \omega_2, \ldots, \omega_k$ takes negligible time compared with the other computations we need, so we can use any straightforward algorithm for square roots. Once the ω_r have been calculated we can compute all of the powers ω^j by starting with $\omega^0 = 1$ and using the following idea for $j > 0$: If $j = 2^{K-r} \cdot q$ where q is odd, and if $j_0 = 2^{K-r} \cdot (q-1)$, we have

$$\omega^j = \omega^{j_0} \cdot \omega_r. \tag{40}$$

This method of calculation keeps errors from propagating, since each ω^j is a product of at most k of the ω_r's. The total time to calculate all the ω^j is $O(KM)$, where M is the time to do an m-bit complex multiplication; this is less time than the subsequent steps will require, so we can ignore it.

Each of the three Fourier transformations comprises k passes, each of which involves K operations of the form $a \leftarrow b + \omega^j c$, so the total time to calculate the Fourier transforms is

$$O(kKM) = O(Mnk/l).$$

Finally, the work involved in computing the binary digits of $u \cdot v$ using (38) is $O(K(k+l)) = O(n + nk/l)$. Summing over all operations, we find that the total time to multiply n-bit numbers u and v will be $O(n) + O(Mnk/l)$.

Now let's see how large the intermediate precision m needs to be, so that we know how large M needs to be. For simplicity we shall be content with safe estimates of the accuracy, instead of finding the best possible bounds. It will be convenient to compute all the ω^j so that our approximations $(\omega^j)'$ will satisfy $|(\omega^j)'| \le 1$; this condition is easy to guarantee if we truncate towards zero instead of rounding. The operations we need to perform with m-bit fixed point complex arithmetic are all obtained by replacing an exact computation of the form $a \leftarrow b + \omega^j c$ by the approximate computation

$$a' \leftarrow \text{truncate}\big(b' + (\omega^j)' c'\big), \tag{41}$$

where b', $(\omega^j)'$, and c' are previously computed approximations; all of these complex numbers and their approximations are bounded by 1 in absolute value. If $|b' - b| \le \delta_1$, $|(\omega^j)' - \omega^j| \le \delta_2$, and $|c' - c| \le \delta_3$, it is not difficult to see that we will have $|a' - a| < \delta + \delta_1 + \delta_2 + \delta_3$, where

$$\delta = |2^{-m} + 2^{-m} i| = 2^{1/2-m},$$

because we have $\big|(\omega^j)'c' - \omega^j c\big| = \big|\big((\omega^j)' - \omega^j\big)c' + \omega^j(c' - c)\big| \le \delta_2 + \delta_3$, and δ is the maximum truncation error. The approximations $(\omega^j)'$ are obtained by starting with approximate values ω_r' to the numbers defined in (39), and we may assume that $|\omega_r' - \omega_r| \le \delta$. Each multiplication (40) has the form of (41) with $b' = 0$, so an additional error of at most 2δ is made per multiplication, and we have $\big|(\omega^j)' - \omega^j\big| \le (2k-1)\delta$ for all j.

If we have errors of at most ϵ before any pass of the fast Fourier transform, the operations of that pass therefore have the form (41) where $\delta_1 = \delta_3 = \epsilon$ and $\delta_2 = (2k-1)\delta$, and the errors after the pass will be at most $2\epsilon + 2k\delta$. There is no error in "Pass 0," so we find by induction on j that the maximum error after "Pass j" is bounded by $(2^j - 1)\cdot 2k\delta$, and the computed values of $\hat{u}_s$ will satisfy $|(\hat{u}_s)' - \hat{u}_s| < (2^k - 1)\cdot 2k\delta$. A similar formula will hold for $(\hat{v}_s)'$; and we will have

$$|(\hat{w}_s)' - \hat{w}_s| < 2(2^k - 1)\cdot 2k\delta + \delta.$$

During the inverse transformation there is an additional accumulation of errors, but the division by $K = 2^k$ ameliorates most of this; by the same argument we find that the computed values w_r' will satisfy

$$|w_r' - w_r| < 4k2^k\delta.$$

We need enough precision to make $2^{2k+2l}w_r'$ round to the correct integer W_r, hence we need

$$2^{2k+2l+2+\lg k+k+1/2-m} \le \frac{1}{2},$$

i.e., $m \ge 3k + 2l + \lg k + 7/2$. This will hold if we simply require that

$$k \ge 7 \qquad \text{and} \qquad m \ge 4k + 2l. \tag{42}$$

Relations (35) and (42) can be used to determine parameters k, l, m so that multiplication takes $O(n) + O(Mnk/l)$ units of time, where M is the time to multiply m-bit fractions.

If we are using MIX, for example, suppose we want to multiply binary numbers having $n = 2^{13} = 8192$ bits each. We can choose $k = 11$, $l = 8$, $m = 60$, so that the necessary m-bit operations are nothing more than double precision arithmetic. The running time M needed to do fixed point m-bit complex multiplication will therefore be comparatively small. With triple-precision operations we can go up for example to $k = l = 15$, $n \le 15\cdot 2^{14}$, which takes us way beyond MIX's memory capacity.

Further study of the choice of k, l, and m leads in fact to a rather surprising conclusion: ***For all practical purposes we can assume that M is constant, and the Schönhage–Strassen multiplication technique will have a running time linearly proportional to n.*** The reason is that we can choose $k = l$ and $m = 6k$; this choice of k is always less than $\lg n$, so we will never need to use more than sextuple precision unless n is larger than the word size of our computer. (In particular, n would have to be larger than the capacity of an index register, so we probably couldn't fit the numbers u and v in main memory.)

The practical problem of fast multiplication is therefore solved, except for improvements in the constant factor. In fact, the all-integer convolution algorithm of exercise 4.6.4–59 is probably a better choice for practical high-precision multiplication, even though it has a slightly worse asymptotic behavior. Our interest in multiplying large numbers is partly theoretical, however, because it

is interesting to explore the ultimate limits of computational complexity. So let's forget practical considerations and suppose that n is extremely huge, perhaps much larger than the number of atoms in the universe. We can let m be approximately $6 \lg n$, and use the same algorithm recursively to do the m-bit multiplications. The running time will satisfy $T(n) = O\big(nT(\log n)\big)$; hence

$$T(n) \le C\, n(C \lg n)(C \lg\lg n)(C \lg\lg\lg n) \ldots,$$

where the product continues until reaching a factor with $\lg \ldots \lg n \le 1$.

Schönhage and Strassen showed how to improve this theoretical upper bound to $O(n \log n \log\log n)$ in their paper, by using *integer* numbers ω to carry out fast Fourier transforms on integers, modulo numbers of the form $2^e + 1$. This upper bound applies to Turing machines, i.e., to computers with bounded memory and a finite number of arbitrarily long tapes.

If we allow ourselves a more powerful computer, with random access to any number of words of bounded size, Schönhage has pointed out that the upper bound drops to $O(n \log n)$. For we can choose $k = l$ and $m = 6k$, and we have time to build a complete multiplication table of all possible products xy for $0 \le x, y < 2^{\lceil m/12 \rceil}$. (The number of such products is 2^k or 2^{k+1}, and we can compute each table entry by addition from one of its predecessors in $O(k)$ steps, hence $O(k2^k) = O(n)$ steps will suffice for the calculation.) In this case M is the time needed to do 12-place arithmetic in radix $2^{\lceil m/12 \rceil}$, and it follows that $M = O(k) = O(\log n)$ because 1-place multiplication can be done by table lookup.

Schönhage discovered in 1979 that a *pointer machine* can carry out n-bit multiplication in $O(n)$ steps; see exercise 12. Such devices (which are also called "storage modification machines" and "linking automata") seem to provide the best models of computation when $n \to \infty$, as discussed at the end of Section 2.6. So we can conclude that multiplication in $O(n)$ steps is possible for theoretical purposes as well as in practice.

D. Division. Now that we have efficient routines for multiplication, let's consider the inverse problem. It turns out that division can be performed just as fast as multiplication, except for a constant factor.

To divide an n-bit number u by an n-bit number v, we may first find an n-bit approximation to $1/v$, then multiply by u to get an approximation $\hat{q}$ to u/v; finally, we can make the slight correction necessary to $\hat{q}$ to ensure that $0 \le u - qv < v$ by using another multiplication. From this reasoning, we see that it suffices to have an efficient algorithm for approximating the reciprocal of an n-bit number. The following algorithm does this, using "Newton's method" as explained at the end of Section 4.3.1.

Algorithm R (*High-precision reciprocal*). Let v have the binary representation $v = (0.v_1v_2v_3\ldots)_2$, where $v_1 = 1$. This algorithm computes an approximation z to $1/v$, such that

$$|z - 1/v| \le 2^{-n}. \tag{43}$$

R1. [Initial approximation.] Set $z \leftarrow \frac{1}{4}\lfloor 32/(4v_1 + 2v_2 + v_3)\rfloor$ and $k \leftarrow 0$.

R2. [Newtonian iteration.] (At this point we have a number z of the binary form $(xx.xx\ldots x)_2$ with $2^k + 1$ places after the radix point, and $z \le 2$.) Calculate $z^2 = (xxx.xx\ldots x)_2$ exactly, using a high-speed multiplication routine. Then calculate $V_k z^2$ exactly, where $V_k = (0.v_1 v_2 \ldots v_{2^{k+1}+3})_2$. Then set $z \leftarrow 2z - V_k z^2 + r$, where $0 \le r < 2^{-2^{k+1}-1}$ is added if necessary to "round up" z so that it is a multiple of $2^{-2^{k+1}-1}$. Finally, set $k \leftarrow k+1$.

R3. [Test for end.] If $2^k < n$, go back to step R2; otherwise the algorithm terminates. ∎

This algorithm is based on a suggestion by S. A. Cook. A similar technique has been used in computer hardware [see Anderson, Earle, Goldschmidt, and Powers, *IBM J. Res. Dev.* **11** (1967), 48–52]. Of course, it is necessary to check the accuracy of Algorithm R quite carefully, because it comes very close to being inaccurate. We will prove by induction that

$$z \le 2 \qquad \text{and} \qquad |z - 1/v| \le 2^{-2^k} \tag{44}$$

at the beginning and end of step R2.

For this purpose, let $\delta_k = 1/v - z_k$, where z_k is the value of z after k iterations of step R2. To start the induction on k, we have

$$\delta_0 = 1/v - 8/v' + (32/v' - \lfloor 32/v'\rfloor)/4 = \eta_1 + \eta_2,$$

where $v' = (v_1 v_2 v_3)_2$ and $\eta_1 = (v' - 8v)/vv'$, so that we have $-\frac{1}{2} < \eta_1 \le 0$ and $0 \le \eta_2 < \frac{1}{4}$. Hence $|\delta_0| < \frac{1}{2}$. Now suppose that (44) has been verified for k; then

$$\begin{aligned}\delta_{k+1} = 1/v - z_{k+1} &= 1/v - z_k - z_k(1 - z_k V_k) - r \\ &= \delta_k - z_k(1 - z_k v) - z_k^2(v - V_k) - r \\ &= \delta_k - (1/v - \delta_k)v\delta_k - z_k^2(v - V_k) - r \\ &= v\delta_k^2 - z_k^2(v - V_k) - r.\end{aligned}$$

Now

$$0 \le v\delta_k^2 < \delta_k^2 \le (2^{-2^k})^2 = 2^{-2^{k+1}},$$

and

$$0 \le z^2(v - V_k) + r < 4(2^{-2^{k+1}-3}) + 2^{-2^{k+1}-1} = 2^{-2^{k+1}},$$

so $|\delta_{k+1}| \le 2^{-2^{k+1}}$. We must still verify the first inequality of (44); to show that $z_{k+1} \le 2$, there are three cases: (a) $V_k = \frac{1}{2}$; then $z_{k+1} = 2$. (b) $V_k \ne \frac{1}{2} = V_{k-1}$; then $z_k = 2$, so $2z_k - z_k^2 V_k \le 2 - 2^{-2^{k+1}-1}$. (c) $V_{k-1} \ne \frac{1}{2}$; then $z_{k+1} = 1/v - \delta_{k+1} < 2 - 2^{-2^{k+1}} \le 2$, since $k > 0$.

The running time of Algorithm R is bounded by

$$2T(4n) + 2T(2n) + 2T(n) + 2T(\tfrac{1}{2}n) + \cdots + O(n)$$

steps, where $T(n)$ is an upper bound on the time needed to do a multiplication of n-bit numbers. If $T(n)$ has the form $nf(n)$ for some monotonically nondecreasing function $f(n)$, we have

$$T(4n) + T(2n) + T(n) + \cdots < T(8n),$$

so division can be done with a speed comparable to that of multiplication except for a constant factor.

R. P. Brent has shown that functions such as $\log x$, $\exp x$, and $\arctan x$ can be evaluated to n significant bits in $O(M(n)\log n)$ steps, if it takes $M(n)$ units of time to multiply n-bit numbers [*JACM* **23** (1976), 242–251].

E. An even faster multiplication method. It is natural to wonder if multiplication of n-bit numbers can be accomplished in just n steps. We have come from order n^2 down to order n, so perhaps we can squeeze the time down to the absolute minimum. In fact, it is actually possible to output the answer as fast as we input the digits, if we leave the domain of conventional computer programming and allow ourselves to build a computer that has an unlimited number of components all acting at once.

A *linear iterative array* of automata is a set of devices M_1, M_2, M_3, ... that can each be in a finite set of "states" at each step of a computation. The machines M_2, M_3, ... all have *identical* circuitry, and their state at time $t+1$ is a function of their own state at time t as well as the states of their left and right neighbors at time t. The first machine M_1 is slightly different: its state at time $t+1$ is a function of its own state and that of M_2, at time t, and also of the *input* at time t. The *output* of a linear iterative array is a function defined on the states of M_1.

Let $u = (u_{n-1}\ldots u_1u_0)_2$, $v = (v_{n-1}\ldots v_1v_0)_2$, and $q = (q_{n-1}\ldots q_1q_0)_2$ be binary numbers, and let $uv+q = w = (w_{2n-1}\ldots w_1w_0)_2$. It is a remarkable fact that a linear iterative array can be constructed, independent of n, that will output w_0, w_1, w_2, ... at times 1, 2, 3, ..., if it is given the inputs (u_0, v_0, q_0), (u_1, v_1, q_1), (u_2, v_2, q_2), ... at times 0, 1, 2, ...

We can state this phenomenon in the language of computer hardware, by saying that it is possible to design a single "integrated circuit module" with the following property: If we wire together sufficiently many of these devices in a straight line, with each module communicating only with its left and right neighbors, the resulting circuitry will produce the $2n$-bit product of n-bit numbers in exactly $2n$ clock pulses.

Here is the basic idea behind this construction: At time 0, machine M_1 senses (u_0, v_0, q_0) and it therefore is able to output $(u_0v_0 + q_0) \bmod 2$ at time 1. Then it sees (u_1, v_1, q_1) and it can output $(u_0v_1 + u_1v_0 + q_1 + k_1) \bmod 2$, where k_1 is the "carry" left over from the previous step, at time 2. Next it sees (u_2, v_2, q_2) and outputs $(u_0v_2 + u_1v_1 + u_2v_0 + q_2 + k_2) \bmod 2$; furthermore, its state holds the values of u_2 and v_2 so that machine M_2 will be able to sense these values at time 3, and M_2 will be able to compute u_2v_2 for the benefit of M_1 at time 4. Machine M_1 essentially arranges to start M_2 multiplying the sequence (u_2, v_2),

Table 1

MULTIPLICATION IN A LINEAR ITERATIVE ARRAY

Time	Input		Module M_1					Module M_2					Module M_3				
	u_j / v_j	q_j	c	x_0 / y_0	x_1 / y_1	x / y	z_2 / z_1 / z_0	c	x_0 / y_0	x_1 / y_1	x / y	z_2 / z_1 / z_0	c	x_0 / y_0	x_1 / y_1	x / y	z_2 / z_1 / z_0
0	1 / 1	1	0	0 / 0	0 / 0	0 / 0	0 / 0 / 0	0	0 / 0	0 / 0	0 / 0	0 / 0 / 0	0	0 / 0	0 / 0	0 / 0	0 / 0 / 0
1	1 / 1	1	1	1 / 1	0 / 0	0 / 0	0 / 1 / 0	0	0 / 0	0 / 0	0 / 0	0 / 0 / 0	0	0 / 0	0 / 0	0 / 0	0 / 0 / 0
2	1 / 1	0	2	1 / 1	1 / 1	0 / 0	1 / 0 / 0	0	0 / 0	0 / 0	0 / 0	0 / 0 / 0	0	0 / 0	0 / 0	0 / 0	0 / 0 / 0
3	0 / 0	1	3	1 / 1	1 / 1	1 / 1	0 / 1 / 1	0	0 / 0	0 / 0	0 / 0	0 / 0 / 1	0	0 / 0	0 / 0	0 / 0	0 / 0 / 0
4	1 / 1	0	3	1 / 1	1 / 1	0 / 0	1 / 0 / 1	1	1 / 1	0 / 0	0 / 0	0 / 0 / 1	0	0 / 0	0 / 0	0 / 0	0 / 0 / 0
5	0 / 0	0	3	1 / 1	1 / 1	1 / 1	0 / 1 / 1	2	1 / 1	0 / 0	0 / 0	0 / 0 / 1	0	0 / 0	0 / 0	0 / 0	0 / 0 / 0
6	0 / 0	0	3	1 / 1	1 / 1	0 / 0	1 / 0 / 0	3	1 / 1	0 / 0	1 / 1	0 / 1 / 0	0	0 / 0	0 / 0	0 / 0	0 / 0 / 0
7	0 / 0	0	3	1 / 1	1 / 1	0 / 0	0 / 0 / 0	3	1 / 1	0 / 0	0 / 0	0 / 1 / 0	1	1 / 1	0 / 0	0 / 0	0 / 0 / 1
8	0 / 0	0	3	1 / 1	1 / 1	0 / 0	0 / 0 / 0	3	1 / 1	0 / 0	0 / 0	0 / 1 / 0	2	1 / 1	0 / 0	0 / 0	0 / 0 / 0
9	0 / 0	0	3	1 / 1	1 / 1	0 / 0	0 / 0 / 0	3	1 / 1	0 / 0	0 / 0	0 / 0 / 1	3	1 / 1	0 / 0	0 / 0	0 / 0 / 0
10	0 / 0	0	3	1 / 1	1 / 1	0 / 0	0 / 0 / 1	3	1 / 1	0 / 0	0 / 0	0 / 0 / 0	3	1 / 1	0 / 0	0 / 0	0 / 0 / 0
11	0 / 0	0	3	1 / 1	1 / 1	0 / 0	0 / 0 / 0	3	1 / 1	0 / 0	0 / 0	0 / 0 / 0	3	1 / 1	0 / 0	0 / 0	0 / 0 / 0

(u_3, v_3), ..., and M_2 will ultimately give M_3 the job of multiplying (u_4, v_4), (u_5, v_5), etc. Fortunately, things just work out so that no time is lost. The reader will find it interesting to deduce further details from the formal description that follows.

Each automaton has 2^{11} states

$$(c, x_0, y_0, x_1, y_1, x, y, z_2, z_1, z_0),$$

where $0 \le c < 4$ and each of the x's, y's, and z's is either 0 or 1. Initially, all devices are in state $(0,0,0,0,0,0,0,0,0,0)$. Suppose that a machine M_j, for $j > 1$, is in state $(c, x_0, y_0, x_1, y_1, x, y, z_2, z_1, z_0)$ at time t, and its left neighbor M_{j-1} is in state $(c^l, x_0^l, y_0^l, x_1^l, y_1^l, x^l, y^l, z_2^l, z_1^l, z_0^l)$ while its right neighbor M_{j+1} is in state $(c^r, x_0^r, y_0^r, x_1^r, y_1^r, x^r, y^r, z_2^r, z_1^r, z_0^r)$ at that time. Then machine M_j will go into state $(c', x_0', y_0', x_1', y_1', x', y', z_2', z_1', z_0')$ at time $t+1$, where

$$\begin{aligned}
c' &= \min(c+1, 3) && \text{if } c^l = 3, && 0 && \text{otherwise;}\\
(x_0', y_0') &= (x^l, y^l) && \text{if } c = 0, && (x_0, y_0) && \text{otherwise;}\\
(x_1', y_1') &= (x^l, y^l) && \text{if } c = 1, && (x_1, y_1) && \text{otherwise;}\\
(x', y') &= (x^l, y^l) && \text{if } c \ge 2, && (x, y) && \text{otherwise;}
\end{aligned} \tag{45}$$

and $(z_2' z_1' z_0')_2$ is the binary notation for

$$z_0^r + z_1 + z_2^l + \begin{cases} x^l y^l, & \text{if } c = 0;\\ x_0 y^l + x^l y_0, & \text{if } c = 1;\\ x_0 y^l + x_1 y_1 + x^l y_0, & \text{if } c = 2;\\ x_0 y^l + x_1 y + x y_1 + x^l y_0, & \text{if } c = 3. \end{cases} \tag{46}$$

The leftmost machine M_1 behaves in almost the same way as the others; it acts exactly as if there were a machine to its left in state $(3,0,0,0,0,u,v,q,0,0)$ when it is receiving the inputs (u, v, q). The output of the array is the z_0 component of M_1.

Table 1 shows an example of this array acting on the inputs

$$u = v = (\ldots 00010111)_2, \qquad q = (\ldots 00001011)_2.$$

The output sequence appears in the lower right portion of the states of M_1:

$$0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, \ldots,$$

representing the number $(\ldots 01000011100)_2$ from right to left.

This construction is based on a similar one first published by A. J. Atrubin, *IEEE Trans.* **EC–14** (1965), 394–399.

S. Winograd [*JACM* **14** (1967), 793–802] has investigated the minimum multiplication time achievable in a logical circuit when n is given and when the inputs are available all at once in coded form. See also C. S. Wallace, *IEEE Trans.* **EC–13** (1964), 14–17; A. C. Yao, to appear.

EXERCISES

1. [*22*] The idea expressed in (2) can be generalized to the decimal system, if the radix 2 is replaced by 10. Using this generalization, calculate 2718 times 4742 (reducing this product of four-digit numbers to three products of two-digit numbers, and reducing each of the latter to products of one-digit numbers).

2. [*M22*] Prove that, in step C1 of Algorithm C, the value of R either stays the same or increases by one when we set $R \leftarrow \lfloor\sqrt{Q}\rfloor$. (Therefore, as observed in that step, we need not calculate a square root.)

3. [*M23*] Prove that the sequences q_k, r_k defined in Algorithm C satisfy the inequality $2^{q_k+1}(2r_k)^{r_k} \le 2^{q_{k-1}+q_k}$, when $k > 0$.

▶ **4.** [*28*] (K. Baker.) Show that it is advantageous to evaluate the polynomial $W(x)$ at the points $x = -r, \ldots, 0, \ldots, r$ instead of at the nonnegative points $x = 0, 1, \ldots, 2r$ as in Algorithm C. The polynomial $U(x)$ can be written

$$U(x) = U_e(x^2) + xU_o(x^2),$$

and similarly $V(x)$ and $W(x)$ can be expanded in this way; show how to exploit this idea, obtaining faster calculations in steps C7 and C8.

▶ **5.** [*35*] Show that if in step C1 of Algorithm C we set $R \leftarrow \lceil\sqrt{2Q}\,\rceil + 1$ instead of $R \leftarrow \lfloor\sqrt{Q}\rfloor$, with suitable initial values of q_0, q_1, r_0, and r_1, then (19) can be improved to $t_k \le q_{k+1}2^{\sqrt{2\lg q_{k+1}}}(\lg q_{k+1})$.

6. [*M23*] Prove that the six numbers in (22) are relatively prime in pairs.

7. [*M23*] Prove (23).

▶ **8.** [*25*] Prove that it takes only $O(K \log K)$ arithmetic operations to evaluate the discrete Fourier transform (32), even when K is not a power of 2. [*Hint:* Rewrite (32) in the form

$$\hat{u}_s = \omega^{-s^2/2} \sum_{0 \le t < K} \omega^{(s+t)^2/2}\omega^{-t^2/2}u_t$$

and express this sum as a convolution product.]

9. [*M15*] Suppose the Fourier transformation method of the text is applied with all occurrences of ω replaced by ω^q, where q is some fixed integer. Find a simple relation between the numbers $(\tilde{u}_0, \tilde{u}_1, \ldots, \tilde{u}_{K-1})$ obtained by this general procedure and the numbers $(\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{K-1})$ obtained when $q = 1$.

10. [*M26*] The scaling in (37) makes it clear that all the complex numbers $A^{[j]}$ computed by pass j of the transformation subroutine will be less than 2^{j-k} in absolute value, during the calculations of $\hat{u}_s$ and $\hat{v}_s$ in the Schönhage–Strassen multiplication algorithm. Show that all of the $A^{[j]}$ will be less than 1 in absolute value during the *third* Fourier transformation (the calculation of w_r).

▶ **11.** [*M26*] If n is fixed, how many of the automata in the linear iterative array (45), (46) are needed to compute the product of n-bit numbers? (Note that the automaton M_j is influenced only by the component z_0^r of the machine on its right, so we may remove all automata whose z_0 component is always zero whenever the inputs are n-bit numbers.)

▶ **12.** [*30*] (A. Schönhage.) The purpose of this exercise is to prove that a simple form of pointer machine can multiply n-bit numbers in $O(n)$ steps. The machine has no built-in facilities for arithmetic; all it does is work with nodes and pointers. Each node has the same finite number of link fields, and there are finitely many link registers. The only operations this machine can do are:

i) read one bit of input and jump if that bit is 0;
ii) output 0 or 1;
iii) load a register with the contents of another register or with the contents of a link field in a node pointed to by a register;
iv) store the contents of a register into a link field in a node pointed to by a register;
v) jump if two registers are equal;
vi) create a new node and make a register point to it;
vii) halt.

Implement the Fourier-transform multiplication method efficiently on such a machine. [*Hints:* First show that if N is any positive integer, it is possible to create N nodes representing the integers $\{0, 1, \ldots, N-1\}$, where the node representing p has pointers to the nodes representing $p+1$, $\lfloor p/2 \rfloor$, and $2p$. These nodes can be created in $O(N)$ steps. Show that arithmetic with radix N can now be simulated without difficulty: for example, it takes $O(\log N)$ steps to find the node for $(p+q) \bmod N$ and to determine if $p+q \geq N$, given pointers to p and q; and multiplication can be simulated in $O(\log N)^2$ steps. Now consider the algorithm in the text, with $k = l$ and $m = 6k$ and $N = 2^{\lceil m/13 \rceil}$, so that all quantities in the fixed point arithmetic calculations are 13-place integers with radix N. Finally, show that each pass of the fast Fourier transformations can be done in $O(K + (N \log N)^2) = O(K)$ steps, using the following idea: Each of the K necessary assignments can be "compiled" into a bounded list of instructions for a simulated MIX-like computer whose word size is N, and instructions for K such machines acting in parallel can be simulated in $O(K + (N \log N)^2)$ steps if they are first sorted so that all identical instructions are performed together. (Two instructions are identical if they have the same operation code, the same register contents, and the same memory operand contents.) Note that $N^2 = O(n^{12/13})$, so $(N \log N)^2 = O(K)$.]

13. [*M25*] (A. Schönhage.) What is a good upper bound on the time needed to multiply an m-bit number by an n-bit number, when both m and n are very large but n is much larger than m, based on the results proved in this section for $m = n$?

14. [*M42*] Write a program for Algorithm C, incorporating the improvements of exercise 4. Compare it with a program for Algorithm 4.3.1M and with a program based on (2), to see how large n must be before Algorithm C is an improvement.

15. [*M49*] (S. A. Cook.) A multiplication algorithm is said to be *on line* if the $(k+1)$st input bits of the operands, from right to left, are not read until the kth output bit has been produced. What are the fastest possible on-line multiplication algorithms achievable on various species of automata?

(The best upper bound known is $O(n(\log n)^2 \log\log n)$, due to M. J. Fischer and L. J. Stockmeyer [*J. Comp. and Syst. Sci.* **9** (1974), 317–331]; their construction works on multitape Turing machines, hence also on pointer machines. The best lower bound known is of order $n \log n / \log\log n$, due to M. S. Paterson, M. J. Fischer, and A. R. Meyer [*SIAM/AMS Proceedings* **7** (1974), 97–111]; this applies to multitape Turing machines but not to pointer machines.)

4.4. RADIX CONVERSION

IF OUR ANCESTORS had invented arithmetic by counting with their two fists or their eight fingers, instead of their ten "digits," we would never have to worry about writing binary-decimal conversion routines. (And we would perhaps never have learned as much about number systems.) In this section, we shall discuss the conversion of numbers from positional notation with one radix into positional notation with another radix; this process is, of course, most important on binary computers when converting decimal input data into binary form, and converting binary answers into decimal form.

A. The four basic methods. Binary-decimal conversion is one of the most machine-dependent operations of all, since computer designers keep inventing different ways to provide for it in the hardware. Therefore we shall discuss only the general principles involved, from which a programmer can select the procedure that is best suited to his machine.

We shall assume that only nonnegative numbers enter into the conversion, since the manipulation of signs is easily accounted for.

Let us assume that we are converting from radix b to radix B. (The methods can also be generalized to mixed-radix notations, as shown in exercises 1 and 2.) Most radix-conversion routines are based on multiplication and division, using one of the following four schemes:

1) Conversion of integers (radix point at the right).

• Method (1a) *Division by B* (using radix-b arithmetic). Given an integer number u, we can obtain its radix-B representation $(U_M \dots U_1 U_0)_B$ as follows:

$$\begin{aligned} U_0 &= u \bmod B \\ U_1 &= \lfloor u/B \rfloor \bmod B \\ U_2 &= \lfloor \lfloor u/B \rfloor / B \rfloor \bmod B \\ &\dots \end{aligned}$$

etc., stopping when $\lfloor \dots \lfloor \lfloor u/B \rfloor / B \rfloor \dots / B \rfloor = 0$.

• Method (1b) *Multiplication by b* (using radix-B arithmetic). If u has the radix-b representation $(u_m \dots u_1 u_0)_b$, we can use radix-B arithmetic to evaluate the polynomial $u_m b^m + \cdots + u_1 b + u_0 = u$ in the form

$$\big((\dots(u_m b + u_{m-1})b + \cdots)b + u_1\big)b + u_0.$$

2) Conversion of fractions (radix point at the left). Note that it is often impossible to express a terminating radix-b fraction $(0.u_{-1}u_{-2}\dots u_{-m})_b$ *exactly* as a terminating radix-B fraction $(0.U_{-1}U_{-2}\dots U_{-M})_B$. For example, the fraction $\frac{1}{10}$ has the infinite binary representation $(0.0001100110011\dots)_2$. Therefore methods of rounding the result to M places are sometimes necessary.

• Method (2a) *Multiplication by B* (using radix-b arithmetic). Given a fractional number u, we can obtain the digits of its radix-B representation $(.U_{-1}U_{-2}\ldots)_B$ as follows:

$$U_{-1} = \lfloor uB \rfloor$$
$$U_{-2} = \lfloor \{uB\}B \rfloor$$
$$U_{-3} = \lfloor \{\{uB\}B\}B \rfloor$$
$$\ldots$$

where $\{x\}$ denotes $x \bmod 1 = x - \lfloor x \rfloor$. If it is desired to round the result to M places, the computation can be stopped after U_{-M} has been calculated, and U_{-M} should be increased by unity if $\{\ldots\{\{uB\}B\}\ldots B\}$ is greater than $\frac{1}{2}$. (Note, however, that this may cause carries to propagate, and these carries must be incorporated into the answer using radix-B arithmetic. It would be simpler to add the constant $\frac{1}{2}B^{-M}$ to the original number u before the calculation begins, but this may lead to a terribly incorrect answer when $\frac{1}{2}B^{-M}$ cannot be represented exactly as a radix-b number inside the computer. Note further that it is possible for the answer to round up to $(1.00\ldots0)_B$, if $b^m \geq 2B^M$.)

Exercise 3 shows how to extend this method so that M is *variable*, just large enough to represent the original number to a specified accuracy; in this case the problem of carries does not occur.

• Method (2b) *Division by b* (using radix-B arithmetic). If u has the radix-b representation $(0.u_{-1}u_{-2}\ldots u_{-m})_b$, we can use radix-$B$ arithmetic to evaluate $u_{-1}b^{-1} + u_{-2}b^{-2} + \cdots + u_{-m}b^{-m}$ in the form

$$((\ldots(u_{-m}/b + u_{1-m})/b + \cdots + u_{-2})/b + u_{-1})/b.$$

Care should be taken to control errors that might occur due to truncation or rounding in the division by b; these are often negligible, but not always.

To summarize, Methods (1a), (1b), (2a), and (2b) give us two choices for a conversion process, depending on whether our number is an integer or a fraction. And it is certainly possible to convert between integers and fractions by multiplying or dividing by an appropriate power of b or B; therefore there are at least four methods to choose from when trying to do a conversion.

B. Single-precision conversion. To illustrate these four methods, let us assume that MIX is a binary computer, and suppose that we want to convert a binary integer u to a decimal integer. Thus $b = 2$ and $B = 10$. Method (1a) could be programmed as follows:

```
    ENT1  0            Set j ← 0.
    LDX   U
    ENTA  0            Set rAX ← u.
1H  DIV   =10=         (rA, rX) ← (⌊rAX/10⌋, rAX mod 10).
    STX   ANSWER,1     U_j ← rX.                          (1)
    INC1  1            j ← j + 1.
    SRAX  5            rAX ← rA.
    JXP   1B           Repeat until result is zero.  ∎
```

This requires $18M + 4$ cycles to obtain M digits.

The above method uses division by 10; Method (2a) uses *multiplication* by 10, so it might be a little faster. In order to use Method (2a), we must deal with fractions, and this leads to an interesting situation. Let w be the word size of the computer, and assume that $u < 10^n < w$. With a single division we can find q and r, where

$$wu = 10^n q + r, \qquad 0 \le r < 10^n. \tag{2}$$

Now if we apply Method (2a) to the fraction $(q+1)/w$, we will obtain the digits of u from left to right, in n steps, since

$$\left\lfloor 10^n \frac{q+1}{w} \right\rfloor = \left\lfloor u + \frac{10^n - r}{w} \right\rfloor = u. \tag{3}$$

(This idea is due to P. A. Samet, *Software—Practice and Experience* **1** (1971), 93–96.)

Here is the corresponding MIX program:

```
      JOV   OFLO          Ensure overflow is off.
      LDA   U
      LDX   =10^n=        rAX ← wu + 10^n.
      DIV   =10^n=        rA ← q + 1, rX ← r.
      JOV   ERROR         Jump if u ≥ 10^n.
      ENT1  n-1           Set j ← n − 1.                          (4)
2H    MUL   =10=          Now imagine radix point at left, rA = x.
      STA   ANSWER,1      Set U_j ← ⌊10x⌋.
      SLAX  5             x ← {10x}.
      DEC1  1             j ← j − 1.
      J1NN  2B            Repeat for n > j ≥ 0.  ∎
```

This slightly longer routine requires $16n + 19$ cycles, so it is a little faster than program (1) if $n = M \ge 8$; when leading zeros are present, (1) will be faster.

Program (4) as it stands cannot be used to convert integers $u \ge 10^m$ when $10^m < w < 10^{m+1}$, since we would need to take $n = m + 1$. In this case we can obtain the leading digit of u by computing $\lfloor u/10^m \rfloor$; then $u \bmod 10^m$ can be converted as above with $n = m$.

The fact that the answer digits are obtained from left to right may be an advantage in some applications (e.g., when typing out the answer one digit at a time). Thus we see that a fractional method can be used for conversion of integers, although the use of inexact division makes a little bit of numerical analysis necessary.

A modification of Method (1a) can be used to avoid division by 10, by replacing it with two multiplications. It is worth mentioning this modification here, because radix conversion is often done by small "satellite" computers that have no division capability. If we let x be an approximation to $\frac{1}{10}$, so that $\frac{1}{10} < x < \frac{1}{10} + 1/w$, it is easy to prove (see exercise 7) that $\lfloor ux \rfloor = \lfloor u/10 \rfloor$ or $\lfloor u/10 \rfloor + 1$, so long as $0 \le u < w$. Therefore, if we compute $u - 10\lfloor ux \rfloor$, we

will be able to determine the value of $\lfloor u/10 \rfloor$:

$$\lfloor u/10 \rfloor = \begin{cases} \lfloor ux \rfloor, & \text{if } u - 10\lfloor ux \rfloor \geq 0; \\ \lfloor ux \rfloor - 1, & \text{if } u - 10\lfloor ux \rfloor < 0. \end{cases} \tag{5}$$

This procedure simultaneously determines $u \bmod 10$. A MIX program for conversion using this idea appears in exercise 8; it requires about 33 cycles per digit.

The procedure represented by (5) can be used effectively even if the computer has no built-in multiplication instruction, since multiplication by 10 consists of two shifts and one addition ($10 = 2^3 + 2$). Even the task of multiplication by $\frac{1}{10}$ can be done by judiciously combining a few shifting and adding operations, as explained in exercise 9.

Another way to convert from binary to decimal is to use Method (1b), but to do this we need to simulate doubling in a *decimal* number system. This idea is generally most suitable for incorporation into computer hardware; however, it is possible to program the doubling process for decimal numbers, using binary addition, binary shifting, and binary extraction ("logical AND" on each bit in the register), as shown in the following table.

Table 1

DOUBLING A BINARY-CODED DECIMAL NUMBER

Operation	*General form*	*Example*
1. Given number	$u_1u_2u_3u_4\ u_5u_6u_7u_8\ u_9u_{10}u_{11}u_{12}$	0011 0110 1001 = 3 6 9
2. Add 3 to each digit	$v_1v_2v_3v_4\ v_5v_6v_7v_8\ v_9v_{10}v_{11}v_{12}$	0110 1001 1100
3. Shift left one	$v_1\ v_2v_3v_4v_5\ v_6v_7v_8v_9\ v_{10}v_{11}v_{12}\ 0$	0 1101 0011 1000
4. Extract low bit	$v_1\ 0\ 0\ 0\ v_5\ 0\ 0\ 0\ v_9\ 0\ 0\ 0\ 0$	0 0001 0001 0000
5. Shift right two	$0\ v_1\ 0\ 0\ 0\ v_5\ 0\ 0\ 0\ v_9\ 0\ 0$	0000 0100 0100
6. Shift right one and add	$0\ v_1v_1\ 0\ 0\ v_5v_5\ 0\ 0\ v_9\ v_9\ 0$	0000 0110 0110
7. Add result of step 3	$*\ *\ *\ *\ *\ *\ *\ *\ *\ *\ *\ 0$	0 1101 1001 1110
8. Subtract 6 from each	$y_1\ y_2y_3y_4y_5\ y_6y_7y_8y_9\ y_{10}y_{11}y_{12}\ 0$	0 0111 0011 1000 = 7 3 8

This method changes each individual digit d into $\big((d+3)\times 2+0\big)-6 = 2d$ when $0 \leq d \leq 4$, and into $\big((d+3)\times 2+6\big)-6 = (2d-10)+2^4$ when $5 \leq d \leq q$; and that is just what is needed to double decimal numbers encoded with 4 bits per digit.

Another related idea is to keep a table of the powers of two in decimal form, and to add the appropriate powers together by simulating decimal addition. A survey of bit-manipulation techniques appears in Section 7.1.

Finally, even Method (2b) can be used for the conversion of binary integers to decimal integers. We can find q as in (2), and then we can simulate the decimal division of $q+1$ by w, using a "halving" process (exercise 10) that is similar to the doubling process just described, retaining only the first n digits to the right of the radix point in the answer. In this situation, Method (2b) does not seem to offer advantages over the other three methods already discussed, but we have confirmed the remark made earlier that at least four distinct methods are available for converting integers from one radix to another.

Now let us consider decimal-to-binary conversion (so that $b = 10$, $B = 2$). Method (1a) simulates a decimal division by 2; this is feasible (see exercise 10), but it is primarily suitable for incorporation in hardware instead of programs.

Method (1b) is the most practical method for decimal-to-binary conversion in the great majority of cases. Here it is in MIX code, assuming that there are at least two digits in the number $(u_m \ldots u_1 u_0)_{10}$ being converted:

```
    ENT1 M-1          Set j ← m − 1.
    LDA  INPUT+M      Set U ← u_m.
1H  MUL  =10=
    SLAX 5                                        (6)
    ADD  INPUT,1      U ← 10U + u_j.
    DEC1 1
    J1NN 1B           Repeat for m > j ≥ 0. ∎
```

Note again that adding and shifting may be substituted for the multiplication by 10.

A trickier but perhaps faster method, which uses about $\lg m$ multiplications, extractions, and additions instead of m multiplications and additions, is described in exercise 19.

For the conversion of decimal fractions $(0.u_{-1}u_{-2}\ldots u_{-m})_{10}$ to binary form, we can use Method (2b); or, more commonly, we can convert the integer $(u_{-1}u_{-2}\ldots u_{-m})_{10}$ by Method (1b) and then divide by 10^m.

C. Hand calculation. It is occasionally necessary for computer programmers to convert numbers by hand, and since this is a subject not yet taught in elementary schools, it may be worthwhile to examine it briefly here. There are very simple pencil-and-paper methods for converting between decimal and octal notations, and these methods are easily learned, so they ought to be more widely known.

Converting octal integers to decimal. The simplest conversion is from octal to decimal; this technique was apparently first published by Walter Soden, *Math. Comp.* **7** (1953), 273–274. To do the conversion, write down the given octal number; then at the kth step, double the k leading digits using decimal arithmetic, and subtract this from the $k+1$ leading digits using decimal arithmetic. The process terminates in $n-1$ steps if the given number has n digits. It is a good idea to insert a radix point to show which digits are being doubled, as shown in the following example, in order to prevent embarrassing mistakes.

Example 1. Convert $(5325121)_8$ to decimal.

```
   5.3 2 5 1 2 1
— 1 0
   4 3.2 5 1 2 1
—    8 6
   3 4 6.5 1 2 1
—    6 9 2
   2 7 7 3.1 2 1
—    5 5 4 6
   2 2 1 8 5.2 1
—    4 4 3 7 0
   1 7 7 4 8 2.1
—    3 5 4 9 6 4
   1 4 1 9 8 5 7
```

Answer: $(14198757)_{10}$.

A reasonably good check on the computations may be had by "casting out nines": The sum of the digits of the decimal number must be congruent modulo 9 to the alternating sum and difference of the digits of the octal number, with the rightmost digit of the latter given a plus sign. In the above example, we have $1+4+1+9+8+5+7=35$, and $1-2+1-5+2-3+5=-1$; the difference is 36 (a multiple of 9). If this test fails, it can be applied to the $k+1$ leading digits after the kth step, and the error can be located using a "binary search" procedure; i.e., we start by checking the middle result, then use the same procedure on the first or second half of the calculation, depending on whether the middle result is incorrect or correct.

The "casting-out-nines" process is only about 89 percent reliable, because there is one chance in nine that two *random* integers will differ by a multiple of nine. An even better check is to convert the answer back to octal by using an inverse method, which we shall now consider.

Converting decimal integers to octal. A similar procedure can be used for the opposite conversion: Write down the given decimal number; then at the kth step, double the k leading digits using *octal* arithmetic, and *add* these to the $k+1$ leading digits using *octal* arithmetic. The process terminates in $n-1$ steps if the given number has n digits. (See Example 2 on the following page.)

The two procedures just given are essentially Method (1b) of the general radix-conversion procedures. Doubling and subtracting in decimal notation is like multiplying by $10-2=8$; doubling and adding in octal notation is like multiplying by $8+2=10$. There is a similar method for hexadecimal/decimal conversions, but it is a little more difficult since it involves multiplication by 6 instead of by 2.

To keep these two methods straight in our minds, it is not hard to remember that we must subtract to go from octal to decimal, since the decimal representation of a number is smaller; similarly we must add to go from decimal to

Example 2. Convert $(1419857)_{10}$ to octal.

```
  1.4 1 9 8 5 7
+   2
  1 6.1 9 8 5 7
+   3 4
  2 1 5.9 8 5 7
+   4 3 2
  2 6 1 3.8 5 7
+   5 4 2 6
  3 3 5 6 6.5 7
+   6 7 3 5 4
  4 2 5 2 4 1.7
+ 1 0 5 2 5 0 2
  5 3 2 5 1 2 1
```

(Note that the nonoctal digits 8 and 9 enter into this octal computation.) The answer can be checked as discussed above. This method was published by Charles P. Rozier, *IEEE Trans.* **CE–11** (1962), 708–709.

Answer: $(5325121)_8$.

octal. The computations are performed using the radix of the *answer*, not the radix of the given number, otherwise we couldn't get the desired answer.

Converting fractions. No equally fast method of converting fractions manually is known; the best way seems to be Method (2a), with doubling and adding or subtracting to simplify the multiplications by 10 or by 8. In this case, we reverse the addition-subtraction criterion, adding when we convert to decimal and subtracting when we convert to octal; we also use the radix of the given input number, *not* the radix of the answer, in this computation (see Examples 3 and 4). The process is about twice as hard as the above method for integers.

Example 3. Convert $(.14159)_{10}$ to octal.

```
.1 4 1 5 9
  2 8 3 1 8−
1.1 3 2 7 2
    2 6 5 4 4−
  1.0 6 1 7 6
      1 2 3 5 2−
    0.4 9 4 0 8
        9 8 8 1 6−
      3.9 5 2 4 6
          1 9 0 5 2 8−
        7.6 2 1 1 2
            1 2 4 2 2 4−
          4.9 6 8 9 6
```

Answer: $(.110374\ldots)_8$.

Example 4. Convert $(.110374)_8$ to decimal.

```
.1 1 0 3 7 4
  2 2 0 7 7 0+
1.3 2 4 7 3 0
    6 5 1 6 6 0+
  4.1 2 1 1 6 0
      2 4 2 3 4 0+
    1.4 5 4 1 4 0
        1 1 3 0 3 0 0+
      5.6 7 1 7 0 0
          1 5 6 3 6 0 0+
        8.5 0 2 6 0 0
            1 2 0 5 4 0 0+
          6.2 3 3 4 0 0
```

Answer: $(.141586\ldots)_{10}$.

D. Floating point conversion. When floating point values are to be converted, it is necessary to deal with both the exponent and the fraction parts simultaneously, since conversion of the exponent will affect the fraction part. Given the number $f \cdot 2^e$ to be converted to decimal, we may express 2^e in the form $F \cdot 10^E$ (usually by means of auxiliary tables), and then convert Ff to decimal. Alternatively, we can multiply e by $\log_{10} 2$ and round this to the nearest integer E; then divide $f \cdot 2^e$ by 10^E and convert the result. Conversely, given the number $F \cdot 10^E$ to be converted to binary, we may convert F and then multiply it by the floating point number 10^E (again by using auxiliary tables). Obvious techniques can be used to reduce the maximum size of the auxiliary tables by using several multiplications and/or divisions, although this can cause rounding errors to propagate.

E. Multiple-precision conversion. When converting extremely long numbers, it is most convenient to start by converting blocks of digits, which can be handled by single-precision techniques, and then to combine these blocks by using simple multiple-precision techniques. For example, suppose that 10^n is the highest power of 10 less than the computer word size. Then

a) To convert a multiple-precision *integer* from binary to decimal, divide it repeatedly by 10^n (thus converting from binary to radix 10^n by Method (1a)). Single-precision operations will give the n decimal digits for each place of the radix-10^n representation.

b) To convert a multiple-precision *fraction* from binary to decimal, proceed similarly, multiplying by 10^n (i.e., using Method (2a) with $B = 10^n$).

c) To convert a multiple-precision integer from decimal to binary, convert blocks of n digits first; then use Method (1b) to convert from radix 10^n to binary.

d) To convert a multiple-precision fraction from decimal to binary, convert first to radix 10^n as in (c), then use Method (2b).

F. History and Bibliography. Radix-conversion techniques implicitly originated in ancient problems dealing with weights, measures, and currencies, where mixed-radix systems were generally involved; auxiliary tables were usually prepared to help make the conversions. During the seventeenth century, when sexagesimal fractions were being supplanted by decimal fractions, it was necessary to convert between the two systems in order to use existing books of astronomical tables; a systematic method to transform fractions from radix 60 to radix 10 and vice versa was given in the 1667 edition of William Oughtred's *Clavis Mathematicæ*, Chapter 6, Section 18. (This material was not present in the original 1631 edition of Oughtred's book.) Conversion rules had already been given by al-Kashî of Persia in his *Key to Arithmetic* (c. 1414), where Methods (1a), (1b), and (2a) are clearly explained [*Istoriko-Mat. Issled.* **7** (1954), 126–135], but his work was unknown in Europe. The 18th century American mathematician Hugh Jones used the words "octavation" and "decimation" to describe octal/decimal conversions, but his methods were not as clever as his terminology. A. M. Legendre [*Théorie des nombres* (Paris: 1798), 229] noted that positive integers may be conveniently converted to binary form if they are repeatedly divided by 64.

In 1946, H. H. Goldstine and J. von Neumann gave prominent consideration to radix conversion in their classic memoir, "Planning and coding problems for an electronic computing instrument," because it was necessary to justify the use of binary arithmetic; see John von Neumann, *Collected Works* **5** (New York: Macmillan, 1963), 127–142. Another early discussion of radix conversion on binary computers was published by F. Koons and S. Lubkin, *Math. Comp.* **3** (1949), 427–431, who suggested a rather unusual method. The first discussion of floating point conversion was given somewhat later by F. L. Bauer and K. Samelson [*Zeit. für angewandte Math. und Physik* **4** (1953), 312–316].

The following articles may be useful for further reference: A note by G. T. Lake [*CACM* **5** (1962), 468–469] mentions some hardware techniques for conversion and gives clear examples. A. H. Stroud and D. Secrest [*Comp. J.* **6** (1963), 62–66] have discussed conversion of multiple-precision floating point numbers. The conversion of *unnormalized* floating point numbers, preserving the amount of "significance" implied by the representation, has been discussed by H. Kanner [*JACM* **12** (1965), 242–246] and by N. Metropolis and R. L. Ashenhurst [*Math. Comp.* **19** (1965), 435–441]. See also K. Sikdar, *Sankhyā* (B) **30** (1968), 315–334, and the references cited in his paper.

EXERCISES

▶ **1.** [*25*] Generalize Method (1b) so that it works with arbitrary mixed-radix notations, converting

$$a_m b_{m-1} \ldots b_1 b_0 + \cdots + a_1 b_0 + a_0 \quad \text{into} \quad A_M B_{M-1} \ldots B_1 B_0 + \cdots + A_1 B_0 + A_0,$$

where $0 \le a_j < b_j$ and $0 \le A_J < B_J$ for $0 \le j < m$ and $0 \le J < M$.

Give an example of your generalization by manually converting the quantity "3 days, 9 hours, 12 minutes, and 37 seconds" into long tons, hundredweights, stones, pounds, and ounces. (Let one second equal one ounce. The British system of weights has 1 stone = 14 pounds, 1 hundredweight = 8 stone, 1 long ton = 20 hundredweight.) In other words, let $b_0 = 60$, $b_1 = 60$, $b_2 = 24$, $m = 3$, $B_0 = 16$, $B_1 = 14$, $B_2 = 8$, $B_3 = 20$, $M = 4$; the problem is to find $A_4, \ldots, A_0$ in the proper ranges such that $3b_2b_1b_0 + 2b_1b_0 + 12b_0 + 37 = A_4B_3B_2B_1B_0 + A_3B_2B_1B_0 + A_2B_1B_0 + A_1B_0 + A_0$, using a systematic method that generalizes Method (1b). (All arithmetic is to be done in a mixed-radix system.)

2. [*25*] Generalize Method (1a) so that it works with mixed-radix notations, as in exercise 1, and give an example of your generalization by manually solving the same conversion problem stated in exercise 1.

▶ **3.** [*25*] (D. Taranto.) When fractions are being converted, there is no obvious way to decide how many digits to give in the answer. Design a simple generalization of Method (2a) that, given two positive radix-b fractions u and ϵ between 0 and 1, converts u to a rounded radix-B equivalent U that has just enough places M to the right of the radix point to ensure that $|U - u| < \epsilon$. (In particular if u is a multiple of b^{-m} and $\epsilon = b^{-m}/2$, the value of U will have just enough digits so that u can be recomputed exactly, given U and m. Note that M might be zero; for example, if $\epsilon \le \frac{1}{2}$ and $u > 1 - \epsilon$, the proper answer is $U = 1$.)

4. [*M21*] (a) Prove that every real number with a terminating *binary* representation also has a terminating *decimal* representation. (b) Find a simple condition on the positive integers b and B that is satisfied if and only if every real number that has a terminating radix-b representation also has a terminating radix-B representation.

5. [*M20*] Show that program (4) would still work if the instruction "LDX =10^n=" were replaced by "LDX =c=" for certain other constants c.

6. [*30*] Discuss using Methods (1a), (1b), (2a), and (2b) when b or B is -2.

7. [*M18*] Given that $0 < \alpha \le x \le \alpha + 1/w$ and $0 \le u \le w$, prove that $\lfloor ux \rfloor$ is equal to either $\lfloor \alpha u \rfloor$ or $\lfloor \alpha u \rfloor + 1$. Furthermore $\lfloor ux \rfloor = \lfloor \alpha u \rfloor$ exactly, if $u < \alpha w$ and α^{-1} is an integer.

8. [*24*] Write a MIX program analogous to (1) that uses (5) and includes no division instructions.

9. [*M27*] Let u be an integer, $0 \le u < 2^{34}$. Assume that the following sequence of operations (equivalent to addition and binary "shift-right" instructions) is performed:

$$v \leftarrow \lfloor \tfrac{1}{2}u \rfloor, \quad v \leftarrow v + \lfloor \tfrac{1}{2}v \rfloor, \quad v \leftarrow v + \lfloor 2^{-4}v \rfloor,$$
$$v \leftarrow v + \lfloor 2^{-8}v \rfloor, \quad v \leftarrow v + \lfloor 2^{-16}v \rfloor, \quad v \leftarrow \lfloor \tfrac{1}{8}v \rfloor.$$

Prove that $v = \lfloor u/10 \rfloor$ or $\lfloor u/10 \rfloor - 1$.

10. [*22*] The text shows how a binary-coded decimal number can be doubled by using various shifting, extracting, and addition operations on a binary computer. Give an analogous method that computes *half* of a binary-coded decimal number (throwing away the remainder if the number is odd).

11. [*16*] Convert $(57721)_8$ to decimal.

▶ **12.** [*22*] Invent a rapid pencil-and-paper method for converting integers from ternary notation to decimal, and illustrate your method by converting $(1212011210210)_3$ into decimal. How would you go from decimal to ternary?

▶ **13.** [*25*] Assume that locations U + 1, U + 2, ..., U + m contain a multiple-precision fraction $(.u_{-1}u_{-2}\ldots u_{-m})_b$, where b is the word size of MIX. Write a MIX routine that converts this fraction to decimal notation, truncating it to 180 decimal digits. The answer should be printed on two lines, with the digits grouped into 20 blocks of nine each separated by blanks. (Use the CHAR instruction.)

▶ **14.** [*M27*] (A. Schönhage.) The text's method of converting multiple-precision integers requires an execution time of order n^2 to convert an n-place integer, when n is large. Show that it is possible to convert n-digit decimal integers into binary notation in $O(M(n)\log n)$ steps, where $M(n)$ is an upper bound on the number of steps needed to multiply n-bit binary numbers that satisfies the "smoothness condition" $M(2n) \ge 2M(n)$.

15. [*M47*] Can the upper bound on the time to convert large integers, given in exercise 14, be substantially lowered? (Cf. exercise 4.3.3–12.)

16. [*41*] Construct a fast linear iterative array for radix conversion from decimal to binary (cf. Section 4.3.3).

17. [*M40*] Design "ideal" floating point conversion subroutines, taking p-digit decimal numbers into P-digit binary numbers and vice versa, in both cases producing a true rounded result in the sense of Section 4.2.2.

18. [*HM34*] (David W. Matula.) Let $\text{round}_b(u, p)$ be the function of b, u, and p that represents the best p-digit base b floating point approximation to u, in the sense of Section 4.2.2. Under the assumption that $\log_B b$ is irrational and that the range of exponents is unlimited, prove that

$$u = \text{round}_b(\text{round}_B(u, P), p)$$

holds for all p-digit base b floating point numbers u if and only if $B^{P-1} \geq b^p$. (In other words, an "ideal" input conversion of u into an independent base B, followed by an "ideal" output conversion of this result, will always yield u again if and only if the intermediate precision P is suitably large, as specified by the formula above.)

19. [*M23*] Let the decimal number $u = (u_7 \ldots u_1 u_0)_{10}$ be represented as the binary-coded decimal number $U = (u_7 \ldots u_1 u_0)_{16}$. Find appropriate constants c_i and masks m_i so that the operation $U \leftarrow U - c_i(U \wedge m_i)$, repeated for $i = 1, 2, 3$, will convert U to the binary representation of u, where "$\wedge$" denotes extraction (i.e., "logical AND" on individual bits).

4.5. RATIONAL ARITHMETIC

IT IS OFTEN important to know that the answer to some numerical problem is exactly $\frac{1}{3}$, not a floating point number that gets printed as "0.333333574". If arithmetic is done on fractions instead of on approximations to fractions, many computations can be done entirely *without any accumulated rounding errors.* This results in a comfortable feeling of security that is often lacking when floating point calculations have been made, and it means that the accuracy of the calculation cannot be improved upon.

4.5.1. Fractions

When fractional arithmetic is desired, the numbers can be represented as pairs of integers, (u/u'), where u and u' are relatively prime to each other and $u' > 0$. The number zero is represented as $(0/1)$. In this form, $(u/u') = (v/v')$ if and only if $u = v$ and $u' = v'$.

Multiplication of fractions is, of course, easy; to form $(u/u') \times (v/v') = (w/w')$, we can simply compute uv and $u'v'$. The two products uv and $u'v'$ might not be relatively prime, but if $d = \gcd(uv, u'v')$, the desired answer is $w = uv/d$, $w' = u'v'/d$. (See exercise 2.) Efficient algorithms to compute the greatest common divisor are discussed in Section 4.5.2.

Another way to perform the multiplication is to find $d_1 = \gcd(u, v')$ and $d_2 = \gcd(u', v)$; then the answer is $w = (u/d_1)(v/d_2)$, $w' = (u'/d_2)(v'/d_1)$. (See exercise 3.) This method requires two gcd calculations, but it is not really slower than the former method; the gcd process involves a number of iterations that is essentially proportional to the logarithm of its inputs, so the total number of iterations needed to evaluate both d_1 and d_2 is essentially the same as the number of iterations during the single calculation of d. Furthermore, each iteration in the evaluation of d_1 and d_2 is potentially faster, because comparatively small numbers are being examined. If u, u', v, and v' are single-precision quantities, this method has the advantage that no double-precision numbers appear in the calculation unless it is impossible to represent both of the answers w and w' in single-precision form.

Division may be done in a similar manner; see exercise 4.

Addition and subtraction are slightly more complicated. The obvious procedure is to set $(u/u') \pm (v/v') = \big((uv' \pm u'v)/u'v'\big)$ and then to reduce this fraction to lowest terms by calculating $d = \gcd(uv' \pm u'v, u'v')$ as in the first multiplication method. But again it is possible to avoid working with such large numbers, if we start by calculating $d_1 = \gcd(u', v')$. If $d_1 = 1$, then the desired numerator and denominator are $w = uv' \pm u'v$ and $w' = u'v'$. (According to Theorem 4.5.2D, d_1 will be 1 about 61 percent of the time, if the denominators u' and v' are randomly distributed, so it is wise to single this case out separately.) If $d_1 > 1$, then let $t = u(v'/d_1) \pm v(u'/d_1)$ and calculate $d_2 = \gcd(t, d_1)$; finally the answer is $w = t/d_2$, $w' = (u'/d_1)(v'/d_2)$. (Exercise 6 proves that these values of w and w' are relatively prime to each other.) If single-precision

numbers are being used, this method requires only single-precision operations, except that t may be a double-precision number or slightly larger (see exercise 7); since $\gcd(t, d_1) = \gcd(t \bmod d_1, d_1)$, the calculation of d_2 does not require double precision.

For example, to compute $(7/66) + (17/12)$, we form $d_1 = \gcd(66, 12) = 6$; then $t = 7 \cdot 2 + 17 \cdot 11 = 201$, and $d_2 = \gcd(201, 6) = 3$, so the answer is

$$\frac{201}{3} \Big/ \frac{66}{6}\,\frac{12}{3} = 67/44.$$

To help check out subroutines for rational arithmetic, inversion of matrices with known inverses (e.g., Cauchy matrices, exercise 1.2.3–41) is suggested.

Experience with fractional calculations shows that in many cases the numbers grow to be quite large. So if u and u' are intended to be single-precision numbers for each fraction (u/u'), it is important to include tests for overflow in each of the addition, subtraction, multiplication, and division subroutines. For numerical problems in which perfect accuracy is important, a set of subroutines for fractional arithmetic with *arbitrary* precision allowed in numerator and denominator is very useful.

The methods of this section extend also to other number fields besides the rational numbers; for example, we could do arithmetic on quantities of the form $(u + u'\sqrt{5}\,)/u''$, where u, u', u'' are integers, $\gcd(u, u', u'') = 1$, and $u'' > 0$; or on quantities of the form $(u + u'\sqrt[3]{2} + u''\sqrt[3]{4}\,)/u'''$, etc.

Instead of insisting on exact calculations with fractions, it is interesting to consider also "fixed slash" and "floating slash" numbers, which are analogous to floating point numbers but based on rational fractions instead of radix-oriented fractions. In a binary fixed-slash scheme, the numerator and denominator of a representable fraction each consist of at most p bits, for some given p. In a floating-slash scheme, the *sum* of numerator bits plus denominator bits must be a total of at most q, for some given q, and another field of the representation is used to indicate how many of these q bits belong to the numerator. To do arithmetic on such numbers, we define $x \oplus y = \text{round}(x + y)$, $x \ominus y = \text{round}(x - y)$, etc., where $\text{round}(x) = x$ if x is representable, otherwise it is one of the two representable numbers that surround x.

It may seem at first that the best definition of $\text{round}(x)$ would be to choose the representable number that is closest to x, by analogy with the way we round in floating point arithmetic. But experience has shown that it is best to bias our rounding towards "simple" numbers, since numbers with small numerator and denominator occur much more often than complicated fractions do. We want more numbers to round to $\frac{1}{2}$ than to $\frac{127}{255}$. The rounding rule that turns out to be most successful in practice is called "mediant rounding": If (u/u') and (v/v') are adjacent representable numbers, so that whenever $u/u' \le x \le v/v'$ we must have $\text{round}(x)$ equal to (u/u') or (v/v'), the mediant rounding rule says that

$$\text{round}(x) = \frac{u}{u'} \text{ for } x < \frac{u+v}{u'+v'}, \quad \text{round}(x) = \frac{v}{v'} \text{ for } x > \frac{u+v}{u'+v'}. \qquad (1)$$

If $x = (u + v)/(u' + v')$ exactly, we let round(x) be the neighboring fraction with the smallest denominator (or, if $u' = v'$, with the smallest numerator).

For example, suppose we are doing fixed slash arithmetic with $p = 8$, so that the representable numbers (u/u') have $-128 < u < 128$ and $0 \le v < 256$ and $\gcd(u, u') = 1$. This isn't much precision, but it is enough to give us a feel for slash arithmetic. The numbers adjacent to $0 = (0/1)$ are $(-1/255)$ and $(1/255)$; according to the mediant rounding rule, we will therefore have round$(x) = 0$ if and only if $|x| \le 1/256$. Suppose we have a calculation that would take the overall form $\frac{22}{7} = \frac{314}{159} + \frac{1300}{1113}$ if we were working in exact rational arithmetic, but the intermediate quantities have had to be rounded to representable numbers. In this case $\frac{314}{159}$ would round to $(79/40)$ and $\frac{1300}{1113}$ would round to $(7/6)$. The sum $\frac{79}{40} + \frac{7}{6} = \frac{377}{120}$ rounds to $(22/7)$, so we have obtained the correct answer even though three roundings were required. This example was not specially contrived; when the answer to a problem is a simple fraction, slash arithmetic tends to make the intermediate rounding errors cancel out.

Exact representation of fractions within a computer was first discussed in the literature by P. Henrici, *JACM* **3** (1956), 6–9. Fixed and floating slash arithmetic was proposed by David W. Matula, in *Applications of Number Theory to Numerical Analysis*, ed. by S. K. Zaremba (New York: Academic Press, 1972), 486–489. Further developments of the idea are discussed by Matula and Kornerup in *Proc. IEEE Symp. Computer Arith.* **4** (1978), 29–47; *Lecture Notes in Comp. Sci.* **72** (1979), 383–397; *Computing*, Suppl. **2** (1980), 85–111.

EXERCISES

1. [*15*] Suggest a reasonable computational method for comparing two fractions, to test whether or not $(u/u') < (v/v')$.

2. [*M15*] Prove that if $d = \gcd(u, v)$ then u/d and v/d are relatively prime.

3. [*M20*] Prove that if u and u' are relatively prime, and if v and v' are relatively prime, then $\gcd(uv, u'v') = \gcd(u, v')\gcd(u', v)$.

4. [*11*] Design a division algorithm for fractions, analogous to the second multiplication method of the text. (Note that the sign of v must be considered.)

5. [*10*] Compute $(17/120) + (-27/70)$ by the method recommended in the text.

▶ **6.** [*M23*] Show that if u, u' are relatively prime and if v, v' are relatively prime, then $\gcd(uv' + vu', u'v') = d_1 d_2$, where $d_1 = \gcd(u', v')$ and $d_2 = \gcd(d_1, u(v'/d_1) + v(u'/d_1))$. (Hence if $d_1 = 1$, then $uv' + vu'$ is relatively prime to $u'v'$.)

7. [*M22*] How large can the absolute value of the quantity t become, in the addition-subtraction method recommended in the text, if the numerators and denominators of the inputs are less than N in absolute value?

▶ **8.** [*22*] Discuss using $(1/0)$ and $(-1/0)$ as representations for ∞ and $-\infty$, and/or as representations of overflow.

9. [*M23*] If $1 \le u', v' < 2^n$, show that $\lfloor 2^{2n} u/u' \rfloor = \lfloor 2^{2n} v/v' \rfloor$ implies $u/u' = v/v'$.

10. [*41*] Extend the subroutines suggested in exercise 4.3.1–34 so that they deal with "arbitrary" rational numbers.

11. [*M23*] Consider fractions of the form $(u+u'\sqrt{5})/u''$, where u, u', u'' are integers, $\gcd(u, u', u'') = 1$, and $u'' > 0$. Explain how to divide two such fractions and to obtain a quotient having the same form.

12. [*20*] (Matula and Kornerup.) Discuss the representation of floating slash numbers in a 32-bit word.

13. [*M23*] Explain how to compute the exact number of pairs of integers (u, u') such that $1 \le u \le M$ and $N_1 < u' \le N_2$ and $\gcd(u, u') = 1$. (This can be used to determine how many numbers are representable in slash arithmetic. According to Theorem 4.5.2D, the number will be approximately $(6/\pi^2)M(N_2 - N_1)$.)

14. [*42*] Modify one of the compilers at your installation so that it will replace all floating point calculations by floating slash calculations. Experiment with the use of slash arithmetic by running existing programs that were written by programmers who actually had floating point arithmetic in mind. (When special subroutines like square root or logarithm are called, your system should automatically convert slash numbers to floating point form before the subroutine is invoked, then back to slash form again afterwards. There should be a new option to print slash numbers in a fractional format; however, if you make no changes to a user's source program, you probably will have to print slash numbers in decimal notation, in order to keep from messing up any column alignments.) Are the results better or worse, when floating slash numbers are substituted?

4.5.2. The Greatest Common Divisor

If u and v are integers, not both zero, we say that their *greatest common divisor*, $\gcd(u, v)$, is the largest integer that evenly divides both u and v. This definition makes sense, because if $u \neq 0$ then no integer greater than $|u|$ can evenly divide u, but the integer 1 does divide both u and v; hence there must be a largest integer that divides them both. When u and v are both zero, every integer evenly divides zero, so the above definition does not apply; it is convenient to set

$$\gcd(0, 0) = 0. \tag{1}$$

The definitions just given obviously imply that

$$\gcd(u, v) = \gcd(v, u), \tag{2}$$

$$\gcd(u, v) = \gcd(-u, v), \tag{3}$$

$$\gcd(u, 0) = |u|. \tag{4}$$

In the previous section, we reduced the problem of expressing a rational number in "lowest terms" to the problem of finding the greatest common divisor of its numerator and denominator. Other applications of the greatest common divisor have been mentioned for example in Sections 3.2.1.2, 3.3.3, 4.3.2, 4.3.3. So the concept of $\gcd(u, v)$ is important and worthy of serious study.

The *least common multiple* of two integers u and v, written $\operatorname{lcm}(u, v)$, is a related idea that is also important. It is defined to be the smallest positive integer

that is a multiple of (i.e., evenly divisible by) both u and v; and $\operatorname{lcm}(0,0) = 0$. The classical method for teaching children how to add fractions $u/u' + v/v'$ is to train them to find the "least common denominator," which is $\operatorname{lcm}(u', v')$.

According to the "fundamental theorem of arithmetic" (proved in exercise 1.2.4–21), each positive integer u can be expressed in the form

$$u = 2^{u_2}3^{u_3}5^{u_5}7^{u_7}11^{u_{11}}\ldots = \prod_{p \text{ prime}} p^{u_p}, \tag{5}$$

where the exponents u_2, u_3, ... are uniquely determined nonnegative integers, and where all but a finite number of the exponents are zero. From this canonical factorization of a positive integer, it is easy to discover one way to compute the greatest common divisor of u and v: By (2), (3), and (4), we may assume that u and v are positive integers, and if both of them have been canonically factored into primes we have

$$\gcd(u,v) = \prod_{p \text{ prime}} p^{\min(u_p,v_p)}, \tag{6}$$

$$\operatorname{lcm}(u,v) = \prod_{p \text{ prime}} p^{\max(u_p,v_p)}. \tag{7}$$

Thus, for example, the greatest common divisor of $u = 7000 = 2^3 \cdot 5^3 \cdot 7$ and $v = 4400 = 2^4 \cdot 5^2 \cdot 11$ is $2^{\min(3,4)}\,5^{\min(3,2)}\,7^{\min(1,0)}\,11^{\min(0,1)} = 2^3 \cdot 5^2 = 200$. The least common multiple of the same two numbers is $2^4 \cdot 5^3 \cdot 7 \cdot 11 = 154000$.

From formulas (6) and (7) we can easily prove a number of basic identities concerning the gcd and the lcm:

$$\gcd(u,v)w = \gcd(uw,vw), \qquad \text{if } w \geq 0; \tag{8}$$

$$\operatorname{lcm}(u,v)w = \operatorname{lcm}(uw,vw), \qquad \text{if } w \geq 0; \tag{9}$$

$$u \cdot v = \gcd(u,v) \cdot \operatorname{lcm}(u,v), \qquad \text{if } u,v \geq 0; \tag{10}$$

$$\gcd\big(\operatorname{lcm}(u,v), \operatorname{lcm}(u,w)\big) = \operatorname{lcm}\big(u, \gcd(v,w)\big); \tag{11}$$

$$\operatorname{lcm}\big(\gcd(u,v), \gcd(u,w)\big) = \gcd\big(u, \operatorname{lcm}(v,w)\big). \tag{12}$$

The latter two formulas are "distributive laws" analogous to the familiar identity $uv + uw = u(v + w)$. Equation (10) reduces the calculation of $\gcd(u,v)$ to the calculation of $\operatorname{lcm}(u,v)$, and conversely.

Euclid's algorithm. Although Eq. (6) is useful for theoretical purposes, it is generally no help for calculating a greatest common divisor in practice, because it requires that we first determine the factorization of u and v. There is no known method for finding the prime factors of an integer very rapidly (see Section 4.5.4). But fortunately there is an efficient way to calculate the greatest common divisor of two integers without factoring them, and, in fact, such a method was discovered over 2250 years ago; this is "Euclid's algorithm," which we have already examined in Sections 1.1 and 1.2.1.

Euclid's algorithm is found in Book 7, Propositions 1 and 2 of his *Elements* (*c.* 300 B.C.), but it probably wasn't his own invention. Scholars believe that the method was known up to 200 years earlier, at least in its subtractive form, and it was almost certainly known to Eudoxus (*c.* 375 B.C.); cf. K. von Fritz, *Ann. Math.* (2) **46** (1945), 242–264. We might call it the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day. (The chief rival for this honor is perhaps the ancient Egyptian method for multiplication, which was based on doubling and adding, and which forms the basis for efficient calculation of nth powers as explained in Section 4.6.3. But the Egyptian manuscripts merely give examples that are not completely systematic, and these examples were certainly not stated systematically; the Egyptian method is therefore not quite deserving of the name "algorithm." Several ancient Babylonian methods, for doing such things as solving special sets of quadratic equations in two variables, are also known. Genuine algorithms are involved in this case, not just special solutions to the equations for certain input parameters; for even though the Babylonians invariably presented each method in conjunction with an example worked with particular input data, they regularly explained the general procedure in the accompanying text. [See D. E. Knuth, *CACM* **15** (1972), 671–677; **19** (1976), 108.] Many of these Babylonian algorithms predate Euclid by 1500 years, and they are the earliest known instances of written procedures for mathematics. But they do not have the stature of Euclid's algorithm, since they do not involve iteration and since they have been superseded by modern algebraic methods.)

In view of the importance of Euclid's algorithm, for historical as well as practical reasons, let us now consider how Euclid himself treated it. Paraphrasing his words into modern terminology, this is essentially what he wrote:

Proposition. *Given two positive integers, find their greatest common divisor.*

Let A, C be the two given positive integers; it is required to find their greatest common divisor. If C divides A, then C is a common divisor of C and A, since it also divides itself. And it clearly is in fact the greatest, since no greater number than C will divide C.

But if C does not divide A, then continually subtract the lesser of the numbers A, C from the greater, until some number is left that divides the previous one. This will eventually happen, for if unity is left, it will divide the previous number.

Now let E be the positive remainder of A divided by C; let F be the positive remainder of C divided by E; and let F be a divisor of E. Since F divides E and E divides $C - F$, F also divides $C - F$; but it also divides itself, so it divides C. And C divides $A - E$; therefore F also divides $A - E$. But it also divides E; therefore it divides A. Hence it is a common divisor of A and C.

I now claim that it is also the greatest. For if F is not the greatest common divisor of A and C, some larger number will divide them both. Let such a number be G.

Now since G divides C while C divides $A-E$, G divides $A-E$. G also divides the whole of A, so it divides the remainder E. But E divides $C - F$; therefore G also divides $C - F$. And G also divides the whole of C, so it divides the remainder F; that is, a greater number divides a smaller one. This is impossible.

Therefore no number greater than F will divide A and C, so F is their greatest common divisor.

Corollary. This argument makes it evident that any number dividing two numbers divides their greatest common divisor. *Q.E.D.*

Note. Euclid's statements have been simplified here in one nontrivial respect: Greek mathematicians did not regard unity as a "divisor" of another positive integer. Two positive integers were either both equal to unity, or they were relatively prime, or they had a greatest common divisor. In fact, unity was not even considered to be a "number," and zero was of course nonexistent. These rather awkward conventions made it necessary for Euclid to duplicate much of his discussion, and he gave two separate propositions that are each essentially like the one appearing here.

In his discussion, Euclid first suggests subtracting the smaller of the two current numbers from the larger, repeatedly, until we get two numbers where one is a multiple of the other. But in the proof he really relies on taking the remainder of one number divided by another; and since he has no simple concept of zero, he cannot speak of the remainder when one number divides the other. It is reasonable to say that he imagines each *division* (not the individual subtractions) as a single step of the algorithm, and hence an "authentic" rendition of his algorithm can be phrased as follows:

Algorithm E (*Original Euclidean algorithm*). Given two integers A and C greater than unity, this algorithm finds their greatest common divisor.

E1. [A divisible by C?] If C divides A, the algorithm terminates with C as the answer.

E2. [Replace A by remainder.] If $A \bmod C$ is equal to unity, the given numbers were relatively prime, so the algorithm terminates. Otherwise replace the pair of values (A, C) by $(C, A \bmod C)$ and return to step E1. ∎

The "proof" Euclid gave, which is quoted above, is especially interesting because it is not really a proof at all! He verifies the result of the algorithm only if step E1 is performed once or thrice. Surely he must have realized that step E1 could take place more than three times, although he made no mention of such a possibility. Not having the notion of a proof by mathematical induction, he could only give a proof for a finite number of cases. (In fact, he often proved only the case $n = 3$ of a theorem that he wanted to establish for general n.) Although Euclid is justly famous for the great advances he made in the art of logical deduction, techniques for giving valid proofs by induction were not discovered until many centuries later, and the crucial ideas for proving the validity of *algorithms* are only now becoming really clear. (See Section 1.2.1 for a complete proof of Euclid's algorithm, together with a short discussion of general proof procedures for algorithms.)

It is worth noting that this algorithm for finding the greatest common divisor was chosen by Euclid to be the very first step in his development of the theory of numbers. The same order of presentation is still in use today in modern

textbooks. Euclid also gave a method (Proposition 34) to find the least common multiple of two integers u and v, namely to divide u by $\gcd(u, v)$ and to multiply the result by v; this is equivalent to Eq. (10).

If we avoid Euclid's bias against the numbers 0 and 1, we can reformulate Algorithm E in the following way.

Algorithm A (*Modern Euclidean algorithm*). Given nonnegative integers u and v, this algorithm finds their greatest common divisor. (*Note:* The greatest common divisor of *arbitrary* integers u and v may be obtained by applying this algorithm to $|u|$ and $|v|$, because of Eqs. (2) and (3).)

A1. [$v = 0$?] If $v = 0$, the algorithm terminates with u as the answer.

A2. [Take $u \bmod v$.] Set $r \leftarrow u \bmod v$, $u \leftarrow v$, $v \leftarrow r$, and return to A1. (The operations of this step decrease the value of v, but they leave $\gcd(u, v)$ unchanged.) ∎

For example, we may calculate $\gcd(40902, 24140)$ as follows:

$$\begin{aligned}\gcd(40902, 24140) &= \gcd(24140, 16762) = \gcd(16762, 7378)\\ &= \gcd(7378, 2006) = \gcd(2006, 1360) = \gcd(1360, 646)\\ &= \gcd(646, 68) = \gcd(68, 34) = \gcd(34, 0) = 34.\end{aligned}$$

A proof that Algorithm A is valid follows readily from Eq. (4) and the fact that

$$\gcd(u, v) = \gcd(v, u - qv), \tag{13}$$

if q is any integer. Equation (13) holds because any common divisor of u and v is a divisor of both v and $u - qv$, and, conversely, any common divisor of v and $u - qv$ must divide both u and v.

The following MIX program illustrates the fact that Algorithm A can easily be implemented on a computer:

Program A (*Euclid's algorithm*). Assume that u and v are single-precision, nonnegative integers, stored respectively in locations U and V; this program puts $\gcd(u, v)$ into rA.

```
    LDX  U     1      rX ← u.
    JMP  2F    1
1H  STX  V     T      v ← rX.
    SRAX 5     T      rAX ← rA.
    DIV  V     T      rX ← rAX mod v.
2H  LDA  V     1+T    rA ← v.
    JXNZ 1B    1+T    Done if rX = 0.   ∎
```

The running time for this program is $19T + 6$ cycles, where T is the number of divisions performed. The discussion in Section 4.5.3 shows that we may take $T = 0.842766 \ln N + 0.06$ as an approximate average value, when u and v are independently and uniformly distributed in the range $1 \le u, v \le N$.

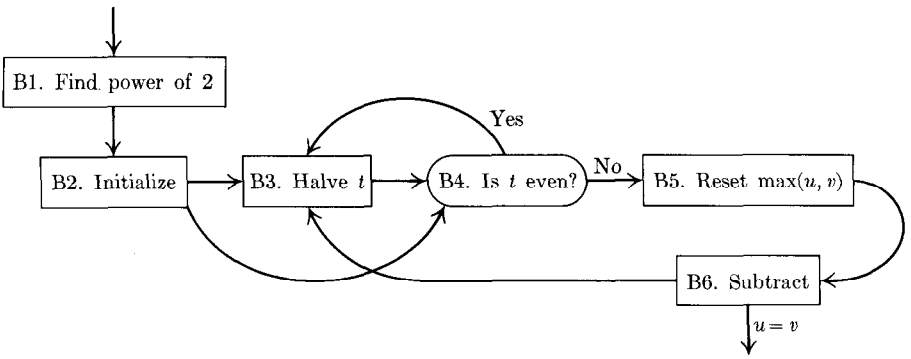


Fig. 9. Binary algorithm for the greatest common divisor.

A binary method. Since Euclid's patriarchal algorithm has been used for so many centuries, it is a rather surprising fact that it may not always be the best method for finding the greatest common divisor after all. A quite different gcd algorithm, which is primarily suited to binary arithmetic, was discovered by J. Stein in 1961 [see *J. Comp. Phys.* **1** (1967), 397–405]. This new algorithm requires no division instruction; it relies solely on the operations of (i) subtraction, (ii) testing whether a number is even or odd, and (iii) shifting the binary representation of an even number to the right (halving).

The binary gcd algorithm is based on four simple facts about positive integers u and v:

a) If u and v are both even, then $\gcd(u, v) = 2\gcd(u/2, v/2)$. [See Eq. (8).]
b) If u is even and v is odd, then $\gcd(u, v) = \gcd(u/2, v)$. [See Eq. (6).]
c) As in Euclid's algorithm, $\gcd(u, v) = \gcd(u - v, v)$. [See Eqs. (13), (2).]
d) If u and v are both odd, then $u - v$ is even, and $|u - v| < \max(u, v)$.

These facts immediately suggest the following algorithm:

Algorithm B (*Binary gcd algorithm*). Given positive integers u and v, this algorithm finds their greatest common divisor.

B1. [Find power of 2.] Set $k \leftarrow 0$, and then repeatedly set $k \leftarrow k + 1$, $u \leftarrow u/2$, $v \leftarrow v/2$, zero or more times until u and v are not both even.

B2. [Initialize.] (Now the original values of u and v have been divided by 2^k, and at least one of their present values is odd.) If u is odd, set $t \leftarrow -v$ and go to B4. Otherwise set $t \leftarrow u$.

B3. [Halve t.] (At this point, t is even, and nonzero.) Set $t \leftarrow t/2$.

B4. [Is t even?] If t is even, go back to B3.

B5. [Reset $\max(u, v)$.] If $t > 0$, set $u \leftarrow t$; otherwise set $v \leftarrow -t$. (The larger of u and v has been replaced by $|t|$, except perhaps during the first time this step is performed.)

B6. [Subtract.] Set $t \leftarrow u - v$. If $t \neq 0$, go back to B3. Otherwise the algorithm terminates with $u \cdot 2^k$ as the output. ∎

As an example of Algorithm B, let us consider $u = 40902$, $v = 24140$, the same numbers we have used to try out Euclid's algorithm. Step B1 sets $k \leftarrow 1$, $u \leftarrow 20451$, $v \leftarrow 12070$. Then t is set to -12070, and replaced by -6035; then v is replaced by 6035, and the computation proceeds as follows:

u	v	t
20451	6035	$+14416, +7208, +3604, +1802, +901;$
901	6035	$-5134, -2567;$
901	2567	$-1666, -833;$
901	833	$+68, +34, +17;$
17	833	$-816, -408, -204, -102, -51;$
17	51	$-34, -17;$
17	17	0.

The answer is $17 \cdot 2^1 = 34$. A few more iterations were necessary here than we needed with Algorithm A, but each iteration was somewhat simpler since no division steps were used.

A MIX program for Algorithm B requires just a little more code than for Algorithm A. In order to make such a program fairly typical of a binary computer's representation of Algorithm B, let us assume that MIX is extended to include the following operators:

• SLB (shift left AX binary). C = 6; F = 6.
The contents of registers A and X are "shifted left" M binary places; that is, $|\mathrm{rAX}| \leftarrow |2^M \mathrm{rAX}| \bmod B^{10}$, where B is the byte size. (As with all MIX shift commands, the signs of rA and rX are not affected.)

• SRB (shift right AX binary). C = 6; F = 7.
The contents of registers A and X are "shifted right" M binary places; that is, $|\mathrm{rAX}| \leftarrow \lfloor |\mathrm{rAX}|/2^M \rfloor$.

• JAE, JAO (jump A even, jump A odd). C = 40; F = 6, 7, respectively.
A JMP occurs if rA is even or odd, respectively.

• JXE, JXO (jump X even, jump X odd). C = 47; F = 6, 7, respectively.
Analogous to JAE, JAO.

Program B (*Binary gcd algorithm*). Assume that u and v are single-precision positive integers, stored respectively in locations U and V; this program uses Algorithm B to put $\gcd(u, v)$ into rA. Register assignments: $t \equiv$ rA, $k \equiv$ rI1.

```
01  ABS EQU  1:5
02  B1  ENT1 0          1     B1. Find power of 2.
03      LDX  U          1     rX ← u.
04      LDAN V          1     rA ← −v.
05      JMP  1F         1
06  2H  SRB  1          A     Halve rA, rX.
07      INC1 1          A     k ← k + 1.
08      STX  U          A     u ← u/2.
09      STA  V(ABS)     A     v ← v/2.
```

```
10  1H  JXO  B4      1 + A      To B4 with t ← −v if u is odd.
11  B2  JAE  2B      B + A      B2. Initialize.
12      LDA  U       B          t ← u.
13  B3  SRB  1       D          B3. Halve t.
14  B4  JAE  B3      1 − B + D  B4. Is t even?
15  B5  JAN  1F      C          B5. Reset max(u,v).
16      STA  U       E          If t > 0, set u ← t.
17      SUB  V       E          t ← u − v.
18      JMP  2F      E
19  1H  STA  V(ABS)  C − E      If t < 0, set v ← −t.
20  B6  ADD  U       C − E      B6. Subtract.
21  2H  JANZ B3      C          To B3 if t ≠ 0.
22      LDA  U       1          rA ← u.
23      ENTX 0       1          rX ← 0.
24      SLB  0,1     1          rA ← 2^k · rA.  ∎
```

The running time of this program is

$$9A + 2B + 6C + 3D + E + 13$$

units, where $A = k$, $B = 1$ if $t \leftarrow u$ in step B2 (otherwise $B = 0$), C is the number of subtraction steps, D is the number of halvings in step B3, and E is the number of times $t > 0$ in step B5. Calculations discussed later in this section imply that we may take $A = \frac{1}{3}$, $B = \frac{1}{3}$, $C = 0.71n - 0.5$, $D = 1.41n - 2.7$, $E = 0.35n - 0.4$ as average values for these quantities, assuming random inputs u and v in the range $1 \leq u, v < 2^n$. The total running time is therefore about $8.8n + 5$ cycles, compared to about $11.1n + 7$ for Program A under the same assumptions. The worst possible running time for u and v in this range occurs when $A = 0$, $B = 0$, $C = n$, $D = 2n - 2$, $E = 1$; this amounts to $12n + 8$ cycles. (The corresponding value for Program A is $26.8n + 19$.)

Thus the greater speed of the iterations in Program B, due to the simplicity of the operations, compensates for the greater number of iterations required. We have found that the binary algorithm is about 20 percent faster than Euclid's algorithm on the MIX computer. Of course, the situation may be different on other computers, and in any event both programs are quite efficient; but it appears that not even a procedure as venerable as Euclid's algorithm can withstand progress.

V. C. Harris [*Fibonacci Quarterly* 8 (1970), 102–103] has suggested an interesting cross between Euclid's algorithm and the binary algorithm. If u and v are odd, with $u \geq v > 0$, we can always write $u = qv \pm r$ where $0 \leq r < v$ and r is even; if $r \neq 0$ we set $r \leftarrow r/2$ until r is odd, then set $u \leftarrow v$, $v \leftarrow r$ and repeat the process. In subsequent iterations, $q \geq 3$.

Extensions. We can extend the methods used to calculate $\gcd(u, v)$ in order to solve some slightly more difficult problems. For example, assume that we want to compute the greatest common divisor of n integers $u_1, u_2, \ldots, u_n$.

One way to calculate $\gcd(u_1, u_2, \ldots, u_n)$, assuming that the u's are all nonnegative, is to extend Euclid's algorithm in the following way: If all u_j are zero,

the greatest common divisor is taken to be zero; otherwise if only one u_j is nonzero, it is the greatest common divisor; otherwise replace u_k by $u_k \bmod u_j$ for all $k \neq j$, where u_j is the minimum of the nonzero u's.

The algorithm sketched in the preceding paragraph is a natural generalization of Euclid's method, and it can be justified in a similar manner. But there is a simpler method available, based on the easily verified identity

$$\gcd(u_1, u_2, \ldots, u_n) = \gcd\bigl(u_1, \gcd(u_2, \ldots, u_n)\bigr). \tag{14}$$

To calculate $\gcd(u_1, u_2, \ldots, u_n)$, we may therefore proceed as follows:

C1. Set $d \leftarrow u_n$, $j \leftarrow n - 1$.

C2. If $d \neq 1$ and $j > 0$, set $d \leftarrow \gcd(u_j, d)$ and $j \leftarrow j - 1$ and repeat this step. Otherwise $d = \gcd(u_1, \ldots, u_n)$.

This method reduces the calculation of $\gcd(u_1, \ldots, u_n)$ to repeated calculations of the greatest common divisor of two numbers at a time. It makes use of the fact that $\gcd(u_1, \ldots, u_j, 1) = 1$; and this will be helpful, since we will already have $\gcd(u_{n-1}, u_n) = 1$ over 60 percent of the time if u_{n-1} and u_n are chosen at random. In most cases, the value of d will decrease rapidly during the first few stages of the calculation, and this will make the remainder of the computation quite fast. Here Euclid's algorithm has an advantage over Algorithm B, in that its running time is primarily governed by the value of $\min(u, v)$, while the running time for Algorithm B is primarily governed by $\max(u, v)$; it would be reasonable to perform one iteration of Euclid's algorithm, replacing u by $u \bmod v$ if u is much larger than v, and then to continue with Algorithm B.

The assertion that $\gcd(u_{n-1}, u_n)$ will be equal to unity more than 60 percent of the time for random inputs is a consequence of the following well-known result of number theory:

Theorem D (G. Lejeune Dirichlet, *Abhandlungen Königlich Preuß. Akad. Wiss.* (1849), 69–83). *If u and v are integers chosen at random, the probability that $\gcd(u, v) = 1$ is $6/\pi^2 \approx .60793$.*

A precise formulation of this theorem, which carefully defines what is meant by being "chosen at random," appears in exercise 10 with a rigorous proof. Let us content ourselves here with a heuristic argument that shows why the theorem is plausible.

If we assume, without proof, the existence of a well-defined probability p that $\gcd(u, v)$ equals unity, then we can determine the probability that $\gcd(u, v) = d$ for any positive integer d, because $\gcd(u, v) = d$ if and only if u is a multiple of d and v is a multiple of d and $\gcd(u/d, v/d) = 1$. Thus the probability that $\gcd(u, v) = d$ is equal to $1/d$ times $1/d$ times p, namely p/d^2. Now let us sum these probabilities over all possible values of d; we should get

$$1 = \sum_{d \geq 1} p/d^2 = p(1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \cdots).$$

Since the sum $1 + \frac{1}{4} + \frac{1}{9} + \cdots = H_\infty^{(2)}$ is equal to $\pi^2/6$ (cf. Section 1.2.7), we need $p = 6/\pi^2$ in order to make this equation come out right. ∎

Euclid's algorithm can be extended in another important way: We can calculate integers u' and v' such that

$$uu' + vv' = \gcd(u, v) \tag{15}$$

at the same time $\gcd(u, v)$ is being calculated. This extension of Euclid's algorithm can be described conveniently in vector notation:

Algorithm X (*Extended Euclid's algorithm*). Given nonnegative integers u and v, this algorithm determines a vector (u_1, u_2, u_3) such that $uu_1 + vu_2 = u_3 = \gcd(u, v)$. The computation makes use of auxiliary vectors (v_1, v_2, v_3), (t_1, t_2, t_3); all vectors are manipulated in such a way that the relations

$$ut_1 + vt_2 = t_3, \qquad uu_1 + vu_2 = u_3, \qquad uv_1 + vv_2 = v_3 \tag{16}$$

hold throughout the calculation.

X1. [Initialize.] Set $(u_1, u_2, u_3) \leftarrow (1, 0, u)$, $(v_1, v_2, v_3) \leftarrow (0, 1, v)$.

X2. [Is $v_3 = 0$?] If $v_3 = 0$, the algorithm terminates.

X3. [Divide, subtract.] Set $q \leftarrow \lfloor u_3/v_3 \rfloor$, and then set

$$(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - (v_1, v_2, v_3)q,$$
$$(u_1, u_2, u_3) \leftarrow (v_1, v_2, v_3), \qquad (v_1, v_2, v_3) \leftarrow (t_1, t_2, t_3).$$

Return to step X2. ∎

For example, let $u = 40902$, $v = 24140$. At step X2 we have

q	u_1	u_2	u_3	v_1	v_2	v_3
—	1	0	40902	0	1	24140
1	0	1	24140	1	−1	16762
1	1	−1	16762	−1	2	7378
2	−1	2	7378	3	−5	2006
3	3	−5	2006	−10	17	1360
1	−10	17	1360	13	−22	646
2	13	−22	646	−36	61	68
9	−36	61	68	337	−571	34
2	337	−571	34	−710	1203	0

The solution is therefore $337 \cdot 40902 - 571 \cdot 24140 = 34 = \gcd(40902, 24140)$.

The validity of Algorithm X follows from (16) and the fact that the algorithm is identical to Algorithm A with respect to its manipulation of u_3 and v_3; a detailed proof of Algorithm X is discussed in Section 1.2.1. Gordon H. Bradley has observed that we can avoid a good deal of the calculation in Algorithm X by suppressing u_2, v_2, and t_2; then u_2 can be determined afterwards using the relation $uu_1 + vu_2 = u_3$.

Exercise 14 shows that the values of $|u_1|$, $|u_2|$, $|v_1|$, $|v_2|$ remain bounded by the size of the inputs u and v. Algorithm B, which computes the greatest common divisor using properties of binary notation, can be extended in a similar way; see exercise 35. For some instructive extensions to Algorithm X, see exercises 18 and 19 in Section 4.6.1.

The ideas underlying Euclid's algorithm can also be applied to find a *general solution in integers* of any set of linear equations with integer coefficients. For example, suppose that we want to find all integers w, x, y, z that satisfy the two equations

$$10w + 3x + 3y + 8z = 1, \tag{17}$$

$$6w - 7x \qquad - 5z = 2. \tag{18}$$

We can introduce a new variable

$$\lfloor 10/3 \rfloor w + \lfloor 3/3 \rfloor x + \lfloor 3/3 \rfloor y + \lfloor 8/3 \rfloor z = 3w + x + y + 2z = t_1,$$

and use it to eliminate y; Eq. (17) becomes

$$(10 \bmod 3)w + (3 \bmod 3)x + 3t_1 + (8 \bmod 3)z = w + 3t_1 + 2z = 1, \tag{19}$$

and Eq. (18) remains unchanged. The new equation (19) may be used to eliminate w, and (18) becomes

$$6(1 - 3t_1 - 2z) - 7x - 5z = 2;$$

that is,

$$7x + 18t_1 + 17z = 4. \tag{20}$$

Now as before we introduce a new variable

$$x + 2t_1 + 2z = t_2$$

and eliminate x from (20):

$$7t_2 + 4t_1 + 3z = 4. \tag{21}$$

Another new variable can be introduced in the same fashion, in order to eliminate the variable z, which has the smallest coefficient:

$$2t_2 + t_1 + z = t_3.$$

Eliminating z from (21) yields

$$t_2 + t_1 + 3t_3 = 4, \tag{22}$$

and this equation, finally, can be used to eliminate t_2. We are left with two independent variables, t_1 and t_3; substituting back for the original variables, we obtain the general solution

$$\begin{aligned} w &= 17 - 5t_1 - 14t_3, \\ x &= 20 - 5t_1 - 17t_3, \\ y &= -55 + 19t_1 + 45t_3, \\ z &= -8 + t_1 + 7t_3. \end{aligned} \tag{23}$$

In other words, all integer solutions (w, x, y, z) to the original equations (17), (18) are obtained from (23) by letting t_1 and t_3 independently run through all integers.

The general method that has just been illustrated is based on the following procedure: Find a nonzero coefficient c of smallest absolute value in the system of equations. Suppose that this coefficient appears in an equation having the form

$$cx_0 + c_1x_1 + \cdots + c_kx_k = d; \tag{24}$$

assume for simplicity that $c > 0$. If $c = 1$, use this equation to eliminate the variable x_0 from the other equations remaining in the system; then repeat the procedure on the remaining equations. (If no more equations remain, the computation stops, and a general solution in terms of the variables not yet eliminated has essentially been obtained.) If $c > 1$, then if $c_1 \bmod c = \cdots = c_k \bmod c = 0$ check that $d \bmod c = 0$, otherwise there is no integer solution; then divide both sides of (24) by c and eliminate x_0 as in the case $c = 1$. Finally, if $c > 1$ and not all of $c_1 \bmod c$, ..., $c_k \bmod c$ are zero, then introduce a new variable

$$\lfloor c/c \rfloor x_0 + \lfloor c_1/c \rfloor x_1 + \cdots + \lfloor c_k/c \rfloor x_k = t; \tag{25}$$

eliminate the variable x_0 from the other equations, in favor of t, and replace the original equation (24) by

$$ct + (c_1 \bmod c)x_1 + \cdots + (c_k \bmod c)x_k = d. \tag{26}$$

(Cf. (19) and (21) in the above example.)

This process must terminate, since each step reduces either the number of equations or the size of the smallest nonzero coefficient in the system. A study of the above procedure will reveal its intimate connection with Euclid's algorithm. The method is a comparatively simple means of solving linear equations when the variables are required to take on integer values only. It isn't the best available method for this problem, however; substantial refinements are possible, but beyond the scope of this book.

High-precision calculation. If u and v are very large integers, requiring a multiple-precision representation, the binary method (Algorithm B) is a simple and fairly efficient means of calculating their greatest common divisor, since it involves only subtractions and shifting.

By contrast, Euclid's algorithm seems much less attractive, since step A2 requires a multiple-precision division of u by v. But this difficulty is not really as bad as it seems, since we will prove in Section 4.5.3 that the quotient $\lfloor u/v \rfloor$ is almost always very small; for example, assuming random inputs, the quotient $\lfloor u/v \rfloor$ will be less than 1000 approximately 99.856 percent of the time. Therefore it is almost always possible to find $\lfloor u/v \rfloor$ and $(u \bmod v)$ using single-precision calculations, together with the comparatively simple operation of calculating $u - qv$ where q is a single-precision number. Furthermore, if it does turn out that u is much larger than v (e.g., the initial input data might have this form),

we don't really mind having a large quotient q, since Euclid's algorithm makes a great deal of progress when it replaces u by $u \bmod v$ in such a case.

A significant improvement in the speed of Euclid's algorithm when high-precision numbers are involved can be achieved by using a method due to D. H. Lehmer [*AMM* **45** (1938), 227–233]. Working only with the leading digits of large numbers, it is possible to do most of the calculations with single-precision arithmetic, and to make a substantial reduction in the number of multiple-precision operations involved. We save a lot of time by doing a "virtual" calculation instead of the actual one.

For example, let us consider the pair of eight-digit numbers $u = 27182818$, $v = 10000000$, assuming that we are using a machine with only four-digit words. Let $u' = 2718$, $v' = 1001$, $u'' = 2719$, $v'' = 1000$; then u'/v' and u''/v'' are approximations to u/v, with

$$u'/v' < u/v < u''/v''. \tag{27}$$

The ratio u/v determines the sequence of quotients obtained in Euclid's algorithm. If we carry out Euclid's algorithm simultaneously on the single-precision values (u', v') and (u'', v'') until we get a different quotient, it is not difficult to see that the same sequence of quotients would have appeared to this point if we had worked with the multiple-precision numbers (u, v). Thus, consider what happens when Euclid's algorithm is applied to (u', v') and to (u'', v''):

u'	v'	q'	u''	v''	q''
2718	1001	2	2719	1000	2
1001	716	1	1000	719	1
716	285	2	719	281	2
285	146	1	281	157	1
146	139	1	157	124	1
139	7	19	124	33	3

The first five quotients are the same in both cases, so they must be the true ones. But on the sixth step we find that $q' \neq q''$, so the single-precision calculations are suspended. We have gained the knowledge that the calculation would have proceeded as follows if we had been working with the original multiple-precision numbers:

u	v	q
u_0	v_0	2
v_0	$u_0 - 2v_0$	1
$u_0 - 2v_0$	$-u_0 + 3v_0$	2
$-u_0 + 3v_0$	$3u_0 - 8v_0$	1
$3u_0 - 8v_0$	$-4u_0 + 11v_0$	1
$-4u_0 + 11v_0$	$7u_0 - 19v_0$	?

(28)

(The next quotient lies somewhere between 3 and 19.) No matter how many digits are in u and v, the first five steps of Euclid's algorithm would be the same as (28),

so long as (27) holds. We can therefore avoid the multiple-precision operations of the first five steps, and replace them all by a multiple-precision calculation of $-4u_0 + 11v_0$ and $7u_0 - 19v_0$. In this case we obtain $u = 1268728$, $v = 279726$; the calculation can now proceed in a similar manner with $u' = 1268$, $v' = 280$, $u'' = 1269$, $v'' = 279$, etc. If we had a larger accumulator, more steps could be done by single-precision calculations; our example showed that only five cycles of Euclid's algorithm were combined into one multiple step, but with (say) a word size of 10 digits we could do about twelve cycles at a time. (Results proved in Section 4.5.3 imply that the number of multiple-precision cycles that can be replaced at each iteration is essentially proportional to the number of digits used in the single-precision calculations.)

Lehmer's method can be formulated as follows:

Algorithm L (*Euclid's algorithm for large numbers*). Let u, v be nonnegative integers, with $u \geq v$, represented in multiple precision. This algorithm computes the greatest common divisor of u and v, making use of auxiliary single-precision p-digit variables $\hat{u}$, $\hat{v}$, A, B, C, D, T, q, and auxiliary multiple-precision variables t and w.

L1. [Initialize.] If v is small enough to be represented as a single-precision value, calculate $\gcd(u, v)$ by Algorithm A and terminate the computation. Otherwise, let $\hat{u}$ be the p leading digits of u, and let $\hat{v}$ be the corresponding digits of v; in other words, if radix-b notation is being used, $\hat{u} \leftarrow \lfloor u/b^k \rfloor$ and $\hat{v} \leftarrow \lfloor v/b^k \rfloor$, where k is as small as possible consistent with the condition $\hat{u} < b^p$.

Set $A \leftarrow 1$, $B \leftarrow 0$, $C \leftarrow 0$, $D \leftarrow 1$. (These variables represent the coefficients in (28), where

$$u = Au_0 + Bv_0, \qquad v = Cu_0 + Dv_0, \tag{29}$$

in the equivalent actions of Algorithm A on multiple-precision numbers. We also have

$$u' = \hat{u} + B, \qquad v' = \hat{v} + D, \qquad u'' = \hat{u} + A, \qquad v'' = \hat{v} + C \tag{30}$$

in terms of the notation in the example worked above.)

L2. [Test quotient.] Set $q \leftarrow \lfloor (\hat{u} + A)/(\hat{v} + C) \rfloor$. If $q \neq \lfloor (\hat{u} + B)/(\hat{v} + D) \rfloor$, go to step L4. (This step tests if $q' \neq q''$, in the notation of the above example. Note that single-precision overflow can occur in special circumstances during the computation in this step, but only when $\hat{u} = b^p - 1$ and $A = 1$ or when $\hat{v} = b^p - 1$ and $D = 1$; the conditions

$$\begin{aligned} 0 \leq \hat{u} + A \leq b^p, \qquad & 0 \leq \hat{v} + C < b^p, \\ 0 \leq \hat{u} + B < b^p, \qquad & 0 \leq \hat{v} + D \leq b^p \end{aligned} \tag{31}$$

will always hold, because of (30). It is possible to have $\hat{v} + C = 0$ or $\hat{v} + D = 0$, but not both simultaneously; therefore division by zero in this step is taken to mean "Go directly to L4.")

L3. [Emulate Euclid.] Set $T \leftarrow A - qC$, $A \leftarrow C$, $C \leftarrow T$, $T \leftarrow B - qD$, $B \leftarrow D$, $D \leftarrow T$, $T \leftarrow \hat{u} - q\hat{v}$, $\hat{u} \leftarrow \hat{v}$, $\hat{v} \leftarrow T$, and go back to step L2. (These single-precision calculations are the equivalent of multiple-precision operations, as in (28), under the conventions of (29).)

L4. [Multiprecision step.] If $B = 0$, set $t \leftarrow u \bmod v$, $u \leftarrow v$, $v \leftarrow t$, using multiple-precision division. (This happens only if the single-precision operations cannot simulate any of the multiple-precision ones. It implies that Euclid's algorithm requires a very large quotient, and this is an extremely rare occurrence.) Otherwise, set $t \leftarrow Au$, $t \leftarrow t + Bv$, $w \leftarrow Cu$, $w \leftarrow w + Dv$, $u \leftarrow t$, $v \leftarrow w$ (using straightforward multiple-precision operations). Go back to step L1. ∎

The values of A, B, C, D remain as single-precision numbers throughout this calculation, because of (31).

Algorithm L requires a somewhat more complicated program than Algorithm B, but with large numbers it will be faster on many computers. The binary technique of Algorithm B can, however, be speeded up in a similar way (see exercise 34), to the point where it continues to win. Algorithm L has the advantage that it can readily be extended, as in Algorithm X (see exercise 17); furthermore, it determines the sequence of quotients obtained in Euclid's algorithm, and this yields the regular continued fraction expansion of a real number (see exercise 4.5.3–18).

Analysis of the binary algorithm. Let us conclude this section by studying the running time of Algorithm B, in order to justify the formulas stated earlier.

An exact determination of the behavior of Algorithm B appears to be exceedingly difficult to derive, but we can begin to study it by means of an approximate model of its behavior. Suppose that u and v are odd numbers, with $u > v$ and

$$\lfloor \lg u \rfloor = m, \qquad \lfloor \lg v \rfloor = n. \tag{32}$$

(Thus, u is an $(m+1)$-bit number, and v is an $(n+1)$-bit number.) Algorithm B forms $u - v$ and shifts this quantity right until obtaining an odd number u' that replaces u. Under random conditions, we would expect to have $u' = (u - v)/2$ about one-half of the time, $u' = (u - v)/4$ about one-fourth of the time, $u' = (u - v)/8$ about one-eighth of the time, and so on. We have

$$\lfloor \lg u' \rfloor = m - k - r, \tag{33}$$

where k is the number of places that $u - v$ is shifted right, and where r is $\lfloor \lg u \rfloor - \lfloor \lg(u - v) \rfloor$, the number of bits lost at the left during the subtraction of v from u. Note that $r \leq 1$ when $m \geq n + 2$, and $r \geq 1$ when $m = n$. For simplicity, we will assume that $r = 0$ when $m \neq n$ and that $r = 1$ when $m = n$, although this lower bound tends to make u' seem larger than it usually is.

The approximate model we shall use to study Algorithm B is based solely on the values $m = \lfloor \lg u \rfloor$ and $n = \lfloor \lg v \rfloor$ throughout the course of the algorithm,

not on the actual values of u and v. Let us call this approximation a *lattice-point model*, since we will say that we are "at the point (m, n)" when $\lfloor \lg u \rfloor = m$ and $\lfloor \lg v \rfloor = n$. From point (m, n) the algorithm takes us to (m', n) if $u > v$, or to (m, n') if $u < v$, or terminates if $u = v$. For example, the calculation starting with $u = 20451$ and $v = 6035$ begins at the point $(14, 12)$, then goes to $(9, 12)$, $(9, 11)$, $(9, 9)$, $(4, 9)$, $(4, 5)$, $(4, 4)$, and terminates. In line with the comments of the preceding paragraph, we will make the following assumptions about the probability that we reach a given point just after point (m, n):

Case 1, $m > n$.

Next point	Probability
$(m-1, n)$	$\frac{1}{2}$
$(m-2, n)$	$\frac{1}{4}$
$\ldots$	$\ldots$
$(1, n)$	$(\frac{1}{2})^{m-1}$
$(0, n)$	$(\frac{1}{2})^{m-1}$

Case 2, $m < n$.

Next point	Probability
$(m, n-1)$	$\frac{1}{2}$
$(m, n-2)$	$\frac{1}{4}$
$\ldots$	$\ldots$
$(m, 1)$	$(\frac{1}{2})^{n-1}$
$(m, 0)$	$(\frac{1}{2})^{n-1}$

Case 3, $m = n > 0$.

Next point	Probability
$(m-2, n)$, $(m, n-2)$	$\frac{1}{4}$, $\frac{1}{4}$
$(m-3, n)$, $(m, n-3)$	$\frac{1}{8}$, $\frac{1}{8}$
$\ldots$	$\ldots$
$(0, n)$, $(m, 0)$	$(\frac{1}{2})^m$, $(\frac{1}{2})^m$
terminate	$(\frac{1}{2})^{m-1}$

For example, from points $(5, 3)$ the lattice-point model would go to points $(4, 3)$, $(3, 3)$, $(2, 3)$, $(1, 3)$, $(0, 3)$ with the respective probabilities $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{16}$; from $(4, 4)$ it would go to $(2, 4)$, $(1, 4)$, $(0, 4)$, $(4, 2)$, $(4, 1)$, $(4, 0)$, or would terminate, with the respective probabilities $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$, $\frac{1}{8}$. When m and n are both 0, the formulas above do not apply; the algorithm always terminates in such a case, since $m = n = 0$ implies that $u = v = 1$.

The detailed calculations of exercise 18 show that this lattice-point model is somewhat pessimistic. In fact, when $m > 3$ the actual probability that Algorithm B goes from (m, m) to one of the two points $(m-2, m)$ or $(m, m-2)$ is equal to $\frac{1}{8}$, although we have assumed the value $\frac{1}{2}$; the algorithm actually goes from (m, m) to $(m-3, m)$ or $(m, m-3)$ with probability $\frac{7}{32}$, not $\frac{1}{4}$; and it actually goes from $(m+1, m)$ to (m, m) with probability $\frac{1}{8}$, not $\frac{1}{2}$. The probabilities in the model are nearly correct when $|m - n|$ is large, but when $|m - n|$ is small the model predicts slower convergence than is actually obtained. In spite of the fact that our model is not a completely faithful representation of Algorithm B, it has one great virtue: It can be completely analyzed! Furthermore, empirical experiments with Algorithm B show that the behavior predicted by the lattice-point model is analogous to the true behavior.

An analysis of the lattice-point model can be carried out by solving the following rather complicated set of double recurrence relations:

$$\begin{aligned} A_{mm} &= a + \frac{1}{2}A_{m(m-2)} + \cdots + \frac{1}{2^{m-1}}A_{m0} + \frac{b}{2^{m-1}}, && \text{if } m \geq 1; \\ A_{mn} &= c + \frac{1}{2}A_{(m-1)n} + \cdots + \frac{1}{2^{m-1}}A_{1n} + \frac{1}{2^{m-1}}A_{0n}, && \text{if } m > n \geq 0; \\ A_{mn} &= A_{nm}, && \text{if } n > m \geq 0. \end{aligned} \tag{34}$$

The problem is to solve for A_{mn} in terms of m, n, and the parameters a, b, c, and A_{00}. This is an interesting set of recurrence equations, which have an interesting solution.

First we observe that if $0 \leq n < m$,

$$\begin{aligned} A_{(m+1)n} &= c + \sum_{1\leq k\leq m} 2^{-k}A_{(m+1-k)n} + 2^{-m}A_{0n} \\ &= c + \tfrac{1}{2}A_{mn} + \sum_{1\leq k<m} 2^{-k-1}A_{(m-k)n} + 2^{-m}A_{0n} \\ &= c + \tfrac{1}{2}A_{mn} + \tfrac{1}{2}(A_{mn} - c) \\ &= \tfrac{1}{2}c + A_{mn}. \end{aligned}$$

Hence $A_{(m+k)n} = \frac{1}{2}ck + A_{mn}$, by induction on k. In particular, since $A_{10} = c + A_{00}$, we have

$$A_{m0} = \tfrac{1}{2}c(m+1) + A_{00}, \qquad m > 0. \tag{35}$$

Now let $A_m = A_{mm}$. If $m > 0$, we have

$$\begin{aligned} A_{(m+1)m} &= c + \sum_{1\leq k\leq m+1} 2^{-k}A_{(m+1-k)m} + 2^{-m-1}A_{0m} \\ &= c + \tfrac{1}{2}A_{mm} + \sum_{1\leq k\leq m} \big(2^{-k-1}(A_{(m-k)(m+1)} - c/2)\big) + 2^{-m-1}A_{0m} \\ &= c + \tfrac{1}{2}A_m + \tfrac{1}{2}(A_{(m+1)(m+1)} - a - 2^{-m}b) - \tfrac{1}{4}c(1-2^{-m}) \\ &\qquad + 2^{-m-1}\big(\tfrac{1}{2}c(m+1) + A_{00}\big) \\ &= \tfrac{1}{2}(A_m + A_{m+1}) + \tfrac{3}{4}c - \tfrac{1}{2}a + 2^{-m-1}(c - b + A_{00}) + m2^{-m-2}c. \end{aligned} \tag{36}$$

Similar maneuvering, as shown in exercise 19, establishes the relation

$$A_{n+1} = \tfrac{3}{4}A_n + \tfrac{1}{4}A_{n-1} + \alpha + 2^{-n-1}\beta + (n+2)2^{-n-1}\gamma, \quad n \geq 2, \tag{37}$$

where $\alpha = \frac{1}{4}a + \frac{7}{8}c$, $\beta = A_{00} - b - \frac{3}{2}c$, and $\gamma = \frac{1}{2}c$.

Thus the double recurrence (34) can be transformed into the single recurrence relation in (37). Use of the generating function $G(z) = A_0 + A_1 z + A_2 z^2 + \cdots$ now transforms (37) into the equation

$$(1 - \tfrac{3}{4}z - \tfrac{1}{4}z^2)G(z) = a_0 + a_1 z + a_2 z^2 + \frac{\alpha}{1-z} + \frac{\beta}{1-z/2} + \frac{\gamma}{(1-z/2)^2}, \quad (38)$$

where a_0, a_1, and a_2 are constants that can be determined by the values of A_0, A_1, and A_2. Since $1 - \frac{3}{4}z + \frac{1}{4}z^2 = (1 + \frac{1}{4}z)(1 - z)$, we can express $G(z)$ by the method of partial fractions in the form

$$G(z) = b_0 + b_1 z + \frac{b_2}{(1-z)^2} + \frac{b_3}{1-z} + \frac{b_4}{(1-z/2)^2} + \frac{b_5}{1-z/2} + \frac{b_6}{1+z/4}.$$

Tedious manipulations produce the values of these constants $b_0, \ldots, b_6$, and thus the coefficients of $G(z)$ are determined. We finally obtain the solution

$$\begin{aligned} A_{nn} &= n(\tfrac{1}{5}a + \tfrac{7}{10}c) + (\tfrac{16}{25}a + \tfrac{2}{5}b - \tfrac{23}{50}c + \tfrac{3}{5}A_{00}) \\ &\quad + 2^{-n}(-\tfrac{1}{3}cn + \tfrac{2}{3}b - \tfrac{1}{9}c - \tfrac{2}{3}A_{00}) \\ &\quad + (-\tfrac{1}{4})^n(-\tfrac{16}{25}a - \tfrac{16}{15}b + \tfrac{16}{225}c + \tfrac{16}{15}A_{00}) + \tfrac{1}{2}c\delta_{n0}; \\ A_{mn} &= \tfrac{1}{2}mc + n(\tfrac{1}{5}a + \tfrac{1}{5}c) + (\tfrac{6}{25}a + \tfrac{2}{5}b + \tfrac{7}{50}c + \tfrac{3}{5}A_{00}) + 2^{-n}(\tfrac{1}{3}c) \\ &\quad + (-\tfrac{1}{4})^n(-\tfrac{6}{25}a - \tfrac{2}{5}b + \tfrac{2}{75}c + \tfrac{2}{5}A_{00}), \qquad m > n. \end{aligned} \quad (39)$$

With these elaborate calculations done, we can readily determine the behavior of the lattice-point model. Assume that the inputs u and v to the algorithm are odd, and let $m = \lfloor \lg u \rfloor$, $n = \lfloor \lg v \rfloor$. The average number of subtraction cycles, namely the quantity C in the analysis of Program B, is obtained by setting $a = 1$, $b = 0$, $c = 1$, $A_{00} = 1$ in the recurrence (34). By (39) we see that (for $m \geq n$) the lattice model predicts

$$C = \tfrac{1}{2}m + \tfrac{2}{5}n + \tfrac{49}{50} - \tfrac{1}{5}\delta_{mn} \quad (40)$$

subtraction cycles, plus terms that rapidly go to zero as n approaches infinity.

The average number of times that $\gcd(u, v) = 1$ is obtained by setting $a = b = c = 0$, $A_{00} = 1$; this gives the probability that u and v are relatively prime, approximately $\frac{3}{5}$. Actually, since u and v are assumed to be odd, they should be relatively prime with probability $8/\pi^2$ (see exercise 13), so this reflects the degree of inaccuracy of the lattice-point model.

The average number of times that a path from (m, n) goes through one of the "diagonal" points (m', m') for some $m' \geq 1$ is obtained by setting $a = 1$, $b = c = A_{00} = 0$ in (34); so we find that this quantity is approximately

$$\tfrac{1}{5}n + \tfrac{6}{25} + \tfrac{2}{5}\delta_{mn}, \qquad \text{when } m \geq n.$$

Now we can determine the average number of shifts, i.e., the number of times that step B3 is performed. (This is the quantity D in Program B.) In any

Table 1

NUMBER OF SUBTRACTIONS IN ALGORITHM B

	n						n						
	0	1	2	3	4	5	0	1	2	3	4	5	
0	1.00	2.00	2.50	3.00	3.50	4.00	1.00	2.00	2.50	3.00	3.50	4.00	0
1	2.00	1.00	2.50	3.00	3.50	4.00	2.00	1.00	3.00	2.75	3.63	3.94	1
2	2.50	2.50	2.25	3.38	3.88	4.38	2.50	3.00	2.00	3.50	3.88	4.25	2
3	3.00	3.00	3.38	3.25	4.22	4.72	3.00	2.75	3.50	2.88	4.13	4.34	3
4	3.50	3.50	3.88	4.22	4.25	5.10	3.50	3.63	3.88	4.13	3.94	4.80	4
5	4.00	4.00	4.38	4.72	5.10	5.19	4.00	3.94	4.25	4.34	4.80	4.60	5
m	Predicted by model						Actual average values						m

execution of Algorithm B, with u and v both odd, the corresponding path in the lattice model must satisfy the relation

$$\text{number of shifts} + \text{number of diagonal points} + 2\lfloor \lg \gcd(u,v) \rfloor = m + n,$$

since we are assuming that r in (33) is always either 0 or 1. The average value of $\lfloor \lg \gcd(u,v) \rfloor$ predicted by the lattice-point model is approximately $\frac{4}{5}$ (see exercise 20). Therefore we have, for the total number of shifts,

$$\begin{aligned} D &= m + n - (\tfrac{1}{5}n + \tfrac{6}{25} + \tfrac{2}{5}\delta_{mn}) - \tfrac{4}{5} \\ &= m + \tfrac{4}{5}n - \tfrac{46}{25} - \tfrac{2}{5}\delta_{mn}, \end{aligned} \tag{41}$$

when $m \geq n$, plus terms that decrease rapidly to zero for large n.

To summarize the most important facts we have derived from the lattice-point model, we have shown that if u and v are odd and if $\lfloor \lg u \rfloor = m$, $\lfloor \lg v \rfloor = n$, then the quantities C and D that are the critical factors in the running time of Program B will have average values given by

$$C = \tfrac{1}{2}m + \tfrac{2}{5}n + O(1), \qquad D = m + \tfrac{4}{5}n + O(1), \qquad m \geq n. \tag{42}$$

But the model that we have used to derive (42) is only a pessimistic approximation to the true behavior; Table 1 compares the true average values of C, computed by actually running Algorithm B with all possible inputs, to the values predicted by the lattice-point model, for small m and n. The lattice model is completely accurate when m or n is zero, but it tends to be less accurate when $|m - n|$ is small and $\min(m,n)$ is large. When $m = n = 9$, the lattice-point model gives $C = 8.78$, compared to the true value $C = 7.58$.

Empirical tests of Algorithm B with several million random inputs and with various values of m, n in the range $29 \leq m, n \leq 37$ indicate that the actual average behavior of the algorithm is given by

$$\begin{aligned} C &\approx \tfrac{1}{2}m + 0.203n + 1.9 - 0.4(0.6)^{m-n}, \\ D &\approx \quad m + 0.41n \quad - 0.5 - 0.7(0.6)^{m-n}, \end{aligned} \qquad m \geq n. \tag{43}$$

These experiments showed a rather small standard deviation from the observed average values. The coefficients $\frac{1}{2}$ and 1 of m in (42) and (43) can be verified rigorously without using the lattice-point approximation (see exercise 21); so the error in the lattice-point model is apparently in the coefficient of n, which is too high.

The above calculations have been made under the assumption that u and v are odd and in the ranges $2^m \le u < 2^{m+1}$ and $2^n \le v < 2^{n+1}$. If we assume instead that u and v are to be *any* integers, independently and uniformly distributed over the ranges

$$1 \le u < 2^N, \qquad 1 \le v < 2^N,$$

then we can calculate the average values of C and D from the data already given; in fact, if C_{mn} denotes the average value of C under our earlier assumptions, exercise 22 shows that we have

$$\begin{aligned}(2^N-1)^2C = N^2C_{00} &+ 2N\sum_{1\le n\le N}(N-n)2^{n-1}C_{n0}\\ &+ 2\sum_{1\le n<m\le N}(N-m)(N-n)2^{m+n-2}C_{mn}\\ &+ \sum_{1\le n\le N}(N-n)^2 2^{2n-2}C_{nn}.\end{aligned} \tag{44}$$

The same formula holds for D in terms of D_{mn}. If the indicated sums are carried out using the approximations in (43), we obtain

$$C \approx 0.70N + O(1), \qquad D \approx 1.41N + O(1).$$

(See exercise 23.) This agrees perfectly with the results of further empirical tests, made on several million random inputs for $N \le 30$; the latter tests show that we may take

$$C = 0.70N - 0.5, \qquad D = 1.41N - 2.7 \tag{45}$$

as good estimates of the values, given this distribution of the inputs u and v.

Richard Brent has discovered a continuous model that accounts for the leading terms in (45). Let us assume that u and v are large, and that the number of shifts per iteration has the value d with probability exactly 2^{-d}. If we let $X = u/v$, the effect of steps B3–B5 is to replace X by $(X-1)/2^d$ if $X > 1$, or by $2^d/(X^{-1}-1)$ if $X < 1$. The random variable X has a limiting distribution that makes it possible to analyze the average value of the ratio by which $\max(u, v)$ decreases at each iteration; see exercise 25. Numerical calculations show that this maximum decreases by $\beta = 0.705971246102$ bits per iteration; the agreement with experiment is so good that Brent's constant β must be the true value of the number "0.70" in (45), and we should replace 0.203 by 0.206 in (43). [See *Algorithms and Complexity*, ed. by J. F. Traub (New York: Academic Press, 1976), 321–355.]

This completes our analysis of the average values of C and D. The other three quantities appearing in the running time of Algorithm B are rather easily analyzed; see exercises 6, 7, and 8.

Thus we know approximately how Algorithm B behaves on the average. Let us now consider a "worst case" analysis: What values of u and v are in some sense the hardest to handle? If we assume as before that

$$\lfloor \lg u \rfloor = m \qquad \text{and} \qquad \lfloor \lg v \rfloor = n,$$

let us try to find (u, v) that make the algorithm run most slowly. In view of the fact that the subtractions take somewhat longer than the shifts, when the auxiliary bookkeeping is considered, this question may be rephrased by asking for the inputs u and v that require most subtractions. The answer is somewhat surprising; the maximum value of C is exactly

$$\max(m, n) + 1, \tag{46}$$

although the lattice-point model would predict that substantially higher values of C are possible (see exercise 26). The derivation of the worst case (46) is quite interesting, so it has been left as an amusing problem for the reader to work out by himself (see exercises 27 and 28).

EXERCISES

1. [*M21*] How can (8), (9), (10), (11), and (12) be derived easily from (6) and (7)?

2. [*M22*] Given that u divides $v_1 v_2 \ldots v_n$, prove that u divides

$$\gcd(u, v_1)\gcd(u, v_2)\ldots\gcd(u, v_n).$$

3. [*M23*] Show that the number of ordered pairs of positive integers (u, v) such that $\operatorname{lcm}(u, v) = n$ is the number of divisors of n^2.

4. [*M21*] Given positive integers u and v, show that there are divisors u' of u and v' of v such that $\gcd(u', v') = 1$ and $u'v' = \operatorname{lcm}(u, v)$.

▶ **5.** [*M26*] Invent an algorithm (analogous to Algorithm B) for calculating the greatest common divisor of two integers based on their *balanced ternary* representation. Demonstrate your algorithm by applying it to the calculation of $\gcd(40902, 24140)$.

6. [*M22*] Given that u and v are random positive integers, find the mean and the standard deviation of the quantity A that enters into the timing of Program B. (This is the number of right shifts applied to both u and v during the preparatory phase.)

7. [*M20*] Analyze the quantity B that enters into the timing of Program B.

▶ **8.** [*M25*] Show that in Program B, the average value of E is approximately equal to $\frac{1}{2}C_{\text{ave}}$, where C_{ave} is the average value of C.

9. [*18*] Using Algorithm B and hand calculation, find $\gcd(31408, 2718)$. Also find integers m and n such that $31408m + 2718n = \gcd(31408, 2718)$, using Algorithm X.

▶ **10.** [*HM24*] Let q_n be the number of ordered pairs of integers (u, v) lying in the range $1 \le u, v \le n$ such that $\gcd(u, v) = 1$. The object of this exercise is to prove that we have $\lim_{n\to\infty} q_n/n^2 = 6/\pi^2$, thereby establishing Theorem D.

a) Use the principle of inclusion and exclusion (Section 1.3.3) to show that

$$q_n = n^2 - \sum_{p_1} \lfloor n/p_1 \rfloor^2 + \sum_{p_1 < p_2} \lfloor n/p_1 p_2 \rfloor^2 - \cdots,$$

where the sums are taken over all *prime* numbers p_i.

b) The *Möbius function* $\mu(n)$ is defined by the rules $\mu(1) = 1$, $\mu(p_1 p_2 \ldots p_r) = (-1)^r$ if $p_1, p_2, \ldots, p_r$ are distinct primes, and $\mu(n) = 0$ if n is divisible by the square of a prime. Show that $q_n = \sum_{k\ge 1} \mu(k)\lfloor n/k \rfloor^2$.

c) As a consequence of (b), prove that $\lim_{n\to\infty} q_n/n^2 = \sum_{k\ge 1} \mu(k)/k^2$.

d) Prove that $(\sum_{k\ge 1} \mu(k)/k^2)(\sum_{m\ge 1} 1/m^2) = 1$. *Hint:* When the series are absolutely convergent we have

$$\Big(\sum_{k\ge 1} a_k/k^z\Big)\Big(\sum_{m\ge 1} b_m/m^z\Big) = \sum_{n\ge 1}\Big(\sum_{d\backslash n} a_d b_{n/d}\Big)\Big/ n^z.$$

11. [*M22*] What is the probability that $\gcd(u, v) \le 3$? (See Theorem D.) What is the *average* value of $\gcd(u, v)$?

12. [*M24*] (E. Cesàro.) If u and v are random positive integers, what is the average number of (positive) divisors they have in common? [*Hint:* See the identity in exercise 10(d), with $a_k = b_m = 1$.]

13. [*HM23*] Given that u and v are random *odd* positive integers, show that they are relatively prime with probability $8/\pi^2$.

14. [*M26*] What are the values of v_1 and v_2 when Algorithm X terminates?

▶ **15.** [*M22*] Design an algorithm to *divide u by v modulo m*, given positive integers u, v, and m, with v relatively prime to m. In other words, your algorithm should find w, in the range $0 \le w < m$, such that $u \equiv vw$ (modulo m).

16. [*21*] Use the text's method to find a general solution in integers to the following sets of equations:

a) $3x + 7y + 11z = 1$
$5x + 7y - 5z = 3$

b) $3x + 7y + 11z = 1$
$5x + 7y - 5z = -3$

▶ **17.** [*M24*] Show how Algorithm L can be extended (as Algorithm A was extended to Algorithm X) to obtain solutions of (15) when u and v are large.

18. [*M37*] Let u and v be odd integers, independently and uniformly distributed in the ranges $2^m \le u < 2^{m+1}$, $2^n \le v < 2^{n+1}$. What is the *exact* probability that a single "subtract and shift" cycle in Algorithm B, namely an operation that starts at step B6 and then stops after step B5 is finished, reduces u and v to the ranges $2^{m'} \le u < 2^{m'+1}$, $2^{n'} \le v < 2^{n'+1}$, as a function of m, n, m', and n'? (This exercise gives more accurate values for the transition probabilities than the text's model does.)

19. [*M24*] Complete the text's derivation of (38) by establishing (37).

20. [*M26*] Let $\lambda = \lfloor \lg \gcd(u, v) \rfloor$. Show that the lattice-point model gives $\lambda = 1$ with probability $\frac{1}{5}$, $\lambda = 2$ with probability $\frac{1}{10}$, $\lambda = 3$ with probability $\frac{1}{20}$, etc., plus correction terms that go rapidly to zero as u and v approach infinity; hence the average value of λ is approximately $\frac{4}{5}$. [*Hint:* Consider the relation between the probability of a path from (m, n) to $(k + 1, k + 1)$ and a corresponding path from $(m - k, n - k)$ to $(1, 1)$.]

21. [*HM26*] Let C_{mn} and D_{mn} be the average number of subtraction and shift cycles, respectively, in Algorithm B, when u and v are odd, $\lfloor \lg u \rfloor = m$, $\lfloor \lg v \rfloor = n$. Show that for fixed n, $C_{mn} = \frac{1}{2}m + O(1)$ and $D_{mn} = m + O(1)$ as $m \to \infty$.

22. [*23*] Prove Eq. (44).

23. [*M28*] Show that if $C_{mn} = \alpha m + \beta n + \gamma$ for some constants α, β, and γ, then

$$\sum_{1 \le n \le m \le N} (N - m)(N - n)2^{m+n-2} C_{mn} = 2^{2N}\left(\tfrac{11}{27}(\alpha + \beta)N + O(1)\right),$$

$$\sum_{1 \le n \le N} (N - n)^2 2^{2n-2} C_{nn} = 2^{2N}\left(\tfrac{5}{27}(\alpha + \beta)N + O(1)\right).$$

▶ **24.** [*M30*] If $v = 1$ but u is large, during Algorithm B, it may take fairly long for the algorithm to determine that $\gcd(u, v) = 1$. Perhaps it would be worthwhile to add a test at the beginning of step B5: "If $t = 1$, the algorithm terminates with 2^k as the answer." Explore the question of whether or not this would be an improvement when the algorithm deals with random inputs, by determining the average number of times that step B6 is executed with $u = 1$ or $v = 1$, using the lattice-point model.

▶ **25.** [*M26*] (R. P. Brent.) Let u_n and v_n be the values of u and v after n iterations of steps B3–B5; let $X_n = u_n/v_n$, and assume that $F_n(x)$ is the probability that $X_n \le x$, for $0 \le x < \infty$. (a) Express $F_{n+1}(x)$ in terms of $F_n(x)$, under the assumption that step B4 always branches to B3 with independent probability $\frac{1}{2}$. (b) Let $G_n(x) = F_n(x) + 1 - F_n(x^{-1})$ be the probability that $Y_n \le x$, for $0 \le x \le 1$, where $Y_n = \min(u_n, v_n)/\max(u_n, v_n)$. Express G_{n+1} in terms of G_n. (c) Express the distribution $H_n(x) =$ probability that $\max(u_{n+1}, v_{n+1})/\max(u_n, v_n) < x$ in terms of G_n.

26. [*M23*] What is the length of the longest path from (m, n) to $(0, 0)$ in the lattice-point model?

▶ **27.** [*M28*] Given $m \ge n \ge 1$, find values of u, v with $\lfloor \lg u \rfloor = m$, $\lfloor \lg v \rfloor = n$ such that Algorithm B requires $m + 1$ subtraction steps.

28. [*M37*] Prove that the subtraction step B6 of Algorithm B is never executed more than $1 + \lfloor \lg \max(u, v) \rfloor$ times.

29. [*M30*] Evaluate the determinant

$$\begin{vmatrix} \gcd(1,1) & \gcd(1,2) & \ldots & \gcd(1,n) \\ \gcd(2,1) & \gcd(2,2) & \ldots & \gcd(2,n) \\ \vdots & \vdots & & \vdots \\ \gcd(n,1) & \gcd(n,2) & \ldots & \gcd(n,n) \end{vmatrix}.$$

30. [*M25*] Show that Euclid's algorithm (Algorithm A) applied to two n-bit binary numbers requires $O(n^2)$ units of time, as $n \to \infty$. (The same upper bound obviously holds for Algorithm B.)

31. [*M22*] Use Euclid's algorithm to find a simple formula for $\gcd(2^m - 1, 2^n - 1)$ when m and n are nonnegative integers.

32. [*M43*] Can the upper bound $O(n^2)$ in exercise 30 be decreased, if another algorithm for calculating the greatest common divisor is used?

33. [*M46*] Analyze V. C. Harris's "binary Euclidean algorithm."

▶ **34.** [*M32*] (R. W. Gosper.) Demonstrate how to modify Algorithm B for large numbers, using ideas analogous to those in Algorithm L.

▶ **35.** [*M28*] (V. R. Pratt.) Extend Algorithm B to an Algorithm Y that is analogous to Algorithm X.

36. [*HM49*] Find a rigorous proof that Brent's model describes the asymptotic behavior of Algorithm B.

*4.5.3. Analysis of Euclid's Algorithm

The execution time of Euclid's algorithm depends on T, the number of times the division step A2 is performed. (See Algorithm 4.5.2A and Program 4.5.2A.) The quantity T is also an important factor in the running time of other algorithms, such as the evaluation of functions satisfying a reciprocity formula (see Section 3.3.3). We shall see in this section that the mathematical analysis of this quantity T is interesting and instructive.

Relation to continued fractions. Euclid's algorithm is intimately connected with *continued fractions*, which are expressions of the form

$$\cfrac{b_1}{a_1+\cfrac{b_2}{a_2+\cfrac{b_3}{\ddots\,\cfrac{}{a_{n-1}+\cfrac{b_n}{a_n}}}}} = b_1/(a_1+b_2/(a_2+b_3/(\cdots/(a_{n-1}+b_n/a_n)\ldots))). \tag{1}$$

Continued fractions have a beautiful theory that is the subject of several books. [See, for example, O. Perron, *Die Lehre von den Kettenbrüchen*, 3rd ed. (Stuttgart: Teubner, 1954), 2 vols.; A. Khinchin, *Continued Fractions*, tr. by Peter Wynn (Groningen: P. Noordhoff, 1963); H. S. Wall, *Analytic Theory of Continued Fractions* (New York: Van Nostrand, 1948); and see also J. Tropfke, *Geschichte der Elementar-Mathematik* **6** (Berlin: Gruyter, 1924), 74–84, for the early history of the subject.] It is necessary to limit ourselves to a comparatively brief treatment of the theory here, studying only those aspects that give us more insight into the behavior of Euclid's algorithm.

The continued fractions of primary interest to us are those in which all of the b's in (1) are equal to unity. For convenience in notation, let us define

$$/x_1, x_2, \ldots, x_n/ = 1/(x_1 + 1/(x_2 + 1/(\cdots(x_{n-1} + 1/x_n)\ldots))). \tag{2}$$

Thus, for example,

$$/x_1/ = \frac{1}{x_1}, \qquad /x_1, x_2/ = \frac{1}{x_1 + 1/x_2} = \frac{x_2}{x_1 x_2 + 1}. \tag{3}$$

If $n = 0$, the symbol $/x_1, \ldots, x_n/$ is taken to mean 0. Let us also define the polynomials $Q_n(x_1, x_2, \ldots, x_n)$ of n variables, for $n \geq 0$, by the rule

$$Q_n(x_1, x_2, \ldots, x_n) = \begin{cases} 1, & \text{if } n = 0; \\ x_1, & \text{if } n = 1; \\ x_1 Q_{n-1}(x_2, \ldots, x_n) + Q_{n-2}(x_3, \ldots, x_n), & \text{if } n > 1. \end{cases} \tag{4}$$

Thus $Q_2(x_1, x_2) = x_1 x_2 + 1, Q_3(x_1, x_2, x_3) = x_1 x_2 x_3 + x_1 + x_3$, etc. In general, as noted by L. Euler in the eighteenth century, $Q_n(x_1, x_2, \ldots, x_n)$ is the sum of all terms obtainable by starting with $x_1 x_2 \ldots x_n$ and deleting zero or more non-overlapping pairs of consecutive variables $x_j x_{j+1}$; there are F_{n+1} such terms. The polynomials defined in (4) are called "continuants."

The basic property of the Q-polynomials is that

$$/x_1, x_2, \ldots, x_n/ = Q_{n-1}(x_2, \ldots, x_n)/Q_n(x_1, x_2, \ldots, x_n), \qquad n \geq 1. \tag{5}$$

This can be proved by induction, since it implies that

$$x_0 + /x_1, \ldots, x_n/ = Q_{n+1}(x_0, x_1, \ldots, x_n)/Q_n(x_1, \ldots, x_n);$$

hence $/x_0, x_1, \ldots, x_n/$ is the reciprocal of the latter quantity.

The Q-polynomials are symmetrical in the sense that

$$Q_n(x_1, x_2, \ldots, x_n) = Q_n(x_n, \ldots, x_2, x_1). \tag{6}$$

This follows from Euler's observation above, and as a consequence we have

$$Q_n(x_1, \ldots, x_n) = x_n Q_{n-1}(x_1, \ldots, x_{n-1}) + Q_{n-2}(x_1, \ldots, x_{n-2}) \tag{7}$$

for $n > 1$. The Q-polynomials also satisfy the important identity

$$Q_n(x_1, \ldots, x_n) Q_n(x_2, \ldots, x_{n+1}) - Q_{n+1}(x_1, \ldots, x_{n+1}) Q_{n-1}(x_2, \ldots, x_n) = (-1)^n, \qquad n \geq 1. \tag{8}$$

(See exercise 4.) The latter equation in connection with (5) implies that

$$/x_1, \ldots, x_n/ = \frac{1}{q_0 q_1} - \frac{1}{q_1 q_2} + \frac{1}{q_2 q_3} - \cdots + \frac{(-1)^{n-1}}{q_{n-1} q_n}, \quad \text{where } q_k = Q_k(x_1, \ldots, x_k). \tag{9}$$

Thus the Q-polynomials are intimately related to continued fractions.

Every real number X in the range $0 \le X < 1$ has a *regular continued fraction* defined as follows: Let $X_0 = X$, and for all $n \ge 0$ such that $X_n \neq 0$ let

$$A_{n+1} = \lfloor 1/X_n \rfloor, \qquad X_{n+1} = 1/X_n - A_{n+1}. \tag{10}$$

If $X_n = 0$, the quantities A_{n+1} and X_{n+1} are not defined, and the regular continued fraction for X is $/A_1, \ldots, A_n/$. If $X_n \neq 0$, this definition guarantees that $0 \le X_{n+1} < 1$, so each of the A's is a positive integer. The definition (10) clearly implies that

$$X = X_0 = \cfrac{1}{A_1 + X_1} = \cfrac{1}{A_1 + 1/(A_2 + X_2)} = \cdots;$$

hence

$$X = /A_1, \ldots, A_{n-1}, A_n + X_n/ \tag{11}$$

for all $n \ge 1$, whenever X_n is defined. In particular, we have $X = /A_1, \ldots, A_n/$ when $X_n = 0$. If $X_n \neq 0$, the number X always lies *between* $/A_1, \ldots, A_n/$ and $/A_1, \ldots, A_n + 1/$, since by (7) the quantity $q_n = Q_n(A_1, \ldots, A_n + X_n)$ increases monotonically from $Q_n(A_1, \ldots, A_n)$ up to $Q_n(A_1, \ldots, A_n + 1)$ as X_n increases from 0 to 1, and by (9) the continued fraction increases or decreases when q_n increases, according as n is even or odd. In fact,

$$\begin{aligned}
|X - /A_1, \ldots, A_n/| &= |/A_1, \ldots, A_n + X_n/ - /A_1, \ldots, A_n/| \\
&= |/A_1, \ldots, A_n, 1/X_n/ - /A_1, \ldots, A_n/| \\
&= \left| \frac{Q_n(A_2, \ldots, A_n, 1/X_n)}{Q_{n+1}(A_1, \ldots, A_n, 1/X_n)} - \frac{Q_{n-1}(A_2, \ldots, A_n)}{Q_n(A_1, \ldots, A_n)} \right| \\
&= 1/(Q_n(A_1, \ldots, A_n) Q_{n+1}(A_1, \ldots, A_n, 1/X_n)) \\
&\le 1/(Q_n(A_1, \ldots, A_n) Q_{n+1}(A_1, \ldots, A_n, A_{n+1})) \qquad (12)
\end{aligned}$$

by (5), (7), (8), and (10). Therefore $/A_1, \ldots, A_n/$ is an extremely close approximation to X, unless n is small. If X is irrational, it is impossible to have $X_n = 0$ for any n, so the regular continued fraction expansion in this case is an *infinite continued fraction* $/A_1, A_2, A_3, \ldots/$. The value of an infinite continued fraction is defined to be

$$\lim_{n \to \infty} /A_1, A_2, \ldots, A_n/,$$

and from the inequality (12) it is clear that this limit equals X.

The regular continued fraction expansion of real numbers has several properties analogous to the representation of numbers in the decimal system. If we use the formulas above to compute the regular continued fraction expansions of

some familiar real numbers, we find, for example, that

$$\begin{aligned}
\tfrac{8}{29} &= /3,1,1,1,2/;\\
\sqrt{\tfrac{8}{29}} &= /1,1,9,2,2,3,2,2,9,1,2,1,9,2,2,3,2,2,9,1,2,1,9,2,2,3,2,2,9,1,\ldots/;\\
\sqrt[3]{2} &= 1+/3,1,5,1,1,4,1,1,8,1,14,1,10,2,1,4,12,2,3,2,1,3,4,1,1,2,14,3,\ldots/;\\
\pi &= 3+/7,15,1,292,1,1,1,2,1,3,1,14,2,1,1,2,2,2,2,1,84,2,1,1,15,3,13,\ldots/;\\
e &= 2+/1,2,1,1,4,1,1,6,1,1,8,1,1,10,1,1,12,1,1,14,1,1,16,1,1,18,1,\ldots/;\\
\gamma &= /1,1,2,1,2,1,4,3,13,5,1,1,8,1,2,4,1,1,40,1,11,3,7,1,7,1,1,5,\ldots/;\\
\phi &= 1+/1,1,1,1,1,1,1,1,1,\ldots/. \qquad (13)
\end{aligned}$$

The numbers $A_1, A_2, \ldots$ are called the *partial quotients* of X. Note the regular pattern that appears in the partial quotients for $\sqrt{8/29}$, ϕ, and e; the reasons for this behavior are discussed in exercises 12 and 16. There is no apparent pattern in the partial quotients for $\sqrt[3]{2}$, π, or γ.

It is interesting to note that the ancient Greeks' first definition of real numbers, once they had discovered the existence of irrationals, was essentially stated in terms of infinite continued fractions. (Later they adopted the suggestion of Eudoxus that $x = y$ should be defined instead as "$x < r$ if and only if $y < r$, for all rational r.") See O. Becker, *Quellen und Studien zur Geschichte Math., Astron., Physik* (B) **2** (1933), 311–333.

When X is a rational number, the regular continued fraction corresponds in a natural way to Euclid's algorithm. Let us assume that $X = v/u$, where $u > v \geq 0$. The regular continued fraction process starts with $X_0 = X$; let us define $U_0 = u$, $V_0 = v$. Assuming that $X_n = V_n/U_n \neq 0$, (10) becomes

$$\begin{aligned}
A_{n+1} &= \lfloor U_n/V_n \rfloor,\\
X_{n+1} &= U_n/V_n - A_{n+1} = (U_n \bmod V_n)/V_n.
\end{aligned} \qquad (14)$$

Therefore, if we define

$$U_{n+1} = V_n, \qquad V_{n+1} = U_n \bmod V_n, \qquad (15)$$

the condition $X_n = V_n/U_n$ holds throughout the process. Furthermore, (15) is precisely the transformation made on the variables u and v in Euclid's algorithm (see Algorithm 4.5.2A, step A2). For example, since $\frac{8}{29} = /3,1,1,1,2/$, we know that Euclid's algorithm applied to $u = 29$ and $v = 8$ will require exactly five division steps, and the quotients $\lfloor u/v \rfloor$ in step A2 will be successively 3, 1, 1, 1, and 2. Note that the last partial quotient A_n must be 2 or more when $X_n = 0$ and $n \geq 1$, since X_{n-1} is less than unity.

From this correspondence with Euclid's algorithm we can see that the regular continued fraction for X terminates at some step with $X_n = 0$ if and only if X is rational; for it is obvious that X_n cannot be zero if X is irrational, and, conversely, we know that Euclid's algorithm always terminates. If the partial quotients obtained during Euclid's algorithm are $A_1, A_2, \ldots, A_n$, then we have,

by (5),

$$\frac{v}{u} = \frac{Q_{n-1}(A_2, \ldots, A_n)}{Q_n(A_1, A_2, \ldots, A_n)}. \tag{16}$$

This formula holds also if Euclid's algorithm is applied for $u < v$, when $A_1 = 0$. Furthermore, because of (8), $Q_{n-1}(A_2, \ldots, A_n)$ and $Q_n(A_1, A_2, \ldots, A_n)$ are relatively prime, and the fraction on the right-hand side of (16) is in lowest terms; therefore

$$u = Q_n(A_1, A_2, \ldots, A_n)d, \qquad v = Q_{n-1}(A_2, \ldots, A_n)d, \tag{17}$$

where $d = \gcd(u, v)$.

The worst case. We can now apply these observations to determine the behavior of Euclid's algorithm in the "worst case," or in other words to give an upper bound on the number of division steps. The worst case occurs when the inputs are consecutive Fibonacci numbers:

Theorem F (G. Lamé, 1845). *For $n \geq 1$, let u and v be integers with $u > v > 0$ such that Euclid's algorithm applied to u and v requires exactly n division steps, and such that u is as small as possible satisfying these conditions. Then $u = F_{n+2}$ and $v = F_{n+1}$.*

Proof. By (17), we must have $u = Q_n(A_1, A_2, \ldots, A_n)d$, where A_1, A_2, $\ldots$, A_n, and d are positive integers and $A_n \geq 2$. Since Q_n is a polynomial with nonnegative coefficients, involving all of the variables, the minimum value is achieved only when $A_1 = 1$, $\ldots$, $A_{n-1} = 1$, $A_n = 2$, $d = 1$. Putting these values in (17) yields the desired result. ∎

(This theorem has the historical claim of being the first practical application of the Fibonacci sequence; since then many other applications of Fibonacci numbers to algorithms and to the study of algorithms have been discovered.)

As a consequence of Theorem F we have an important corollary:

Corollary L. *If $0 \leq u, v < N$, the number of division steps required when Algorithm 4.5.2A is applied to u and v is at most $\lceil \log_\phi(\sqrt{5}\,N) \rceil - 2$.*

Proof. By Theorem F, the maximum number of steps, n, occurs when $u = F_n$ and $v = F_{n+1}$, where n is as large as possible with $F_{n+1} < N$. (The first division step in this case merely interchanges u and v when $n > 1$.) Since $F_{n+1} < N$, we have $\phi^{n+1}/\sqrt{5} < N$ (see Eq. 1.2.8–15), so $n + 1 < \log_\phi(\sqrt{5}\,N)$. This completes the proof. ∎

Note that $\log_\phi(\sqrt{5}\,N)$ is approximately $2.078 \ln N + 1.672 \approx 4.785 \log_{10} N + 1.672$. See exercises 31 and 36 for extensions of Theorem F.

An approximate model. Now that we know the maximum number of division steps that can occur, let us attempt to find the *average* number. Let $T(m,n)$ be the number of division steps that occur when $u = m$ and $v = n$ are input to Euclid's algorithm. Thus

$$T(m,0) = 0; \qquad T(m,n) = 1 + T(n, m \bmod n) \qquad \text{if } n \geq 1. \tag{18}$$

Let T_n be the average number of division steps when $v = n$ and when u is chosen at random; since only the value of $u \bmod v$ affects the algorithm after the first division step, we may write

$$T_n = \frac{1}{n} \sum_{0 \leq k < n} T(k,n). \tag{19}$$

For example, $T(0,5) = 1$, $T(1,5) = 2$, $T(2,5) = 3$, $T(3,5) = 4$, $T(4,5) = 3$, so

$$T_5 = \tfrac{1}{5}(1+2+3+4+3) = 2\tfrac{3}{5}.$$

In order to estimate T_n for large n, let us first try an approximation suggested by R. W. Floyd: We might assume that, for $0 \leq k < n$, the value of n is essentially "random" modulo k, so that we can set

$$T_n \approx 1 + \frac{1}{n}(T_0 + T_1 + \cdots + T_{n-1}).$$

Then $T_n \approx S_n$, where the sequence $\langle S_n \rangle$ is the solution to the recurrence relation

$$S_0 = 0, \qquad S_n = 1 + \frac{1}{n}(S_0 + S_1 + \cdots + S_{n-1}), \qquad n \geq 1. \tag{20}$$

(This approximation is analogous to the "lattice-point model" used to investigate Algorithm B in Section 4.5.2.)

The recurrence (20) is readily solved by the use of generating functions. A more direct way to solve it, analogous to our solution of the lattice-point model, is by noting that

$$\begin{aligned} S_{n+1} &= 1 + \frac{1}{n+1}(S_0 + S_1 + \cdots + S_{n-1} + S_n) \\ &= 1 + \frac{1}{n+1}\big(n(S_n - 1) + S_n\big) = S_n + \frac{1}{n+1}; \end{aligned}$$

hence S_n is $1 + \frac{1}{2} + \cdots + \frac{1}{n} = H_n$, a harmonic number. The approximation $T_n \approx S_n$ now suggests that $T_n \approx \ln n + O(1)$.

Comparison of this approximation with tables of the true value of T_n show, however, that $\ln n$ is too large; T_n does not grow this fast. One way to account for the fact that this approximation is too pessimistic is to observe that the average

value of $n \bmod k$ is less than the average value of $\frac{1}{2}k$, in the range $1 \le k \le n$:

$$\begin{aligned}\frac{1}{n}\sum_{1\le k\le n}(n \bmod k) &= \frac{1}{n}\sum_{\substack{1\le q\le n\\ \lfloor n/(q+1)\rfloor < k \le \lfloor n/q\rfloor}}(n - qk)\\ &= n - \frac{1}{n}\sum_{1\le q\le n} q\left(\binom{\lfloor n/q\rfloor+1}{2} - \binom{\lfloor n/(q+1)\rfloor+1}{2}\right)\\ &= n - \frac{1}{n}\sum_{1\le q\le n}\binom{\lfloor n/q\rfloor+1}{2}\\ &= \left(1-\frac{\pi^2}{12}\right)n + O(\log n) \qquad (21)\end{aligned}$$

(cf. exercise 4.5.2–10(c)). This is only about $.1775n$, not $.25n$; so the value of $n \bmod k$ tends to be smaller than the above model predicts, and Euclid's algorithm works faster than we might expect.

A continuous model. The behavior of Euclid's algorithm with $v = N$ is essentially determined by the behavior of the regular continued fraction process when $X = 0/N,\ 1/N,\ \ldots,\ (N-1)/N$. Assuming that N is very large, we are led naturally to a study of regular continued fractions when X is a random real number uniformly distributed in $[0, 1)$. Therefore let us define the distribution function

$$F_n(x) = \text{probability that } X_n \le x, \qquad \text{for } 0 \le x \le 1, \qquad (22)$$

given a uniform distribution of $X = X_0$. By the definition of regular continued fractions, we have $F_0(x) = x$, and

$$\begin{aligned}F_{n+1}(x) &= \sum_{k\ge 1}\text{probability that } (k \le 1/X_n \le k + x)\\ &= \sum_{k\ge 1}\text{probability that } \big(1/(k+x) \le X_n \le 1/k\big)\\ &= \sum_{k\ge 1}\big(F_n(1/k) - F_n\big(1/(k+x)\big)\big). \qquad (23)\end{aligned}$$

If the distributions $F_0(x), F_1(x), \ldots$ defined by these formulas approach a limiting distribution $F_\infty(x) = F(x)$, we will have

$$F(x) = \sum_{k\ge 1}\big(F(1/k) - F\big(1/(k+x)\big)\big). \qquad (24)$$

One function that satisfies this relation is $F(x) = \log_b(1+x)$, for any base $b > 1$; see exercise 19. The further condition $F(1) = 1$ implies that we should take $b = 2$. Thus it is reasonable to make a guess that $F(x) = \lg(1+x)$, and that $F_n(x)$ approaches this behavior.

We might conjecture, for example, that $F(\frac{1}{2}) = \lg(\frac{3}{2}) \approx 0.58496$; let us see how close $F_n(\frac{1}{2})$ comes to this value for small n. We have $F_0(\frac{1}{2}) = \frac{1}{2}$, and

$$\begin{aligned}
F_1(\tfrac{1}{2}) &= \frac{1}{1} - \frac{1}{1+\frac{1}{2}} + \frac{1}{2} - \frac{1}{2+\frac{1}{2}} + \cdots \\
&= 2\left(\frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \frac{1}{5} + \cdots\right) = 2(1 - \ln 2) \approx 0.6137; \\
F_2(\tfrac{1}{2}) &= \sum_{m\ge 1} \frac{2}{m}\left(\frac{1}{2m+2} - \frac{1}{3m+2} + \frac{1}{4m+2} - \frac{1}{5m+2} + \cdots\right) \\
&= \sum_{m\ge 1} \frac{2}{m^2}\left(\frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \cdots\right) \\
&\quad - \sum_{m\ge 1} \frac{4}{m}\left(\frac{1}{2m(2m+2)} - \frac{1}{3m(3m+2)} + \cdots\right) \\
&= \frac{1}{3}\pi^2(1 - \ln 2) - \sum_{m\ge 1} \frac{4S_m}{m^2},
\end{aligned}$$

where $S_m = 1/(4m+4) - 1/(9m+6) + 1/(16m+8) - \cdots$. Using the values of H_x for fractional x found in Table 3 of Appendix B, we find that

$$S_1 = \tfrac{1}{12}, \qquad S_2 = \tfrac{3}{4} - \ln 2, \qquad S_3 = \tfrac{19}{20} - \pi/(2\sqrt{3}),$$

etc.; a numerical evaluation yields $F_2(\frac{1}{2}) \approx 0.5748$. Although $F_1(x) = H_x$, it is clear that $F_n(x)$ is difficult to calculate exactly when n is large.

The distributions $F_n(x)$ were first studied by K. F. Gauss, who thought of the problem in 1800. His notebook for that year lists various recurrence relations and gives a brief table of values, including the four-place value for $F_2(\frac{1}{2})$ that has just been mentioned. After performing these calculations, Gauss wrote, "*Tam complicatæ evadunt, ut nulla spes superesse videatur,*" i.e., "They come out so complicated that no hope appears to be left." Twelve years later, Gauss wrote a letter to Laplace in which he posed the problem as one he could not resolve to his satisfaction. He said, "I found by very simple reasoning that, for n infinite, $F_n(x) = \log(1+x)/\log 2$. But the efforts that I made since then in my inquiries to assign $F_n(x) - \log(1+x)/\log 2$ for very large but not infinite values of n were fruitless." He never published his "very simple reasoning," and it is not completely clear that he had found a rigorous proof. More than 100 years went by before a proof was finally published, by R. O. Kuz'min [*Atti del Congresso internazionale dei matematici* **6** (Bologna, 1928), 83–89], who showed that

$$F_n(x) = \lg(1+x) + O(e^{-A\sqrt{n}})$$

for some positive constant A. The error term was improved to $O(e^{-An})$ by Paul Lévy shortly afterward [*Bull. Soc. Math. de France* **57** (1929), 178–194]*; but

*An exposition of Lévy's interesting proof appeared in the first edition of this book.

Gauss's problem, namely to find the asymptotic behavior of $F_n(x) - \lg(1+x)$, was not really resolved until 1974, when Eduard Wirsing published a beautiful analysis of the situation [*Acta Arithmetica* **24** (1974), 507–528]. We shall study the simplest aspects of Wirsing's approach here, since his method is an instructive use of linear operators.

If G is any function of x defined for $0 \le x \le 1$, let SG be the function defined by

$$SG(x) = \sum_{k\ge 1}\left(G\Big(\frac{1}{k}\Big) - G\Big(\frac{1}{k+x}\Big)\right). \tag{25}$$

Thus, S is an operator that changes one function into another. In particular, by (23) we have $F_{n+1}(x) = SF_n(x)$, hence

$$F_n = S^n F_0. \tag{26}$$

(In this discussion F_n stands for a distribution function, *not* for a Fibonacci number.) Note that S is a "linear operator"; i.e., $S(cG) = c(SG)$ for all constants c, and $S(G_1 + G_2) = SG_1 + SG_2$.

Now if G has a bounded first derivative, we can differentiate (25) term by term to show that

$$(SG)'(x) = \sum_{k\ge 1} \frac{1}{(k+x)^2} G'\Big(\frac{1}{k+x}\Big); \tag{27}$$

hence SG also has a bounded first derivative. (Term-by-term differentiation of a convergent series is justified when the series of derivatives is uniformly convergent; cf. K. Knopp, *Theory and Application of Infinite series* (Glasgow: Blackie, 1951), §47.)

Let $H = SG$, and let $g(x) = (1+x)G'(x)$, $h(x) = (1+x)H'(x)$. It follows that

$$\begin{aligned} h(x) &= \sum_{k\ge 1} \frac{1+x}{(k+x)^2}\left(1+\frac{1}{k+x}\right)^{-1} g\Big(\frac{1}{k+x}\Big) \\ &= \sum_{k\ge 1}\left(\frac{k}{k+1+x} - \frac{k-1}{k+x}\right) g\Big(\frac{1}{k+x}\Big). \end{aligned}$$

In other words, $h = Tg$, where T is the linear operator defined by

$$Tg(x) = \sum_{k\ge 1}\left(\frac{k}{k+1+x} - \frac{k-1}{k+x}\right) g\Big(\frac{1}{k+x}\Big). \tag{28}$$

Continuing, we see that if g has a bounded first derivative, we can differentiate term by term to show that Tg does also:

$$\begin{aligned}(Tg)'(x) &= -\sum_{k\ge1}\biggl(\Bigl(\frac{k}{(k+1+x)^2}-\frac{k-1}{(k+x)^2}\Bigr)g\Bigl(\frac{1}{k+x}\Bigr)\\ &\qquad+\Bigl(\frac{k}{k+1+x}-\frac{k-1}{k+x}\Bigr)\frac{1}{(k+x)^2}g'\Bigl(\frac{1}{k+x}\Bigr)\biggr)\\ &= -\sum_{k\ge1}\biggl(\frac{k}{(k+1+x)^2}\Bigl(g\Bigl(\frac{1}{k+x}\Bigr)-g\Bigl(\frac{1}{k+1+x}\Bigr)\Bigr)\\ &\qquad+\frac{1+x}{(k+x)^3(k+1+x)}g'\Bigl(\frac{1}{k+x}\Bigr)\biggr).\end{aligned}$$

There is consequently a third linear operator, U, such that $(Tg)' = -U(g')$, namely

$$U\varphi(x) = \sum_{k\ge1}\biggl(\frac{k}{(k+1+x)^2}\int_{1/(k+1+x)}^{1/(k+x)}\varphi(t)\,dt + \frac{1+x}{(k+x)^3(k+1+x)}\varphi\Bigl(\frac{1}{k+x}\Bigr)\biggr). \tag{29}$$

What is the relevance of all this to our problem? Well, if we set

$$F_n(x) = \lg(1+x) + R_n\bigl(\lg(1+x)\bigr), \tag{30}$$

$$f_n(x) = (1+x)F_n'(x) = \frac{1}{\ln 2}\bigl(1+R_n'\bigl(\lg(1+x)\bigr)\bigr), \tag{31}$$

we have

$$f_n'(x) = R_n''\bigl(\lg(1+x)\bigr)/\bigl((\ln 2)^2(1+x)\bigr); \tag{32}$$

the effect of the $\lg(1+x)$ term disappears, after these transformations. Furthermore since $F_n = S^nF_0$ we have $f_n = T^nf_0$ and $f_n' = (-1)^nU^nf_0'$. Both F_n and f_n have bounded derivatives, by induction on n. Thus (32) becomes

$$(-1)^nR_n''\bigl(\lg(1+x)\bigr) = (1+x)(\ln 2)^2\,U^nf_0'(x). \tag{33}$$

Now $F_0(x) = x$, $f_0(x) = 1+x$, and $f_0'(x)$ is the constant function 1. We are going to show that the operator U^n takes the constant function into a function with very small values, hence $|R_n''(x)|$ must be very small for $0 \le x \le 1$. Finally we can clinch the argument by showing that $R_n(x)$ itself is small: Since we have $R_n(0) = R_n(1) = 1$, it follows from a well-known interpolation formula (cf. exercise 4.6.4–15 with $x_0 = 0$, $x_1 = x$, $x_2 = 1$) that

$$R_n(x) = -\frac{x(1-x)}{2}R_n''\bigl(\xi(x)\bigr) \tag{34}$$

for some function $\xi(x)$, where $0 \le \xi(x) \le 1$ when $0 \le x \le 1$.

Thus everything hinges on our being able to prove that U^n produces small function values, where U is the linear operator defined in (29). Note that U is a *positive* operator, in the sense that $U\varphi(x) \geq 0$ for all x if $\varphi(x) \geq 0$ for all x. It follows that U is order-preserving: If $\varphi_1(x) \leq \varphi_2(x)$ for all x then we have $U\varphi_1(x) \leq U\varphi_2(x)$ for all x.

One way to exploit this property is to find a function φ for which we can calculate $U\varphi$ exactly and to use constant multiples of this function to bound the ones that we are really interested in. First let us look for a function g such that Tg is easy to compute. If we consider functions defined for all $x \geq 0$, instead of only on $[0, 1]$, it is easy to remove the summation from (25) by observing that

$$SG(x+1)-SG(x) = G\Big(\frac{1}{1+x}\Big) - \lim_{k\to\infty} G\Big(\frac{1}{k+x}\Big) = G\Big(\frac{1}{1+x}\Big) - G(0) \tag{35}$$

when G is continuous. Since $T\big((1+x)G'\big) = (1+x)(SG)'$, it follows (see exercise 20) that

$$\frac{Tg(x)}{x+1} - \frac{Tg(x+1)}{x+2} = \Big(\frac{1}{x+1} - \frac{1}{x+2}\Big) g\Big(\frac{1}{1+x}\Big). \tag{36}$$

If we set $Tg(x) = 1/(x+1)$, we find that the corresponding value of $g(x)$ is $1 + x - 1/(1+x)$. Let $\varphi(x) = g'(x) = 1 + 1/(1+x)^2$, so that $U\varphi(x) = -(Tg)'(x) = 1/(1+x)^2$; this is the function φ we have been looking for.

For this choice of φ we have $2 \leq \varphi(x)/U\varphi(x) = (1+x)^2 + 1 \leq 5$ for $0 \leq x \leq 1$, hence

$$\tfrac{1}{5}\varphi \leq U\varphi \leq \tfrac{1}{2}\varphi.$$

By the positivity of U and φ we can apply U to this inequality again, obtaining $\frac{1}{25}\varphi \leq \frac{1}{5}U\varphi \leq U^2\varphi \leq \frac{1}{2}U\varphi \leq \frac{1}{4}\varphi$; and after $n-1$ applications we have

$$5^{-n}\varphi \leq U^n\varphi \leq 2^{-n}\varphi \tag{37}$$

for this particular φ. Let $\chi(x) = f_0'(x) = 1$ be the constant function; then for $0 \leq x \leq 1$ we have $\frac{5}{4}\chi \leq \varphi \leq 2\chi$, hence

$$\tfrac{5}{8}5^{-n}\chi \leq \tfrac{1}{2}5^{-n}\varphi \leq \tfrac{1}{2}U^n\varphi \leq U^n\chi \leq \tfrac{4}{5}U^n\varphi \leq \tfrac{4}{5}2^{-n}\varphi \leq \tfrac{8}{5}2^{-n}\chi.$$

It follows by (33) that

$$\tfrac{5}{8}(\ln 2)^2 5^{-n} \leq (-1)^n R_n''(x) \leq \tfrac{16}{5}(\ln 2)^2 2^{-n}, \qquad \text{for } 0 \leq x \leq 1;$$

hence by (30) and (34) we have proved the following result:

Theorem W. *The distribution $F_n(x)$ equals $\lg(1+x) + O(2^{-n})$ as $n \to \infty$. In fact, $F_n(x) - \lg(1+x)$ lies between $\frac{5}{16}(-1)^{n+1}5^{-n}\big(\ln(1+x)\big)\big(\ln 2/(1+x)\big)$ and $\frac{8}{5}(-1)^{n+1}2^{-n}\big(\ln(1+x)\big)\big(\ln 2/(1+x)\big)$, for $0 \leq x \leq 1$.* ∎

With a slightly different choice of φ, we can obtain tighter bounds (see exercise 21). In fact, Wirsing went much further in his paper, proving that

$$F_n(x) = \lg(1+x) + (-\lambda)^n \Psi(x) + O\big(x(1-x)(\lambda - 0.031)^n\big), \tag{38}$$

where

$$\begin{aligned} \lambda &= 0.30366\,30028\,98732\,65860\ldots \\ &= /3,3,2,2,3,13,1,174,1,1,1,2,2,2,1,1,1,2,2,1,\ldots/ \end{aligned} \tag{39}$$

is a fundamental constant (apparently unrelated to more familiar constants), and where Ψ is an interesting function that is analytic in the entire complex plane except for the negative real axis from -1 to $-\infty$. Wirsing's function satisfies $\Psi(0) = \Psi(1) = 0$, $\Psi'(0) < 0$, and $S\Psi = -\lambda\Psi$; thus by (35) it satisfies the identity

$$\Psi(z) - \Psi(z+1) = \frac{1}{\lambda}\Psi\Big(\frac{1}{1+z}\Big). \tag{40}$$

Furthermore, Wirsing demonstrated that

$$\Psi\Big(-\frac{u}{v} + \frac{i}{N}\Big) = c\lambda^{-n}\log N + O(1) \qquad \text{as } N \to \infty, \tag{41}$$

where c is a constant and $n = T(u,v)$ is the number of iterations when Euclid's algorithm is applied to the integers $u > v > 0$.

A complete solution to Gauss's problem was found a few years later by K. I. Babenko [*Doklady Akad. Nauk SSSR* **238** (1978), 1021–1024], who used powerful techniques of functional analysis to prove that

$$F_n(x) = \lg(1+x) + \sum_{j\ge 2} \lambda_j^n \Psi_j(x) \tag{42}$$

for all $0 \le x \le 1$, $n \ge 1$. Here $|\lambda_2| > |\lambda_3| \ge |\lambda_4| \ge \cdots$, and each $\Psi_j(z)$ is an analytic function in the complex plane except for a cut at $[-\infty, -1]$. The function Ψ_2 is Wirsing's Ψ, and $\lambda_2 = -\lambda$, while $\lambda_3 = 0.1009$, $\lambda_4 = -0.0408$, $\lambda_5 = -0.0355$, $\lambda_6 = 0.0128$. Babenko also established further properties of the eigenvalues λ_j, proving in particular that they are exponentially small as $j \to \infty$, and that the sum for $j \ge k$ in (42) is bounded by $(\pi^2/6)|\lambda_k|^{n-1}\min(x, 1-x)$. [Further information appears in a paper by Babenko and Îur'ev, *Doklady Akad. Nauk SSSR* **240** (1978), 1273–1276.]

From continuous to discrete. We have now derived results about the probability distributions for continued fractions when X is a real number uniformly distributed in the interval $[0,1)$. But a real number is rational with probability zero (almost all numbers are irrational), so these results do not apply directly to Euclid's algorithm. Before we can apply Theorem W to our problem, some technicalities must be overcome. Consider the following observation based on elementary measure theory:

Lemma M. *Let $I_1, I_2, \ldots, J_1, J_2, \ldots$ be pairwise disjoint intervals contained in the interval $[0,1)$, and let*

$$I = \bigcup_{k\ge 1} I_k, \qquad J = \bigcup_{k\ge 1} J_k, \qquad K = [0,1] \setminus (I \cup J).$$

Assume that K has measure zero. Let P_n be the set $\{0/n, 1/n, \ldots, (n-1)/n\}$. Then

$$\lim_{n\to\infty} \frac{\|I \cap P_n\|}{n} = \mu(I). \tag{43}$$

Here $\mu(I)$ is the Lebesgue measure of I, namely, $\sum_{k\ge 1} \text{length}(I_k)$; and $\|I \cap P_n\|$ denotes the number of elements in the set $I \cap P_n$.

Proof. Let $I_N = \bigcup_{1\le k\le N} I_k$ and $J_N = \bigcup_{1\le k\le N} J_k$. Given $\epsilon > 0$, find N large enough so that $\mu(I_N) + \mu(J_N) \ge 1 - \epsilon$, and let

$$K_N = K \cup \bigcup_{k>N} I_k \cup \bigcup_{k>N} J_k.$$

If I is an interval, having any of the forms (a,b) or $[a,b)$ or $(a,b]$ or $[a,b]$, it is clear that $\mu(I) = b - a$ and

$$n\mu(I) - 1 \le \|I \cap P_n\| \le n\mu(I) + 1.$$

Now let $r_n = \|I_N \cap P_n\|$, $s_n = \|J_N \cap P_n\|$, $t_n = \|K_N \cap P_n\|$; we have

$$\begin{aligned} r_n + s_n + t_n &= n; \\ n\mu(I_N) - N \le r_n &\le n\mu(I_N) + N; \\ n\mu(I_N) - N \le s_n &\le n\mu(J_N) + N. \end{aligned}$$

Hence

$$\begin{aligned} \mu(I) - \frac{N}{n} - \epsilon \le \mu(I_N) - \frac{N}{n} \le \frac{r_n}{n} &\le \frac{r_n + t_n}{n} \\ &= 1 - \frac{s_n}{n} \le 1 - \mu(J_N) + \frac{N}{n} \le \mu(I) + \frac{N}{n} + \epsilon. \end{aligned}$$

This holds for all n and for all ϵ; hence $\lim_{n\to\infty} r_n/n = \mu(I)$. ∎

Exercise 25 shows that Lemma M is not trivial, in the sense that some rather restrictive hypotheses are needed to prove (43).

Distribution of partial quotients. Now we can put Theorem W and Lemma M together to derive some solid facts about Euclid's algorithm.

Theorem E. *Let n and k be positive integers, and let $p_k(a,n)$ be the probability that the $(k+1)$st quotient A_{k+1} in Euclid's algorithm is equal to a, when $v = n$ and u is chosen at random. Then*

$$\lim_{n\to\infty} p_k(a,n) = F_k\Big(\frac{1}{a}\Big) - F_k\Big(\frac{1}{a+1}\Big),$$

where $F_k(x)$ is the distribution function (21).

Proof. The set $\mathcal{I}$ of all X in $[0,1)$ for which $A_{k+1} = a$ is a union of disjoint intervals, and so is the set $\mathcal{J}$ of all X for which $A_{k+1} \neq a$. Lemma M therefore applies, with $\mathcal{K}$ the set of all X for which A_{k+1} is undefined. Furthermore, $F_k(1/a) - F_k\big(1/(a+1)\big)$ is the probability that $1/(a+1) < X_k \le 1/a$, which is $\mu(\mathcal{I})$, the probability that $A_{k+1} = a$. ∎

As a consequence of Theorems E and W, we can say that a quotient equal to a occurs with the approximate probability

$$\lg(1 + 1/a) - \lg\big(1 + 1/(a+1)\big) = \lg\big((a+1)^2/((a+1)^2 - 1)\big).$$

Thus

a quotient of 1 occurs about $\lg(\frac{4}{3}) = 41.504$ percent of the time;
a quotient of 2 occurs about $\lg(\frac{9}{8}) = 16.992$ percent of the time;
a quotient of 3 occurs about $\lg(\frac{16}{15}) = 9.311$ percent of the time;
a quotient of 4 occurs about $\lg(\frac{25}{24}) = 5.890$ percent of the time.

Actually, if Euclid's algorithm produces the quotients $A_1, A_2, \ldots, A_t$, the nature of the proofs above will guarantee this behavior only for A_k when k is comparatively small with respect to t; the values $A_{t-1}, A_{t-2}, \ldots$ are not covered by this proof. But we can in fact show that the distribution of the last quotients $A_{t-1}, A_{t-2}, \ldots$ is essentially the same as the first.

For example, consider the regular continued fraction expansions for the set of all proper fractions whose denominator is 29:

$\frac{1}{29} = /\,29\,/$	$\frac{8}{29} = /\,3,1,1,1,2\,/$	$\frac{15}{29} = /\,1,1,14\,/$	$\frac{22}{29} = /\,1,3,7\,/$
$\frac{2}{29} = /\,14,2\,/$	$\frac{9}{29} = /\,3,4,2\,/$	$\frac{16}{29} = /\,1,1,4,3\,/$	$\frac{23}{29} = /\,1,3,1,5\,/$
$\frac{3}{29} = /\,9,1,2\,/$	$\frac{10}{29} = /\,2,1,9\,/$	$\frac{17}{29} = /\,1,1,2,2,2\,/$	$\frac{24}{29} = /\,1,4,1,4\,/$
$\frac{4}{29} = /\,7,4\,/$	$\frac{11}{29} = /\,2,1,1,1,3\,/$	$\frac{18}{29} = /\,1,1,1,1,1,3\,/$	$\frac{25}{29} = /\,1,6,4\,/$
$\frac{5}{29} = /\,5,1,4\,/$	$\frac{12}{29} = /\,2,2,2,2\,/$	$\frac{19}{29} = /\,1,1,1,9\,/$	$\frac{26}{29} = /\,1,8,1,2\,/$
$\frac{6}{29} = /\,4,1,5\,/$	$\frac{13}{29} = /\,2,4,3\,/$	$\frac{20}{29} = /\,1,2,4,2\,/$	$\frac{27}{29} = /\,1,13,2\,/$
$\frac{7}{29} = /\,4,7\,/$	$\frac{14}{29} = /\,2,14\,/$	$\frac{21}{29} = /\,1,2,1,1,1,2\,/$	$\frac{28}{29} = /\,1,28\,/$

Several things can be observed in this table.

a) As mentioned earlier, the last quotient is always 2 or more. Furthermore, we have the obvious identity

$$/x_1, \ldots, x_{n-1}, x_n + 1/ = /x_1, \ldots, x_{n-1}, x_n, 1/, \tag{44}$$

and this shows how partial fractions whose last quotient is unity are related to regular continued fractions.

b) The values in the right-hand columns have a simple relationship to the values in the left-hand columns; can the reader see the correspondence before reading any further? The relevant identity is

$$1 - /x_1, x_2, \ldots, x_n/ = /1, x_1 - 1, x_2, \ldots, x_n/; \tag{45}$$

see exercise 9.

c) There is symmetry between left and right in the first two columns: If $/A_1, A_2, \ldots, A_t/$ occurs, so does $/A_t, \ldots, A_2, A_1/$. This will always be the case (see exercise 26).

d) If we examine all of the quotients in the table, we find that there are 96 in all, of which $\frac{39}{96} = 40.6$ percent are equal to 1, $\frac{21}{96} = 21.9$ percent are equal to 2, $\frac{8}{96} = 8.3$ percent are equal to 3; this agrees reasonably well with the probabilities listed above.

The number of division steps. Let us now return to our original problem and investigate T_n, the average number of division steps when $v = n$. (See Eq. (19).) Here are some sample values of T_n:

$n =$	95	96	97	98	99	100	101	102	103	104	105
$T_n =$	5.0	4.4	5.3	4.8	4.7	4.6	5.3	4.6	5.3	4.7	4.6
$n =$	996	997	998	999	1000	1001	$\cdots$	9999	10000	10001	
$T_n =$	6.5	7.3	7.0	6.8	6.4	6.7	$\cdots$	8.6	8.3	9.1	
$n =$	49999	50000	50001	$\cdots$	99999	100000	100001				
$T_n =$	10.6	9.7	10.0	$\cdots$	10.7	10.3	11.0				

Note the somewhat erratic behavior; T_n tends to be higher than its neighbors when n is prime, and it is correspondingly lower when n has many divisors. (In this list, 97, 101, 103, 997, and 49999 are primes; $10001 = 73 \cdot 137$, $50001 = 3 \cdot 7 \cdot 2381$, $99999 = 3 \cdot 3 \cdot 41 \cdot 271$, and $100001 = 11 \cdot 9091$.) It is not difficult to understand why this happens: if $\gcd(u, v) = d$, Euclid's algorithm applied to u and v behaves essentially the same as if it were applied to u/d and v/d. Therefore, when $v = n$ has several divisors, there are many choices of u for which n behaves as if it were smaller.

Accordingly let us consider *another* quantity, τ_n, which is the average number of division steps when $v = n$ and when u is *relatively prime* to n. Thus

$$\tau_n = \frac{1}{\varphi(n)} \sum_{\substack{0 \le m < n \\ \gcd(m,n)=1}} T(m, n). \tag{46}$$

It follows that

$$T_n = \frac{1}{n}\sum_{d\backslash n} \varphi(d)\tau_d. \tag{47}$$

Here is a table of τ_n for the same values of n considered above:

$n =$	95	96	97	98	99	100	101	102	103	104	105
$\tau_n =$	5.4	5.3	5.3	5.6	5.2	5.2	5.4	5.3	5.4	5.3	5.6
$n =$	996	997	998	999	1000	1001	$\cdots$	9999	10000	10001	
$\tau_n =$	7.2	7.3	7.3	7.3	7.3	7.4	$\cdots$	9.21	9.21	9.22	
$n =$	49999	50000	50001	$\cdots$	99999	100000	100001				
$\tau_n =$	10.58	10.57	10.59	$\cdots$	11.170	11.172	11.172				

Clearly τ_n is much more well-behaved than T_n, and it should be more susceptible to analysis. Inspection of a table of τ_n for small n reveals some curious anomalies; for example, $\tau_{50} = \tau_{100}$ and $\tau_{60} = \tau_{120}$. But as n grows, the values of τ_n behave quite regularly indeed, as the above table indicates, and they show no significant relation to the factorization properties of n. If the reader will plot the values of τ_n versus $\ln n$ on graph paper, for the values of τ_n given above, he will see that the values lie very nearly on a straight line, and that the formula

$$\tau_n \approx 0.843 \ln n + 1.47 \tag{48}$$

is a very good approximation.

We can account for this behavior if we study the regular continued fraction process a little further. Note that in Euclid's algorithm as expressed in (15) we have

$$\frac{V_0}{U_0}\frac{V_1}{U_1}\cdots\frac{V_{t-1}}{U_{t-1}} = \frac{V_{t-1}}{U_0},$$

since $U_{k+1} = V_k$; therefore if $U = U_0$ and $V = V_0$ are relatively prime, and if there are t division steps, we have

$$X_0X_1\ldots X_{t-1} = 1/U.$$

Setting $U = N$ and $V = m < N$, we find that

$$\ln X_0 + \ln X_1 + \cdots + \ln X_{t-1} = -\ln N. \tag{49}$$

We know the approximate distribution of X_0, X_1, X_2, $\ldots$, so we can use this equation to estimate

$$t = T(N, m) = T(m, N) - 1.$$

Returning to the formulas preceding Theorem W, we find that the average value of $\ln X_n$, when X_0 is a real number uniformly distributed in $[0, 1)$, is

$$\int_0^1 \ln x\, F_n'(x)\, dx = \int_0^1 \ln x\, f_n(x)\, dx/(1+x), \tag{50}$$

where $f_n(x)$ is defined in (31). Now

$$f_n(x) = \frac{1}{\ln 2} + O(2^{-n}), \tag{51}$$

using the facts we have derived earlier (see exercise 23); hence the average value of $\ln X_n$ is very well approximated by

$$\begin{aligned}
\frac{1}{\ln 2}\int_0^1 \frac{\ln x}{1+x}\,dx &= -\frac{1}{\ln 2}\int_0^\infty \frac{ue^{-u}}{1+e^{-u}}\,du \\
&= -\frac{1}{\ln 2}\sum_{k\ge 1}(-1)^{k+1}\int_0^\infty ue^{-ku}\,du \\
&= -\frac{1}{\ln 2}\left(1-\frac{1}{4}+\frac{1}{9}-\frac{1}{16}+\frac{1}{25}-\cdots\right) \\
&= -\frac{1}{\ln 2}\left(1+\frac{1}{4}+\frac{1}{9}+\cdots-2\left(\frac{1}{4}+\frac{1}{16}+\frac{1}{36}+\cdots\right)\right) \\
&= -\frac{1}{2\ln 2}\left(1+\frac{1}{4}+\frac{1}{9}+\cdots\right) \\
&= -\pi^2/(12\ln 2).
\end{aligned}$$

By (49) we therefore expect to have the approximate formula

$$-t\pi^2/(12\ln 2) \approx -\ln N;$$

that is, t should be approximately equal to $\big((12\ln 2)/\pi^2\big)\ln N$. This constant $(12\ln 2)/\pi^2 = 0.842765913\ldots$ agrees perfectly with the empirical formula (48) obtained earlier, so we have good reason to believe that the formula

$$\tau_n \approx \frac{12\ln 2}{\pi^2}\ln n + 1.47 \tag{52}$$

indicates the true asymptotic behavior of τ_n as $n \to \infty$.

If we assume that (52) is valid, we obtain the formula

$$T_n \approx \frac{12\ln 2}{\pi^2}\left(\ln n - \sum_{d\backslash n}\Lambda(d)/d\right) + 1.47, \tag{53}$$

where $\Lambda(d)$ is *von Mangoldt's function* defined by the rules

$$\Lambda(n) = \begin{cases} \ln p, & \text{if } n = p^r \text{ for } p \text{ prime and } r \ge 1; \\ 0, & \text{otherwise.} \end{cases} \tag{54}$$

For example,

$$\begin{aligned} T_{100} &\approx \frac{12\ln 2}{\pi^2}\left(\ln 100 - \frac{\ln 2}{2} - \frac{\ln 2}{4} - \frac{\ln 2}{5} - \frac{\ln 5}{25}\right) + 1.47 \\ &\approx (0.843)(4.605 - 0.347 - 0.173 - 0.322 - 0.064) + 1.47 \\ &\approx 4.59; \end{aligned}$$

the exact value of T_{100} is 4.56.

We can also estimate the average number of division steps when u and v are both uniformly distributed between 1 and N, by calculating

$$\frac{1}{N}\sum_{1\le n\le N} T_n. \tag{55}$$

Assuming formula (53), exercise 27 shows that this sum has the form

$$\frac{12\ln 2}{\pi^2}\ln N + O(1), \tag{56}$$

and empirical calculations with the same numbers used to derive Eq. 4.5.2–45 show good agreement with the formula

$$\frac{12\ln 2}{\pi^2}\ln N + 0.06. \tag{57}$$

Of course we have not yet *proved* anything about T_n and τ_n in general; so far we have only been considering plausible reasons why the above formulas ought to hold. Fortunately it is now possible to supply rigorous proofs, based on a careful analysis by several mathematicians.

The leading coefficient $(12\ln 2)/\pi^2$ in the above formulas was established first, in independent studies by John D. Dixon and Hans A. Heilbronn. Dixon [*J. Number Theory* **2** (1970), 414–422] developed the theory of the $F_n(x)$ distributions to show that individual partial quotients are essentially independent of each other in an appropriate sense, and proved that for all positive ϵ we have $\left|T(m,n) - \left((12\ln 2)/\pi^2\right)\ln n\right| < (\ln n)^{(1/2)+\epsilon}$ except for $\exp\left(-c(\epsilon)(\log N)^{\epsilon/2}\right)N^2$ values of m and n in the range $1 \le m < n \le N$, where $c(\epsilon) > 0$. Heilbronn's approach was completely different, working entirely with integers instead of continuous variables. His idea, which is presented in slightly modified form in exercises 33 and 34, is based on the fact that τ_n can be related to the number of ways to represent n in a certain manner. Furthermore, his paper [*Number Theory and Analysis*, ed. by Paul Turán (New York: Plenum, 1969), 87–96] shows that the distribution of individual partial quotients 1, 2, ... that we have discussed above actually applies to the entire collection of partial quotients belonging to the fractions having a given denominator; this is a sharper form of Theorem E. A still sharper result was obtained several years later by J. W. Porter [*Mathematika* **22**

(1975), 20–28], who established that

$$\tau_n = \frac{12\ln 2}{\pi^2}\ln n + C + O(n^{-1/6+\epsilon}), \tag{58}$$

where [illegible] is the constant

[illegible]

see D. E. Knuth, *Computers and Math. with Applic.* **2** (1976), 137–139. Thus the conjecture (48) is fully proved.

The average running time for Euclid's algorithm on multiple-precision integers, using classical algorithms for arithmetic, was shown to be of order

$$\big(1 + \log(\max(u,v)/\gcd(u,v))\big)\log\min(u,v)$$

by G. E. Collins, in *SIAM J. Computing* **3** (1974), 1–10.

Summary. We have found that the worst case of Euclid's algorithm occurs when its inputs u and v are consecutive Fibonacci numbers (Theorem F); the number of division steps when $v = n$ will never exceed $\lceil 4.8\log_{10} N - 0.32\rceil$. We have determined the frequency of the values of various partial quotients, showing, for example, that the division step finds $\lfloor u/v\rfloor = 1$ about 41 percent of the time (Theorem E). And, finally, the theorems of Heilbronn and Porter prove that the average number T_n of division steps when $v = n$ is approximately

$$\big((12\ln 2)/\pi^2\big)\ln n \approx 1.9405\log_{10} n,$$

minus a correction term based on the divisors of n as shown in Eq. (53).

EXERCISES

▶ **1.** [*20*] Since the quotient $\lfloor u/v\rfloor$ is equal to unity over 40 percent of the time in Algorithm 4.5.2A, it may be advantageous on some computers to make a test for this case and to avoid the division when the quotient is unity. Is the following MIX program for Euclid's algorithm more efficient than Program 4.5.2A?

```
    LDX  U     rX ← u.
    JMP  2F
1H  STX  V     v ← rX.
    SUB  V     rA ← u − v.
    CMPA V
    SRAX 5     rAX ← rA.
    JL   2F    Is u − v < v?
    DIV  V     rX ← rAX mod v.
2H  LDA  V     rA ← v.
    JXNZ 1B    Done if rX = 0.  ∎
```

2. [*M21*] Evaluate the matrix product

$$\begin{pmatrix} x_1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} x_2 & 1 \\ 1 & 0 \end{pmatrix}\cdots\begin{pmatrix} x_n & 1 \\ 1 & 0 \end{pmatrix}.$$

3. [*M21*] What is the value of

$$\det\begin{pmatrix} x_1 & 1 & 0 & \dots & 0 \\ -1 & x_2 & 1 & & 0 \\ 0 & -1 & x_3 & 1 & \vdots \\ \vdots & & -1 & & \\ 0 & 0 & \dots & -1 & x_n \end{pmatrix}?$$

4. [*M20*] Prove Eq. (8).

5. [*HM25*] Let x_1, x_2, ... be a sequence of real numbers that are each greater than some positive real number ϵ. Prove that the infinite continued fraction $/x_1, x_2, \dots/ = \lim_{n\to\infty} /x_1, \dots, x_n/$ exists. Show also that $/x_1, x_2, \dots/$ need not exist if we assume only that $x_j > 0$ for all j.

6. [*M23*] Prove that the regular continued fraction expansion of a number is *unique* in the following sense: If B_1, B_2, ... are positive integers, then the infinite continued fraction $/B_1, B_2, \dots/$ is an irrational number X between 0 and 1 whose regular continued fraction has $A_n = B_n$ for all $n \geq 1$; and if B_1, ..., B_m are positive integers with $B_m > 1$, then the regular continued fraction for $X = /B_1, \dots, B_m/$ has $A_n = B_n$ for $1 \leq n \leq m$.

7. [*M26*] Find all permutations $p(1)p(2)\dots p(n)$ of the integers $\{1, 2, \dots, n\}$ such that $Q_n(x_1, x_2, \dots, x_n) = Q_n(x_{p(1)}, x_{p(2)}, \dots, x_{p(n)})$ holds for all x_1, x_2, ..., x_n.

8. [*M20*] Show that $-1/X_n = /A_n, \dots, A_1, -X/$, whenever X_n is defined, in the regular continued fraction process.

9. [*M21*] Show that continued fractions satisfy the following identities:

a) $/x_1, \dots, x_n/ = /x_1, \dots, x_k + /x_{k+1}, \dots, x_n//, \qquad 1 \leq k \leq n;$

b) $/0, x_1, x_2, \dots, x_n/ = x_1 + /x_2, \dots, x_n/, \qquad n \geq 1;$

c) $/x_1, \dots, x_{k-1}, x_k, 0, x_{k+1}, x_{k+2}, \dots, x_n/$
$= /x_1, \dots, x_{k-1}, x_k + x_{k+1}, x_{k+2}, \dots, x_n/, \qquad 1 \leq k < n;$

d) $1 - /x_1, x_2, \dots, x_n/ = /1, x_1 - 1, x_2, \dots, x_n/, \qquad n \geq 1.$

10. [*M28*] By the result of exercise 6, every irrational real number X has a unique regular continued fraction representation of the form

$$X = A_0 + /A_1, A_2, A_3, \dots/,$$

where A_0 is an integer and A_1, A_2, A_3, ... are positive integers. Show that if X has this representation then the regular continued fraction for $1/X$ is

$$1/X = B_0 + /B_1, \dots, B_m, A_5, A_6, \dots/$$

for suitable integers B_0, B_1, ..., B_m. (The case $A_0 < 0$ is, of course, the most interesting.) Explain how to determine the B's in terms of A_0, A_1, A_2, A_3, and A_4.

11. [*M30*] (J. Lagrange.) Let $X = A_0 + /A_1, A_2, \ldots/$, $Y = B_0 + /B_1, B_2, \ldots/$ be the regular continued fraction representations of two real numbers X and Y, in the sense of exercise 10. Show that these representations [illegible] in the sense that $A_{m+k} = B_{n+k}$ for some m and n and for all $k \geq 0$, if and only if we have $X = (qY + r)/(sY + t)$ for some integers q, r, s, t with $|qt - rs| = 1$. (This theorem is the analog, for continued fraction representations, of the simple result that the representations of X and Y in the decimal system eventually agree if and only if [illegible] for some integers q, r, and s.)

▶ **12.** [*M30*] A *quadratic irrationality* is a number of the form $(\sqrt{D} - U)/V$, where D, U, and V are integers, $D > 0$, $V \neq 0$, and D is not a perfect square. We may assume without loss of generality that V is a divisor of $D - U^2$, for otherwise the number may be rewritten as $(\sqrt{DV^2} - U|V|)/V|V|$.

a) Prove that the regular continued fraction expansion (in the sense of exercise 10) of a quadratic irrationality $X = (\sqrt{D} - U)/V$ is obtained by the following formulas:

$$V_0 = V, \qquad A_0 = \lfloor X \rfloor, \qquad U_0 = U + A_0 V;$$
$$V_{n+1} = (D - U_n^2)/V_n, \qquad A_{n+1} = \lfloor (\sqrt{D} + U_n)/V_{n+1} \rfloor,$$
$$U_{n+1} = A_{n+1} V_{n+1} - U_n.$$

[*Note:* An algorithm based on this process has many applications to the solution of quadratic equations in integers; see, for example, H. Davenport, *The Higher Arithmetic* (London: Hutchinson, 1952); W. J. LeVeque, *Topics in Number Theory* (Reading, Mass.: Addison–Wesley, 1956); and see also Section 4.5.4. By exercise 1.2.4–35, we have $A_{n+1} = \lfloor (\lfloor \sqrt{D} \rfloor + U_n)/V_{n+1} \rfloor$ when $V_{n+1} > 0$, and $A_{n+1} = \lfloor (\lfloor \sqrt{D} \rfloor + 1 + U_n)/V_{n+1} \rfloor$ when $V_{n+1} < 0$; hence such an algorithm need only work with the positive integer $\lfloor \sqrt{D} \rfloor$.]

b) Prove that $0 < U_n < \sqrt{D}$, $0 < V_n < 2\sqrt{D}$, for all $n > N$, where N is some integer depending on X; hence the regular continued fraction representation of every quadratic irrationality is eventually periodic. [*Hint:* Show that $(-\sqrt{D} - U)/V = A_0 + /A_1, \ldots, A_n, -V_n/(\sqrt{D} + U_n)/$, and use Eq. (5) to prove that $(\sqrt{D} + U_n)/V_n$ is positive when n is large.]

c) Letting $p_n = Q_{n+1}(A_0, A_1, \ldots, A_n)$ and $q_n = Q_n(A_1, \ldots, A_n)$, prove the identity $Vp_n^2 + 2Up_nq_n + ((U^2 - D)/V)q_n^2 = (-1)^{n+1} V_{n+1}$.

d) Prove that the regular continued fraction representation of an irrational number X is eventually periodic if and only if X is a quadratic irrationality. (This is the continued fraction analog of the fact that the decimal expansion of a real number X is eventually periodic if and only if X is rational.)

13. [*M40*] (J. Lagrange, 1797.) Let $f(x) = a_n x^n + \cdots + a_0$, $a_n > 0$, be a polynomial with integer coefficients, having no rational roots, and having exactly one real root $\xi > 1$. Design a computer program to find the first thousand or so partial quotients of ξ, using the following algorithm (which essentially involves only addition):

L1. Set $A \leftarrow 1$.

L2. For $k = 0, 1, \ldots, n - 1$ (in this order) and for $j = n - 1, \ldots, k$ (in this order), set $a_j \leftarrow a_{j+1} + a_j$. (This step replaces $f(x)$ by $g(x) = f(x + 1)$, a polynomial whose roots are one less than those of f.)

L3. If $a_n + a_{n-1} + \cdots + a_0 < 0$, set $A \leftarrow A + 1$ and return to L2.

L4. Output A (which is the value of the next partial quotient). Replace the coefficients $(a_n, a_{n-1}, \ldots, a_0)$ by $(-a_0, -a_1, \ldots, -a_n)$ and return to L1. (This step replaces $f(x)$ by a polynomial whose roots are reciprocals of those of f.)

For example, starting with $f(x) = x^3 - 2$, the algorithm will output "1" (changing $f(x)$ to $x^3 - 3x^2 - 3x - 1$); then "3" (changing $f(x)$ to $10x^3 - 6x^2 - 6x - 1$); etc.

14. [*M22*] (A. Hurwitz, 1891.) Show that the following rules make it possible to find the regular continued fraction expansion of $2X$, given the partial quotients of X:

$$2/\,2a, b, c, \ldots/ = /\,a, 2b + 2/c, \ldots//;$$
$$2/\,2a + 1, b, c, \ldots/ = /\,a, 1, 1 + 2/b - 1, c, \ldots//.$$

Use this idea to find the regular continued fraction expansion of $\frac{1}{2}e$, given the expansion of e in (13).

▶ **15.** [*M31*] (R. W. Gosper.) Generalizing exercise 14, design an algorithm that computes the continued fraction $X_0 + /X_1, X_2, \ldots/$ for $(ax + b)/(cx + d)$, given the continued fraction $x_0 + /x_1, x_2, \ldots/$ for x, and given integers a, b, c, d with $ad \neq bc$. Make your algorithm an "on-line coroutine" that outputs as many X_k as possible before inputting each x_j. Demonstrate how your algorithm computes $(97x + 39)/(-62x - 25)$ when $x = -1 + /5, 1, 1, 1, 2, 1, 2/$.

16. [*HM30*] (L. Euler, 1731.) Let $f_0(z) = (e^z - e^{-z})/(e^z + e^{-z}) = \tanh z$, and let $f_{n+1}(z) = 1/f_n(z) - (2n + 1)/z$. Prove that, for all n, $f_n(z)$ is an analytic function of the complex variable z in a neighborhood of the origin, and it satisfies the differential equation $f_n'(z) = 1 - f_n(z)^2 - 2nf_n(z)/z$. Use this fact to prove that

$$\tanh z = /z^{-1}, 3z^{-1}, 5z^{-1}, 7z^{-1}, \ldots/.$$

Then apply Hurwitz's rule (exercise 14) to prove that

$$e^{-1/n} = /\overline{1, (2m + 1)n - 1, 1}/, \qquad m \geq 0.$$

(This notation denotes the infinite continued fraction $/1,\ n - 1,\ 1,\ 1,\ 3n - 1,\ 1,\ 1,\ 5n - 1,\ 1,\ \ldots/$.) Also find the regular continued fraction expansion of $e^{-2/n}$ when $n > 0$ is odd.

▶ **17.** [*M23*] (a) Prove that $/x_1, -x_2/ = /x_1 - 1, 1, x_2 - 1/$. (b) Generalize this identity, obtaining a formula for $/x_1, -x_2, x_3, -x_4, \ldots, x_{2n-1}, -x_{2n}/$ in which all partial quotients are positive integers when the x's are large positive integers. (c) The result of exercise 16 implies that $\tan 1 = /1, -3, 5, -7, \ldots/$. Find the regular continued fraction expansion of $\tan 1$.

18. [*M40*] Develop a computer program to find as many partial quotients of x as possible, when x is a real number given with high precision. Use your program to calculate the first one thousand or so partial quotients of Euler's constant γ, based on D. W. Sweeney's 3566-place value [*Math. Comp.* **17** (1963), 170–178]. (According to the theory in the text, we expect to get about 0.97 partial quotients per decimal digit. Cf. Algorithm 4.5.2L and the article by J. W. Wrench, Jr. and D. Shanks, *Math. Comp.* **20** (1966), 444–447.)

19. [*M20*] Prove that $F(x) = \log_b(1 + x)$ satisfies Eq. (24).

20. [*HM20*] Derive (36) from (35).

21. [*HM29*] (E. Wirsing.) The bounds (37) were obtained for a function φ corresponding to g with $Tg(x) = 1/(x+1)$. Show that the function corresponding to $Tg(x) = 1/(x+c)$ yields better bounds, when $c > 0$ is an appropriate constant.

22. [*HM46*] (K. I. Babenko.) Develop efficient means to calculate accurate approximations to the quantities λ_j and $\Psi_j(x)$ in (42), for small $j \geq 3$ and for $0 \leq x \leq 1$.

23. [*HM23*] Prove (51), using results from the proof of Theorem W.

24. [*M22*] What is the average value of a partial quotient A_n in the regular continued fraction expansion of a random real number?

25. [*HM25*] Find an example of a set $I = I_1 \cup I_2 \cup I_3 \cup \cdots \subseteq [0,1]$, where the I's are disjoint intervals, for which (43) does not hold.

26. [*M23*] Show that if the numbers $\{1/n, 2/n, \ldots, \lfloor n/2 \rfloor / n\}$ are expressed as regular continued fractions, the result is symmetric between left and right, in the sense that $/A_t, \ldots, A_2, A_1/$ appears whenever $/A_1, A_2, \ldots, A_t/$ does.

27. [*M21*] Derive (53) from (47) and (52).

28. [*M23*] Prove the following identities involving the three number-theoretic functions $\varphi(n)$, $\mu(n)$, $\Lambda(n)$:

a) $\displaystyle\sum_{d\backslash n} \mu(d) = \delta_{n1}$. b) $\displaystyle\ln n = \sum_{d\backslash n} \Lambda(d)$, $\displaystyle n = \sum_{d\backslash n} \varphi(d)$.

c) $\displaystyle\Lambda(n) = \sum_{d\backslash n} \mu\Big(\frac{n}{d}\Big) \ln d$, $\displaystyle\varphi(n) = \sum_{d\backslash n} \mu\Big(\frac{n}{d}\Big) d$.

29. [*M23*] Assuming that T_n is given by (53), show that (55) equals (56).

▶ **30.** [*HM32*] The following variant of Euclid's algorithm is often suggested: Instead of replacing v by $u \bmod v$ during the division step, replace it by $|(u \bmod v) - v|$ if $u \bmod v > \frac{1}{2}v$. Thus, for example, if $u = 26$ and $v = 7$, we have $\gcd(26,7) = \gcd(-2,7) = \gcd(7,2)$; -2 is the *remainder of smallest magnitude* when multiples of 7 are subtracted from 26. Compare this procedure with Euclid's algorithm; estimate the number of division steps this method saves, on the average.

▶ **31.** [*M35*] Find the "worst case" of the modification of Euclid's algorithm suggested in exercise 30; what are the smallest inputs $u > v > 0$ that require n division steps?

32. [*20*] (a) A Morse code sequence of length n is a string of r dots and s dashes, where $r + 2s = n$. For example, the Morse code sequences of length 4 are

$$\cdot\cdot\cdot\cdot,\quad \cdot\cdot{-},\quad \cdot{-}\cdot,\quad {-}\cdot\cdot,\quad {-}{-}.$$

Noting that the continuant $Q_4(x_1, x_2, x_3, x_4)$ is $x_1x_2x_3x_4 + x_1x_2 + x_1x_4 + x_3x_4 + 1$, find and prove a simple relation between $Q_n(x_1, \ldots, x_n)$ and Morse code sequences of length n. (b) (L. Euler, *Novi Comm. Acad. Sci. Pet.* **9** (1762), 53–69.) Prove that

$$Q_{m+n}(x_1, \ldots, x_{m+n}) = Q_m(x_1, \ldots, x_m)Q_n(x_{m+1}, \ldots, x_{m+n}) + Q_{m-1}(x_1, \ldots, x_{m-1})Q_{n-1}(x_{m+2}, \ldots, x_{m+n}).$$

33. [*M32*] Let $h(n)$ be the number of representations of n in the form

$$n = xx' + yy', \quad x > y > 0, \quad x' > y' > 0, \quad \gcd(x, y) = 1, \quad \text{integer } x, x', y, y'.$$

(a) Show that if the conditions are relaxed to allow $x' = y'$, the number of representations is $h(n) + \lfloor (n-1)/2 \rfloor$. (b) Show that for fixed $y > 0$ and $0 < t \leq y$, where $\gcd(t, y) = 1$, and for each fixed x' in the range $0 < x' < n/(y+t)$ such that $x't \equiv n$ (modulo y), there is exactly one representation of n satisfying the restrictions of (a) and the condition $x \equiv t$ (modulo y). (c) Consequently

$$h(n) = \sum \left\lceil \left(\frac{n}{y+t} - t' \right) \frac{1}{y} \right\rceil - \lfloor (n-1)/2 \rfloor,$$

where the sum is over all positive integers y, t, t' such that $\gcd(t, y) = 1$, $t \leq y$, $t' \leq y$, $tt' \equiv n$ (modulo y). (d) Show that each of the $h(n)$ representations can be expressed uniquely in the form

$$\begin{aligned} x &= Q_m(x_1, \ldots, x_m), & y &= Q_{m-1}(x_1, \ldots, x_{m-1}), \\ x' &= Q_k(x_{m+1}, \ldots, x_{m+k})\, d, & y' &= Q_{k-1}(x_{m+2}, \ldots, x_{m+k})\, d, \end{aligned}$$

where m, k, d, and the x_j are positive integers with $x_1 \geq 2$, $x_{m+k} \geq 2$, and d is a divisor of n. The identity of exercise 32 now implies that $n/d = Q_{m+k}(x_1, \ldots, x_{m+k})$. Conversely, any given sequence of positive integers $x_1, \ldots, x_{m+k}$ such that $x_1 \geq 2$, $x_{m+k} \geq 2$, and $Q_{m+k}(x_1, \ldots, x_{m+k})$ divides n, corresponds in this way to $m + k - 1$ representations of n. (e) Therefore $nT_n = \lfloor (5n-3)/2 \rfloor + 2h(n)$.

34. [*HM40*] (H. Heilbronn.) (a) Let $h_d(n)$ be the number of representations of n as in exercise 33 such that $xd < x'$, plus half the number of representations with $xd = x'$. Let $g(n)$ be the number of representations without the requirement that $\gcd(x, y) = 1$. Prove that

$$h(n) = \sum_{d \backslash n} \mu(d) g\Big(\frac{n}{d}\Big), \qquad g(n) = 2 \sum_{d \backslash n} h_d\Big(\frac{n}{d}\Big).$$

(b) Generalizing exercise 33(b), show that for $d \geq 1$, $h_d(n) = \sum (n/(y(y+t))) + O(n)$, where the sum is over all integers y and t such that $\gcd(t, y) = 1$ and $0 < t \leq y < \sqrt{n/d}$. (c) Show that $\sum_{1 \leq y \leq n} (y/(y+t)) = \varphi(y) \ln 2 + O(\sigma_{-1}(y))$, where the sum is over the range $0 < t \leq y$, $\gcd(t, y) = 1$; and where $\sigma_{-1}(y) = \sum_{d \backslash y} (1/d)$. (d) Show that $\sum_{1 \leq y \leq n} \varphi(y)/y^2 = \sum_{1 \leq d \leq n} \mu(d) H_{\lfloor n/d \rfloor}/d^2$. (e) Hence we have the asymptotic formula $T_n = ((12 \ln 2)/\pi^2)(\ln n - \sum_{d \backslash n} \Lambda(d)/d) + O(\sigma_{-1}(n)^2)$.

35. [*HM41*] (A. C. Yao and D. E. Knuth.) Prove that the sum of all partial quotients for the fractions m/n, for $1 \leq m < n$, is equal to $2(\sum \lfloor x/y \rfloor + \lfloor n/2 \rfloor)$, where the sum is over all representations $n = xx' + yy'$ satisfying the conditions of exercise 33(a). Show that $\sum \lfloor x/y \rfloor = 3\pi^{-2} n (\ln n)^2 + O(n \log n (\log \log n)^2)$, and apply this to the "ancient" form of Euclid's algorithm that uses only subtraction instead of division.

36. [*M35*] (G. H. Bradley.) What is the smallest value of u_n such that the calculation of $\gcd(u_1, \ldots, u_n)$ by steps C1 and C2 in Section 4.5.2 requires N divisions, if Euclid's algorithm is used throughout? Assume that $N \geq n$.

37. [*M38*] (T. S. Motzkin and E. G. Straus.) Let $a_1, \ldots, a_n$ be positive integers. Show that $\max Q_n(a_{p(1)}, \ldots, a_{p(n)})$, over all permutations $p(1)\ldots p(n)$ of $\{1, 2, \ldots, n\}$, occurs when $a_{p(1)} \geq a_{p(n)} \geq a_{p(2)} \geq a_{p(n-1)} \geq \cdots$; and the minimum occurs when $a_{p(1)} \leq a_{p(n)} \leq a_{p(3)} \leq a_{p(n-2)} \leq a_{p(5)} \leq \cdots \leq a_{p(6)} \leq a_{p(n-3)} \leq a_{p(4)} \leq a_{p(n-1)} \leq a_{p(2)}$.

38. [*M25*] (J. Mikusiński.) Let $K(n) = \max_{m \geq 0} T(m, n)$. Theorem F shows that $K(n) \leq \lfloor \log_\phi(\sqrt{5}\, n + 1) \rfloor - 2$; prove that $K(n) \geq \frac{1}{2} \lceil \log_\phi(\sqrt{5}\, n + 1) \rceil - 2$.

▶ **39.** [*M25*] (R. W. Gosper.) If a baseball player's batting average is .334, what is the fewest possible number of times he has been at bat? [Note for non-baseball-fans: Batting average = (number of hits)/(times at bat), rounded to three decimal places.]

▶ **40.** [*M28*] (*The Stern–Peirce tree.*) Consider an infinite binary tree in which each node is labeled with the fraction $(p_l + p_r)/(q_l + q_r)$, where p_l/q_l is the label of the node's nearest left ancestor and p_r/q_r is the label of the node's nearest right ancestor. (A left ancestor is one that precedes a node in symmetric order, while a right ancestor follows the node. See Section 2.3.1 for the definition of symmetric order.) If the node has no left ancestors, $p_l/q_l = 0/1$; if it has no right ancestors, $p_r/q_r = 1/0$. Thus the label of the root is 1/1; the labels of its two sons are 1/2 and 2/1; the labels of the four nodes on level 2 are 1/3, 2/3, 3/2, and 3/1, from left to right; the labels of the eight nodes on level 3 are 1/4, 2/5, 3/5, 3/4, 4/3, 5/3, 5/2, 4/1; and so on.

Prove that p is relatively prime to q in each label p/q; furthermore, the node labeled p/q precedes the node labeled p'/q' in symmetric order if and only if the labels satisfy $p/q < p'/q'$. Find a connection between the continued fraction for the label of a node and the path to that node, thereby showing that each positive rational number appears as the label of exactly one node in the tree.

[redacted][*M40*] (J. Shallit, 1979.) Show that the regular continued fraction expansion of

$$\frac{1}{2^1} + \frac{1}{2^3} + \frac{1}{2^7} + \cdots = \sum_{n \geq 1} \frac{1}{2^{2^n - 1}}$$

contains only 1's and 2's and has a fairly [redacted]" Prove that the partial quotients of Liouville's numbers $\sum_{n \geq 1} l^{-n!}$ also have a regular pattern, when l is any integer ≥ 2. [The latter numbers, introduced by J. Liouville in *J. de Math. Pures et Appl.* **16** (1851), 133–142, were the first explicitly defined numbers to be proved *transcendental*. The former number and similar constants were first proved transcendental by A. J. Kempner, *Trans. Amer. Math. Soc.* **17** (1916), 476–482.]

42. [*M30*] (J. Lagrange, 1798.) Let X have the regular continued fraction expansion $/A_1, A_2, \ldots/$, and let $q_n = Q_n(A_1, \ldots, A_n)$. Let $\|x\|$ denote the distance from x to the nearest integer, namely $\min_p |x - p|$. Show that $\|qX\| \geq \|q_{n-1}X\|$ for $1 \leq q < q_n$. (Thus the denominators q_n of the so-called convergents $p_n/q_n = /A_1, \ldots, A_n/$ are the "record-breaking" integers that make $\|qX\|$ achieve new lows.)

43. [*M30*] (D. W. Matula.) Show that the "mediant rounding" rule for fixed-slash or floating-slash numbers, Eq. 4.5.1–1, can be implemented simply as follows, when the number $x > 0$ is not representable: Let the regular continued fraction expansion of x be $a_0 + /a_1, a_2, \ldots/$, and let $p_n = Q_{n+1}(a_0, \ldots, a_n)$, $q_n = Q_n(a_1, \ldots, a_n)$. Then $\text{round}(x) = (p_i/q_i)$, where (p_i/q_i) is representable but (p_{i+1}/q_{i+1}) is not. [*Hint:* See exercise 40.]

44. [*M25*] Suppose we are doing fixed slash arithmetic with mediant rounding, where the fraction (u/u') is representable if and only if $|u| < M$ and $0 \le u' < N$ and $\gcd(u, u') = 1$. Prove or disprove the identity $((u/u') \oplus (v/v')) \ominus (v/v') = (u/u')$ for all representable (u/u') and (v/v'), provided that $u' < \sqrt{N}$ and no overflow occurs.

45. [*HM48*] Develop the analysis of algorithms for computing the greatest common divisor of three or more integers.

4.5.4. Factoring into Primes

Several of the computational methods we have encountered in this book rest on the fact that every positive integer n can be expressed in a unique way in the form

$$n = p_1 p_2 \dots p_t, \qquad p_1 \le p_2 \le \cdots \le p_t, \tag{1}$$

where each p_k is prime. (When $n = 1$, this equation holds for $t = 0$.) It is unfortunately not a simple matter to find this prime factorization of n, or to determine whether or not n is prime. So far as anyone knows, it is a great deal harder to factor a large number n than to compute the greatest common divisor of two large numbers m and n; therefore we should avoid factoring large numbers whenever possible. But several ingenious ways to speed up the factoring process have been discovered, and we will now investigate some of them.

Divide and factor. First let us consider the most obvious algorithm for factorization: If $n > 1$, we can divide n by successive primes $p = 2, 3, 5, \dots$ until discovering the smallest p for which $n \bmod p = 0$. Then p is the smallest prime factor of n, and the same process may be applied to $n \leftarrow n/p$ in an attempt to divide this new value of n by p and by higher primes. If at any stage we find that $n \bmod p \neq 0$ but $\lfloor n/p \rfloor \le p$, we can conclude that n is prime; for if n is not prime, then by (1) we must have $n \ge p_1^2$, but $p_1 > p$ implies that $p_1^2 \ge (p+1)^2 > p(p+1) > p^2 + (n \bmod p) \ge \lfloor n/p \rfloor p + (n \bmod p) = n$. This leads us to the following procedure:

Algorithm A (*Factoring by division*). Given a positive integer N, this algorithm finds the prime factors $p_1 \le p_2 \le \cdots \le p_t$ of N as in Eq. (1). The method makes use of an auxiliary sequence of "trial divisors"

$$2 = d_0 < d_1 < d_2 < d_3 < \cdots, \tag{2}$$

which includes all prime numbers $\le \sqrt{N}$ (and which may also include values that are *not* prime, if it is convenient to do so). The sequence of d's must also include at least one value such that $d_k \ge \sqrt{N}$.

A1. [Initialize.] Set $t \leftarrow 0$, $k \leftarrow 0$, $n \leftarrow N$. (During this algorithm the variables t, k, n are related by the following condition: "$n = N/p_1 \dots p_t$, and n has no prime factors less than d_k.")

A2. [$n = 1$?] If $n = 1$, the algorithm terminates.

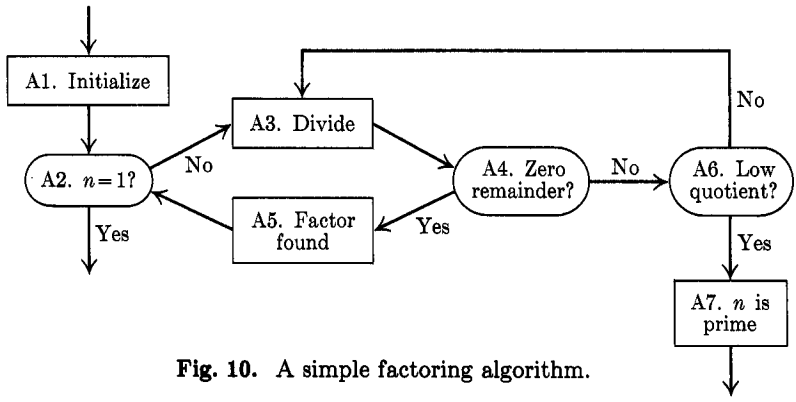


Fig. 10. A simple factoring algorithm.

A3. [Divide.] Set $q \leftarrow \lfloor n/d_k \rfloor$, $r \leftarrow n \bmod d_k$. (Here q and r are the quotient and remainder obtained when n is divided by d_k.)

A4. [Zero remainder?] If $r \neq 0$, go to step A6.

A5. [Factor found.] Increase t by 1, and set $p_t \leftarrow d_k$, $n \leftarrow q$. Return to step A2.

A6. [Low quotient?] If $q > d_k$, increase k by 1 and return to step A3.

A7. [n is prime.] Increase t by 1, set $p_t \leftarrow n$, and terminate the algorithm. ∎

As an example of Algorithm A, consider the factorization of the number $N = 25852$. We immediately find that $N = 2 \cdot 12926$; hence $p_1 = 2$. Furthermore, $12926 = 2 \cdot 6463$, so $p_2 = 2$. But now $n = 6463$ is not divisible by 2, 3, 5, ..., 19; we find that $n = 23 \cdot 281$, hence $p_3 = 23$. Finally $281 = 12 \cdot 23 + 5$ and $12 \leq 23$; hence $p_4 = 281$. The determination of 25852's factors has therefore involved a total of 12 division operations; on the other hand, if we had tried to factor the slightly smaller number 25849 (which is prime), at least 38 division operations would have been performed. This illustrates the fact that Algorithm A requires a running time roughly proportional to $\max(p_{t-1}, \sqrt{p_t})$. (If $t = 1$, this formula is valid if we adopt the convention $p_0 = 1$.)

The sequence d_0, d_1, d_2, ... of trial divisors used in Algorithm A can be taken to be simply 2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, ..., where we alternately add 2 and 4 after the first three terms. This sequence contains all numbers that are not multiples of 2 or 3; it also includes numbers such as 25, 35, 49, etc., which are not prime, but the algorithm will still give the correct answer. A further savings of 20 percent in computation time can be made by removing the numbers $30m \pm 5$ from the list for $m \geq 1$, thereby eliminating all of the spurious multiples of 5. The exclusion of multiples of 7 shortens the list by 14 percent more, etc. A compact bit table can be used to govern the choice of trial divisors.

If N is known to be small, it is reasonable to have a table of all the necessary primes as part of the program. For example, if N is less than a million, we need only include the 168 primes less than a thousand (followed by the value $d_{168} = 1000$, to terminate the list in case N is a prime larger than 997^2). Such a table

can be set up by means of a short auxiliary program, which builds the table just after the factoring program has been loaded into the computer; see Algorithm 1.3.2P, or see exercise 8.

How many trial divisions are necessary in Algorithm A? Let $\pi(x)$ be the number of primes $\le x$, so that $\pi(2) = 1$, $\pi(10) = 4$; the asymptotic behavior of this function has been studied extensively by many of the world's greatest mathematicians, beginning with Legendre in 1798. Numerous advances made during the nineteenth century culminated in 1899, when Charles de la Vallée Poussin proved that, for some $A > 0$,

$$\pi(x) = \int_2^x \frac{dt}{\ln t} + O(xe^{-A\sqrt{\log x}}). \tag{3}$$

[*Mém. Couronnés Acad. Roy. Belgique* **59** (1899), 1–74.] Integrating by parts yields

$$\pi(x) = \frac{x}{\ln x} + \frac{x}{(\ln x)^2} + \frac{2!\,x}{(\ln x)^3} + \cdots + \frac{r!\,x}{(\ln x)^{r+1}} + O\left(\frac{x}{(\log x)^{r+2}}\right) \tag{4}$$

for all fixed $r \ge 0$. The error term in (3) has subsequently been improved; for example, it can be replaced by $O(x\exp(-A(\log x)^{3/5}/(\log\log x)^{1/5}))$. [See A. Walfisz, *Weyl'sche Exponentialsummen in der neueren Zahlentheorie* (Berlin, 1963), Chapter 5.] Bernhard Riemann conjectured in 1859 that

$$\pi(x) = \sum_{k\ge 1} \mu(k)L(\sqrt[k]{x})/k + O(1) = L(x) - \frac{1}{2}L(\sqrt{x}) - \frac{1}{3}L(\sqrt[3]{x}) + \cdots \tag{5}$$

where $L(x) = \int_2^x dt/\ln t$, and his formula agrees well with actual counts when x is of reasonable size. For example, we have the following table:

x	$\pi(x)$	$x/\ln x$	$L(x)$	Riemann's formula
10^3	168	144.8	176.6	168.36
10^6	78498	72382.4	78626.5	78527.40
10^9	50847534	48254942.4	50849233.9	50847455.43

However, the distribution of large primes is not that simple, and Riemann's conjecture (5) was disproved by J. E. Littlewood in 1914; see Hardy and Littlewood, *Acta Math.* **41** (1918), 119–196, where it is shown that there is a positive constant C such that $\pi(x) > L(x) + C\sqrt{x}\log\log\log x/\log x$ for infinitely many x. Littlewood's result shows that prime numbers are inherently somewhat mysterious, and it will be necessary to develop deep properties of mathematics before their distribution is really understood. Riemann made another much more plausible conjecture, the famous "Riemann hypothesis," which states that the complex function $\zeta(z)$ is zero only when the real part of z is equal to $\frac{1}{2}$, except in the trivial cases where z is a negative even integer. This hypothesis, if true,

would imply that $\pi(x) = L(x) + O(\sqrt{x}\log x)$; see exercise 25. Richard Brent has used a method of D. H. Lehmer to verify Riemann's hypothesis computationally for all "small" values of z, by showing that $\zeta(z)$ has exactly 75,000,000 zeros whose imaginary part is in the range $0 < \Im z < 32585736.4$; all of these zeros have $\Re z = \frac{1}{2}$ and $\zeta'(z) \neq 0$. [*Math. Comp.* **33** (1979), 1361–1372.]

In order to analyze the average behavior of Algorithm A, we would like to know how large the largest prime factor p_t will tend to be. This question was first investigated by Karl Dickman [*Arkiv för Mat., Astron. och Fys.* **22A**, 10 (1930), 1–14], who studied the probability that a random integer between 1 and x will have its largest prime factor $\leq x^\alpha$. Dickman gave a heuristic argument to show that this probability approaches the limiting value $F(\alpha)$ as $x \to \infty$, where F can be calculated from the functional equation

$$F(\alpha) = \int_0^\alpha F\left(\frac{t}{1-t}\right)\frac{dt}{t}, \quad \text{for } 0 \leq \alpha \leq 1; \qquad F(\alpha) = 1 \quad \text{for } \alpha \geq 1. \tag{6}$$

His argument was essentially this: Given $0 < t < 1$, the number of integers less than x whose largest prime factor is between x^t and x^{t+dt} is $xF'(t)\,dt$. The number of primes p in that range is $\pi(x^{t+dt}) - \pi(x^t) = \pi(x^t + (\ln x)x^t\,dt) - \pi(x^t) = x^t\,dt/t$. For every such p, the number of integers n such that "$np \leq x$ and the largest prime factor of n is $\leq p$" is the number of $n \leq x^{1-t}$ whose largest prime factor is $\leq (x^{1-t})^{t/(1-t)}$, namely $x^{1-t}F\big(t/(1-t)\big)$. Hence $xF'(t)\,dt = (x^t\,dt/t)\big(x^{1-t}F\big(t/(1-t)\big)\big)$, and (6) follows by integration. This heuristic argument can be made rigorous; V. Ramaswami [*Bull. Amer. Math. Soc.* **55** (1949), 1122–1127] showed that the probability in question for fixed α is asymptotically $F(\alpha) + O(1/\log x)$, as $x \to \infty$, and many other authors have extended the analysis [see the survey by Karl K. Norton, *Memoirs Amer. Math. Soc.* **106** (1971), 9–27].

If $\frac{1}{2} \leq \alpha \leq 1$, formula (6) simplifies to

$$F(\alpha) = 1 - \int_\alpha^1 F\left(\frac{t}{1-t}\right)\frac{dt}{t} = 1 - \int_\alpha^1 \frac{dt}{t} = 1 + \ln\alpha.$$

Thus, for example, the probability that a random positive integer $\leq x$ has a prime factor $> \sqrt{x}$ is $1 - F(\frac{1}{2}) = \ln 2$, about 69 percent. In all such cases, Algorithm A must work hard.

The net result of this discussion is that Algorithm A will give the answer rather quickly if we want to factor a six-digit number; but for large N the amount of computer time for factorization by trial division will rapidly exceed practical limits, unless we are unusually lucky.

Later in this section we will see that there are fairly good ways to determine whether or not a reasonably large number n is prime, without trying all divisors up to $\sqrt{n}$. Therefore Algorithm A would often run faster if we inserted a primality test between steps A2 and A3; the running time for this improved algorithm would then be roughly proportional to p_{t-1}, the *second-largest* prime factor of N, instead of to $\max(p_{t-1}, \sqrt{p_t})$. By an argument analogous to

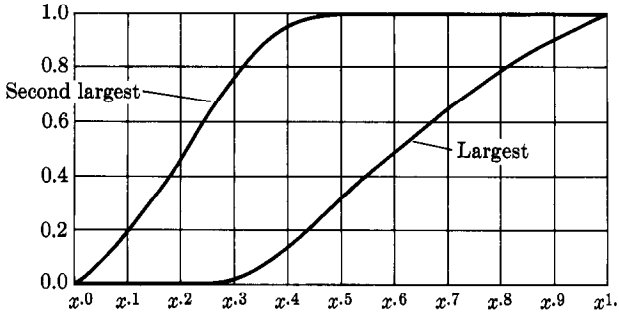


Fig. 11. Probability distribution functions for the two largest prime factors of a random integer $\leq x$.

Dickman's (see exercise 18), we can show that the second-largest prime factor of a random integer x will be $\leq x^{\beta}$ with approximate probability $G(\beta)$, where

$$G(\beta) = \int_0^{\beta} \left(G\left(\frac{t}{1-t}\right) - F\left(\frac{t}{1-t}\right)\right) \frac{dt}{t}, \quad \text{for } 0 \leq \beta \leq \tfrac{1}{2}. \tag{7}$$

Clearly $G(\beta) = 1$ for $\beta \geq \frac{1}{2}$. (See Fig. 11.) Numerical evaluation of (6) and (7) yields the following "percentage points":

$F(\alpha), G(\beta) =$	.01	.05	.10	.20	.35	.50	.65	.80	.90	.95	.99
$\alpha =$	.2697	.3348	.3785	.4430	.5220	.6065	.7047	.8187	.9048	.9512	.9900
$\beta =$	.0056	.0273	.0531	.1003	.1611	.2117	.2582	.3104	.3590	.3967	.4517

Thus, the second-largest prime factor will be $\leq x^{.2117}$ about half the time, etc.

The *total number of prime factors*, t, has also been intensively analyzed. Obviously $1 \leq t \leq \lg N$, but these lower and upper bounds are seldom achieved. It is possible to prove that if N is chosen at random between 1 and x, the probability that $t \leq \ln\ln x + c\sqrt{\ln\ln x}$ approaches

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{c} e^{-u^2/2}\, du \tag{8}$$

as $x \to \infty$, for any fixed c. In other words, the distribution of t is essentially normal, with mean and variance $\ln\ln x$; about 99.73 percent of all the large integers $\leq x$ have $|t - \ln\ln x| \leq 3\sqrt{\ln\ln x}$. Furthermore the average value of $t - \ln\ln x$ for $1 \leq N \leq x$ is known to approach

$$\gamma + \sum_{p \text{ prime}} \big(\ln(1 - 1/p) + 1/(p-1)\big) = \mathbf{1.03465\ 38818\ 97438}.$$

[Cf. G. H. Hardy and E. M. Wright, *Introduction to the Theory of Numbers*, 4th ed. (Oxford, 1960), §22.11; see also [redacted].]

The size of prime factors has a remarkable connection with permutations: The average number of bits in the kth largest prime factor of a random n-bit integer is asymptotically the same as the average length of the kth largest cycle of a random n-element permutation, as $n \to \infty$. [See D. E. Knuth and L. Trabb Pardo, *Theoretical Comp. Sci.* **3** (1976), 321–348.] It follows that Algorithm A usually finds a few small factors and then begins a long-drawn-out search for the big ones that are left.

Factoring à la Monte Carlo. Near the beginning of Chapter 3, we observed that "a random number generator chosen at random isn't very random." This principle, which worked against us in that chapter, has the redeeming virtue that it leads to a surprisingly efficient method of factorization, discovered by J. M. Pollard [*BIT* **15** (1975), 331–334]. The number of computational steps in Pollard's method is on the order of $\sqrt{p_{t-1}}$, so it is significantly faster than Algorithm A when N is large. According to (7) and Fig. 11, the running time will usually be well under $N^{1/4}$.

Let $f(x)$ be any polynomial with integer coefficients, and consider the two sequences defined by

$$x_0 = y_0 = A; \qquad x_{m+1} = f(x_m) \bmod N, \qquad y_{m+1} = f(y_m) \bmod p, \tag{9}$$

where p is any prime factor of N. It follows that

$$y_m = x_m \bmod p, \qquad \text{for } m \geq 1. \tag{10}$$

Now exercise 3.1–7 shows that we will have $y_m = y_{l(m)-1}$ for some $m \geq 1$, where $l(m)$ is the greatest power of 2 that is $\leq m$. Thus $x_m - x_{l(m)-1}$ will be a multiple of p. Furthermore if $f(y) \bmod p$ behaves as a random mapping from the set $\{0, 1, \ldots, p-1\}$ into itself, exercise 3.1–12 shows that the average value of the least such m will be of order $\sqrt{p}$. In fact, exercise 4 below shows that this average value for random mappings is less than $1.625\,Q(p)$, where the function $Q(p) \approx \sqrt{\pi p/2}$ was defined in Section 1.2.11.3. If the different prime divisors of N correspond to different values of m (as they almost surely will, when N is large), we will be able to find them by calculating $\gcd(x_m - x_{l(m)-1}, N)$ for $m = 1, 2, 3, \ldots$, until the unfactored residue is prime.

From the theory in Chapter 3, we know that a linear polynomial $f(x) = ax + c$ will not be sufficiently random for our purposes. The next-simplest case is quadratic, say $f(x) = x^2 + 1$; although we don't *know* that this function is sufficiently random, our lack of knowledge tends to support the hypothesis of randomness, and empirical tests show that this f does work essentially as predicted. In fact, f is probably slightly *better* than random, since $x^2 + 1$ takes on only $\frac{1}{2}(p+1)$ distinct values mod p. Therefore the following procedure is reasonable:

Algorithm B (*Monte Carlo factorization*). This algorithm outputs the prime factors of a given integer $N \geq 2$, with high probability, although there is a chance that it will fail.

B1. [Initialize.] Set $x \leftarrow 5$, $x' \leftarrow 2$, $k \leftarrow 1$, $l \leftarrow 1$, $n \leftarrow N$. (During this algorithm, n is the unfactored part of N, and the variables x and x' represent the quantities $x_m \bmod n$ and $x_{l(m)-1} \bmod n$ in (9), where $f(x) = x^2+1$, $A = 1$, $l = l(m)$, and $k = 2l - m$.)

B2. [Test primality.] If n is prime (see the discussion below), output n; the algorithm terminates.

B3. [Factor found?] Set $g \leftarrow \gcd(x'-x, n)$. If $g = 1$, go on to step B4; otherwise output g. Now if $g = n$, the algorithm terminates (and it has failed, because we know that n isn't prime). Otherwise set $n \leftarrow n/g$, $x \leftarrow x \bmod n$, $x' \leftarrow x' \bmod n$, and return to step B2. (Note that g may not be prime; this should be tested. In the rare event that g isn't prime, its prime factors probably won't be determinable with this algorithm.)

B4. [Advance.] Set $k \leftarrow k - 1$. If $k = 0$, set $x' \leftarrow x$, $l \leftarrow 2l$, $k \leftarrow l$. Set $x \leftarrow (x^2 + 1) \bmod n$ and return to B3. ∎

As an example of Algorithm B, let's try to factor $N = 25852$ again. The third execution of step B3 will output $g = 4$ (which isn't prime). After six more iterations the algorithm finds the factor $g = 23$. Algorithm B has not distinguished itself in this example, but of course it was designed to factor *big* numbers. Algorithm A takes much longer to find large prime factors, but it can't be beat when it comes to removing the small ones. In practice, we should run Algorithm A awhile before switching over to Algorithm B.

We can get a better idea of Algorithm B's prowess by considering the ten largest six-digit primes. The number of iterations, $m(p)$, that Algorithm B needs to find the factor p is given in the following table:

$p =$	999863	999883	999907	999917	999931	999953	999959	999961	999979	999983
$m(p) =$	276	409	2106	1561	1593	1091	474	1819	395	814

Experiments indicate that $m(p)$ has an average value of about $2\sqrt{p}$, and it never exceeds $12\sqrt{p}$ when $p < 1000000$. The maximum $m(p)$ for $p < 10^6$ is $m(874771) = 7685$; and the maximum of $m(p)/\sqrt{p}$ occurs when $p = 290047$, $m(p) = 6251$. According to these experimental results, almost all 12-digit numbers can be factored in fewer than 2000 iterations of Algorithm B (compared to roughly 100,000 divisions in Algorithm A).

The time-consuming operations in each iteration of Algorithm B are the multiple-precision multiplication and division in step B4, and the gcd in step B3. If the gcd operation is slow, Pollard suggests gaining speed by accumulating the product mod n of, say, ten consecutive $(x'-x)$ values before taking each gcd; this replaces 90 percent of the gcd operations by a single multiplication and division while only slightly increasing the chance of failure. He also suggests starting with $m = q$ instead of $m = 1$ in step B1, where q is, say, $\frac{1}{10}$ the number of iterations you are planning to use.

In those rare cases where failure occurs for large N, we could try using $f(x) = x^2 + c$ for some $c \neq 0$ or 1. The value $c = -2$ should also be avoided, since the recurrence $x_{m+1} = x_m^2 - 2$ has solutions of the form $x_m = r^{2^m} + r^{-2^m}$. Other values of c do not seem to lead to simple relationships mod p, and they should all be satisfactory when used with suitable starting values.

Richard Brent used a modification of Algorithm B in 1980 to discover the prime factor 1238926361552897 of $2^{256} + 1$.

Fermat's method. Another approach to the factoring problem, which was used by Pierre de Fermat in 1643, is more suited to finding large factors than small ones. [Fermat's original description of his method, translated into English, can be found in L. E. Dickson's monumental *History of the Theory of Numbers* **1** (New York: Chelsea, 1952), 357.]

Assume that $N = uv$, where $u \leq v$. For practical purposes we may assume that N is odd; this means that u and v are odd, and we can let

$$x = (u + v)/2, \qquad y = (v - u)/2, \tag{11}$$

$$N = x^2 - y^2, \qquad 0 \leq y < x \leq N. \tag{12}$$

Fermat's method consists of searching systematically for values of x and y that satisfy Eq. (12). The following algorithm shows how factoring can therefore be done *without using any division:*

Algorithm C (*Factoring by addition and subtraction*). Given an odd number N, this algorithm determines the largest factor of N less than or equal to $\sqrt{N}$.

C1. [Initialize.] Set $x' \leftarrow 2\lfloor\sqrt{N}\rfloor + 1$, $y' \leftarrow 1$, $r \leftarrow \lfloor\sqrt{N}\rfloor^2 - N$. (During this algorithm x', y', r correspond respectively to $2x + 1$, $2y + 1$, $x^2 - y^2 - N$ as we search for a solution to (12); we will have $|r| < x'$ and $y' < x'$.)

C2. [Test r.] If $r \leq 0$, go to step C4.

C3. [Step y.] Set $r \leftarrow r - y'$, $y' \leftarrow y' + 2$, and return to C2.

C4. [Done?] If $r = 0$, the algorithm terminates; we have

$$N = \big((x' - y')/2\big)\big((x' + y' - 2)/2\big),$$

and $(x' - y')/2$ is the largest factor of N less than or equal to $\sqrt{N}$.

C5. [Step x.] Set $r \leftarrow r + x'$, $x' \leftarrow x' + 2$, and return to C3. ∎

The reader may find it amusing to find the factors of 377 by hand, using this algorithm. The number of steps needed to find the factors u and v of $N = uv$ is essentially proportional to $(x' + y' - 2)/2 - \lfloor\sqrt{N}\rfloor = v - \lfloor\sqrt{N}\rfloor$; this can, of course, be a very large number, although each step can be done very rapidly on most computers. An improvement that requires only $O(N^{1/3})$ operations in the worst case has been developed by R. S. Lehman [*Math. Comp.* **28** (1974), 637–646].

It is not quite correct to call Algorithm C "Fermat's method," since Fermat used a somewhat more streamlined approach. Algorithm C's main loop is quite

fast on computers, but it is not very suitable for hand calculation. Fermat actually did not keep the running value of y; he would look at $x^2 - N$ and tell whether or not this quantity was a perfect square by looking at its least significant digits. (The last two digits of a perfect square must be 00, $e1$, $e4$, 25, $o6$, or $e9$, where e is an even digit and o is an odd digit.) Therefore he avoided the operations of steps C2 and C3, replacing them by an occasional determination that a certain number is not a perfect square.

Fermat's method of looking at the rightmost digits can, of course, be generalized by using other moduli. Suppose for clarity that $N = 11111$, and consider the following table:

m	if $x \bmod m$ is	then $x^2 \bmod m$ is	and $(x^2 - N) \bmod m$ is
3	0, 1, 2	0, 1, 1	1, 2, 2
5	0, 1, 2, 3, 4	0, 1, 4, 4, 1	4, 0, 3, 3, 0
7	0, 1, 2, 3, 4, 5, 6	0, 1, 4, 2, 2, 4, 1	5, 6, 2, 0, 0, 2, 6
8	0, 1, 2, 3, 4, 5, 6, 7	0, 1, 4, 1, 0, 1, 4, 1	1, 2, 5, 2, 1, 2, 5, 2
11	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10	0, 1, 4, 9, 5, 3, 3, 5, 9, 4, 1	10, 0, 3, 8, 4, 2, 2, 4, 8, 3, 0

If $x^2 - N$ is to be a perfect square y^2, it must have a residue mod m consistent with this fact, for all m. For example, if $N = 11111$ and $x \bmod 3 \neq 0$, then $(x^2 - N) \bmod 3 = 2$, so $x^2 - N$ cannot be a perfect square; therefore x must be a multiple of 3 whenever $11111 = x^2 - y^2$. The table tells us, in fact, that

$$\begin{aligned} x \bmod 3 &= 0; \\ x \bmod 5 &= 0, \text{ 1, or 4}; \\ x \bmod 7 &= 2, \text{ 3, 4, or 5}; \\ x \bmod 8 &= 0 \text{ or 4 (hence } x \bmod 4 = 0); \\ x \bmod 11 &= 1, \text{ 2, 4, 7, 9, or 10.} \end{aligned} \tag{13}$$

This narrows down the search for x considerably. For example, x must be a multiple of 12. We must have $x \geq \lceil \sqrt{N} \rceil = 106$, and it is easy to verify that the first value of $x \geq 106$ that satisfies all of the conditions in (13) is $x = 144$. Now $144^2 - 11111 = 9625$, and by attempting to take the square root of 9625 we find that it is not a square. The first value of $x > 144$ that satisfies (13) is $x = 156$. In this case $156^2 - 11111 = 13225 = 115^2$; so we have found the desired solution $x = 156$, $y = 115$. This calculation shows that $11111 = 41 \cdot 271$.

The hand calculations involved in the above example are comparable to the amount of work required to divide 11111 by 13, 17, 19, 23, 29, 31, 37, and 41, even though the factors 41 and 271 are not very close to each other; thus we can see the advantages of Fermat's method.

In place of the moduli considered in (13), we can use any powers of distinct primes. For example, if we had used 25 in place of 5, we would find that the only permissible values of $x \bmod 25$ are 0, 5, 6, 10, 15, 19, and 20. This gives more information than (13). In general, we will get more information modulo p^2 than we do modulo p, for odd primes p, whenever $x^2 - N \equiv 0$ (modulo p) has a solution x.

The modular method just used is called a *sieve procedure*, since we can imagine passing all integers through a "sieve" for which only those values with $x \bmod 3 = 0$ come out, then sifting these numbers through another sieve that allows only numbers with $x \bmod 5 = 0$, 1, or 4 to pass, etc. Each sieve by itself will remove about half of the remaining values (see exercise 6); and when we sieve with respect to moduli that are relatively prime in pairs, each sieve is independent of the others because of the Chinese remainder theorem (Theorem 4.3.2C). So if we sieve with respect to, say, 30 different primes, only about one value in every 2^{30} will need to be examined to see if $x^2 - N$ is a perfect square y^2.

Algorithm D (*Factoring with sieves*). Given an odd number N, this algorithm determines the largest factor of N less than or equal to $\sqrt{N}$. The procedure uses moduli $m_1, m_2, \ldots, m_r$ that are relatively prime to each other in pairs and relatively prime to N. We assume that r "sieve tables" $S[i, j]$ for $0 \le j < m_i$, $1 \le i \le r$, have been prepared, where

$$S[i,j] = \begin{cases} 1, & \text{if } j^2 - N \equiv y^2 \text{ (modulo } m_i) \text{ has a solution } y; \\ 0, & \text{otherwise.} \end{cases}$$

D1. [Initialize.] Set $x \leftarrow \lceil \sqrt{N} \rceil$, and set $k_i \leftarrow (-x) \bmod m_i$ for $1 \le i \le r$. (Throughout this algorithm the index variables $k_1, k_2, \ldots, k_r$ will be set so that $(-x) \bmod m_i = k_i$.)

D2. [Sieve.] If $S[i, k_i] = 1$ for $1 \le i \le r$, go to step D4.

D3. [Step x.] Set $x \leftarrow x + 1$, and set $k_i \leftarrow (k_i - 1) \bmod m_i$ for $1 \le i \le r$. Return to step D2.

D4. [Test $x^2 - N$.] Set $y \leftarrow \lfloor \sqrt{x^2 - N} \rfloor$ or to $\lceil \sqrt{x^2 - N} \rceil$. If $y^2 = x^2 - N$, then $(x - y)$ is the desired factor, and the algorithm terminates. Otherwise return to step D3. ∎

There are several ways to make this procedure run fast. For example, we have seen that if $N \bmod 3 = 2$, then x must be a multiple of 3; we can set $x = 3x'$, and use a different sieve corresponding to x', increasing the speed threefold. If $N \bmod 9 = 1$, 4, or 7, then x must be congruent respectively to ± 1, ± 2, or ± 4 (modulo 9); so we run two sieves (one for x' and one for x'', where $x = 9x' + a$ and $x = 9x'' - a$) to increase the speed by a factor of $4\frac{1}{2}$. If $N \bmod 4 = 3$, then $x \bmod 4$ is known and the speed is increased by an additional factor of 4; in the other case, when $N \bmod 4 = 1$, x must be odd so the speed may be doubled. Another way to double the speed of the algorithm (at the expense of storage space) is to combine pairs of moduli, using $m_{r-k} m_k$ in place of m_k for $1 \le k < \frac{1}{2}r$.

An even more important method of speeding up Algorithm D is to use the "Boolean operations" found on most binary computers. Let us assume, for example, that MIX is a binary computer with 30 bits per word. The tables $S[i, k_i]$ can be kept in memory with one bit per entry; thus 30 values can be stored in a single word. The operation AND, which replaces the kth bit of the accumulator

by zero if the kth bit of a specified word in memory is zero, for $1 \le k \le 30$, can be used to process 30 values of x at once! For convenience, we can make several copies of the tables $S[i,j]$ so that the table entries for m_i involve $\operatorname{lcm}(m_i, 30)$ bits; then the sieve tables for each modulus fill an integral number of words. Under these assumptions, 30 executions of the main loop in Algorithm D are equivalent to code of the following form:

```
D2  LD1  K1      rI1 ← k'_1.
    LDA  S1,1    rA ← S'[1, rI1].
    DEC1 1       rI1 ← rI1 − 1.
    J1NN *+2
    INC1 M1      If rI1 < 0, set rI1 ← rI1 + lcm(m_1, 30).
    ST1  K1      k'_1 ← rI1.
    LD1  K2      rI1 ← k'_2.
    AND  S2,1    rA ← rA ∧ S'[2, rI1].
    DEC1 1       rI1 ← rI1 − 1.
    J1NN *+2
    INC1 M2      If rI1 < 0, set rI1 ← rI1 + lcm(m_2, 30).
    ST1  K2      k'_2 ← rI1.
    LD1  K3      rI1 ← k'_3.
    ...          (m_3 through m_r are like m_2)
    ST1  Kr      k'_r ← rI1.
    INCX 30      x ← x + 30.
    JAZ  D2      Repeat if all sieved out. ∎
```

The number of cycles for 30 iterations is essentially $2 + 8r$; if $r = 11$, this means three cycles are being used on each iteration, just as in Algorithm C, and Algorithm C involves $y = \frac{1}{2}(v - u)$ more iterations.

If the table entries for m_i do not come out to be an integral number of words, further shifting of the table entries would be necessary on each iteration in order to align the bits properly. This would add quite a lot of coding to the main loop and it would probably make the program too slow to compete with Algorithm C unless $v/u \le 100$ (see exercise 7).

Sieve procedures can be applied to a variety of other problems, not necessarily having much to do with arithmetic. A survey of these techniques has been prepared by Marvin C. Wunderlich, *JACM* **14** (1967), 10–19.

Special sieve machines (of reasonably low cost) have been constructed by D. H. Lehmer and his associates over a period of many years; see, for example, *AMM* **40** (1933), 401–406. Lehmer's electronic delay-line sieve, which began operating in 1965, processes one million numbers per second. Thus, each iteration of the loop in Algorithm D can be performed in one microsecond on this device. Another way to factor with sieves is described by D. H. and Emma Lehmer in *Math. Comp.* **28** (1974), 625–635.

Primality testing. None of the algorithms we have discussed so far is an efficient way to determine that a large number n is prime. Fortunately, there are other methods available for settling this question; efficient methods have been devised

by É. Lucas and others, notably D. H. Lehmer [see *Bull. Amer. Math. Soc.* **33** (1927), 327–340].

According to Fermat's theorem (Theorem 1.2.4F), we have $x^{p-1} \bmod p = 1$ whenever p is prime and x is not a multiple of p. Furthermore, there are efficient ways to calculate $x^{n-1} \bmod n$, requiring only $O(\log n)$ operations of multiplication mod n. (We shall study these in Section 4.6.3 below.) Therefore we can often determine that n is *not* prime when this relationship fails.

For example, Fermat once verified that the numbers 2^1+1, 2^2+1, 2^4+1, 2^8+1, and $2^{16}+1$ are prime. In a letter to Mersenne written in 1640, Fermat conjectured that $2^{2^n}+1$ is always prime, but said he was unable to determine definitely whether the number $4294967297 = 2^{32}+1$ is prime or not. Neither Fermat nor Mersenne ever resolved this problem, although they could have done it as follows: The number $3^{2^{32}} \bmod (2^{32}+1)$ can be computed by doing 32 operations of squaring modulo $2^{32}+1$, and the answer is 3029026160; therefore (by Fermat's own theorem, which he discovered in the same year 1640!) the number $2^{32}+1$ is *not* prime. This argument gives us absolutely no idea what the factors are, but it answers Fermat's question.

Fermat's theorem is a powerful test for showing non-primality of a given number. When n is not prime, it is always possible to find a value of $x < n$ such that $x^{n-1} \bmod n \neq 1$; experience shows that, in fact, such a value can almost always be found very quickly. There are some rare values of n for which $x^{n-1} \bmod n$ is frequently equal to unity, but then n has a factor less than $\sqrt[3]{n}$; see exercise 9.

The same method can be extended to prove that a large prime number n really *is* prime, by using the following idea: *If there is a number x for which the order of x modulo n is equal to $n-1$, then n is prime.* (The order of x modulo n is the smallest positive integer k such that $x^k \bmod n = 1$; see Section 3.2.1.2.) For this condition implies that the numbers $x^1 \bmod n, \ldots, x^{n-1} \bmod n$ are distinct and relatively prime to n, so they must be the numbers $1, 2, \ldots, n-1$ in some order; thus n has no proper divisors. If n is prime, such a number x (called a "primitive root" of n) will always exist; see exercise 3.2.1.2–16. In fact, primitive roots are rather numerous. There are $\varphi(n-1)$ of them, and this is quite a substantial number, since $n/\varphi(n-1) = O(\log\log n)$.

It is unnecessary to calculate $x^k \bmod n$ for all $k \leq n-1$ to determine if the order of x is $n-1$ or not. The order of x will be $n-1$ if and only if

i) $x^{n-1} \bmod n = 1$;

ii) $x^{(n-1)/p} \bmod n \neq 1$ for all primes p that divide $n-1$.

For $x^s \bmod n = 1$ if and only if s is a multiple of the order of x modulo n. If the two conditions hold, and if k is the order of x modulo n, we therefore know that k is a divisor of $n-1$, but not a divisor of $(n-1)/p$ for any prime factor p of $n-1$; the only remaining possibility is $k = n-1$. This completes the proof that conditions (i) and (ii) suffice to establish the primality of n.

Exercise 10 shows that we can in fact use different values of x for each of the primes p, and n will still be prime. We may restrict consideration to prime

values of x, since the order of uv modulo n divides the least common multiple of the orders of u and v by exercise 3.2.1.2–15. Conditions (i) and (ii) can be tested efficiently by using the rapid methods for evaluating powers of numbers discussed in Section 4.6.3. But it is necessary to know the prime factors of $n-1$, so we have an interesting situation in which the factorization of n depends on that of $n-1$.

An example. The study of a reasonably typical large factorization will help to fix the ideas we have discussed so far. Let us try to find the prime factors of $2^{214}+1$, a 65-digit number. The factorization can be initiated with a bit of clairvoyance if we notice that

$$2^{214}+1=(2^{107}-2^{54}+1)(2^{107}+2^{54}+1); \tag{14}$$

this identity is a special case of some factorizations discovered by A. Aurifeuille in 1873 [see Dickson's *History*, **1**, p. 383]. The problem now boils down to examining each of the 33-digit factors in (14).

A computer program readily discovers that $2^{107}-2^{54}+1=5\cdot 857\cdot n_0$, where

$$n_0=37866809061660057264219253397 \tag{15}$$

is a 29-digit number having no prime factors less than 1000. A multiple-precision calculation using the "binary method" of Section 4.6.3 shows that

$$3^{n_0-1} \bmod n_0 = 1,$$

so we suspect that n_0 is prime. It is certainly out of the question to prove that n_0 is prime by trying the 10 million million or so potential divisors, but the method discussed above gives a feasible test for primality: our next goal is to factor n_0-1. With little difficulty, our computer will tell us that

$$n_0-1=2\cdot 2\cdot 19\cdot 107\cdot 353\cdot n_1, \qquad n_1=1319127075410822604930 1.$$

Here $3^{n_1-1} \bmod n_1 \neq 1$, so n_1 is not prime; by continuing Algorithm A or Algorithm B we find

$$n_1=91813\cdot n_2, \qquad n_2=143675413657196977.$$

This time $3^{n_2-1} \bmod n_2 = 1$, so we will try to prove that n_2 is prime. This requires the factorization $n_2-1=2\cdot 2\cdot 2\cdot 2\cdot 3\cdot 3\cdot 547\cdot n_3$, where $n_3=1824032775457$. Since $3^{n_3-1} \bmod n_3 \neq 1$, we know that n_3 is composite, and Algorithm A finds that $n_3=1103\cdot n_4$, where $n_4=1653701519$. The number n_4 behaves like a prime (i.e., $3^{n_4-1} \bmod n_4 = 1$), so we calculate

$$n_4-1=2\cdot 7\cdot 19\cdot 23\cdot 137\cdot 1973.$$

Good; this is our first complete factorization. We are now ready to backtrack to the previous subproblem, proving that n_4 is prime. Using the procedure suggested by exercise 10, we compute the following values:

x	p	$x^{(n_4-1)/p} \bmod n_4$	$x^{n_4-1} \bmod n_4$
2	2	1	(1)
2	7	766408626	(1)
2	19	332952683	(1)
2	23	1154237810	(1)
2	137	373782186	(1)
2	1973	490790919	(1)
3	2	1	(1)
5	2	1	(1)
7	2	1653701518	1

(16)

(Here "(1)" means a result of 1 that needn't be computed since it can be deduced from previous calculations.) Thus n_4 is prime, and $n_2 - 1$ has been completely factored. A similar calculation shows that n_2 is prime, and this complete factorization of $n_0 - 1$ finally shows [after still another calculation like (16)] that n_0 is prime.

The last three lines of (16) represent a search for an integer x that satisfies $x^{(n_4-1)/2} \not\equiv x^{n_4-1} \equiv 1$ (modulo n_4). If n_4 is prime, we have only a 50-50 chance of success, so the case $p = 2$ is typically the hardest one to verify. We could streamline this part of the calculation by using the law of quadratic reciprocity (cf. exercise 23), which tells us for example that $5^{(q-1)/2} \equiv 1$ (modulo q) whenever q is a prime congruent to ± 1 (modulo 5). Merely calculating $n_4 \bmod 5$ would have told us right away that $x = 5$ could not possibly help in showing that n_4 is prime. In fact, however, the result of exercise 26 implies that the case $p = 2$ doesn't really need to be considered at all when testing n for primality, unless $n - 1$ is divisible by a high power of 2, so we could have dispensed with the last three lines of (16) entirely.

The next quantity to be factored is the other half of (14), namely

$$n_5 = 2^{107} + 2^{54} + 1.$$

Since $3^{n_5-1} \bmod n_5 \neq 1$, we know that n_5 is not prime, and Algorithm B shows that $n_5 = 843589 \cdot n_6$, where $n_6 = 192343993140277293096491917$. Unfortunately, $3^{n_6-1} \bmod n_6 \neq 1$, so we are left with a 27-digit nonprime. Continuing Algorithm B might well exhaust our patience (not our budget—nobody is paying for this, we're using idle time on a weekend rather than "prime time"). But the sieve method of Algorithm D will be able to crack n_6 into its two factors,

$$n_6 = 8174912477117 \cdot 23528569104401.$$

This result could *not* have been discovered by Algorithm A in a reasonable length of time. (A few million iterations of Algorithm B would probably have sufficed.)

Now the computation is complete: $2^{214}+1$ has the prime factorization

$$5 \cdot 857 \cdot 843589 \cdot 8174912477117 \cdot 23528569104401 \cdot n_0,$$

where n_0 is the 29-digit prime in (15). A certain amount of good fortune entered into these calculations, for if we had not started with the known factorization (14) it is quite probable that we would first have cast out the small factors, reducing n to $n_6 n_0$. This 55-digit number would have been much more difficult to factor—Algorithm D would be useless and Algorithm B would have to work overtime because of the high precision necessary.

Dozens of further numerical examples can be found in an article by John Brillhart and J. L. Selfridge, *Math. Comp.* **21** (1967), 87–96.

Improved primality tests. Since the above procedure for proving that n is prime requires the complete factorization of $n-1$, it will bog down for large n. Another technique, which uses the factorization of $n+1$ instead, is described in exercise 15; if $n-1$ turns out to be too hard, $n+1$ might be easier.

Significant improvements are available for dealing with large n. For example, it is not difficult to prove a stronger converse of Fermat's theorem that requires only a partial factorization of $n-1$. Exercise 26 shows that we could have avoided most of the calculations in (16); the three conditions $2^{n_4-1} \bmod n_4 = \gcd(2^{(n_4-1)/23}-1, n_4) = \gcd(2^{(n_4-1)/1973}-1, n_4) = 1$ are sufficient by themselves to prove that n_4 is prime. Brillhart, Lehmer, and Selfridge have in fact developed a method that works when the numbers $n-1$ and $n+1$ have been only partially factored [*Math. Comp.* **29** (1975), 620–647, Corollary 11]: Suppose $n-1 = f^- r^-$ and $n+1 = f^+ r^+$, where we know the complete factorizations of f^- and f^+, and we also know that all factors of r^- and r^+ are $\geq b$. If the product $\big(b^3 f^- f^+ \max(f^-, f^+)\big)$ is greater than $2n$, a small amount of additional computation, described in their paper, will determine whether or not n is prime. Therefore numbers of up to 35 digits can usually be tested for primality in 2 or 3 seconds, simply by casting out all prime factors < 30030 from $n \pm 1$ [see J. L. Selfridge and M. C. Wunderlich, *Proc. Fourth Manitoba Conf. Numer. Math.* (1974), 109–120]. The partial factorization of other quantities like $n^2 \pm n + 1$ and $n^2 + 1$ can be used to improve this method still further [see H. C. Williams and J. S. Judd, *Math. Comp.* **30** (1976), 157–172, 867–886].

In practice, when n has no small prime factors and $3^{n-1} \bmod n = 1$, it has almost always turned out that n is prime. (One of the rare exceptions in the author's experience is $n = \frac{1}{7}(2^{28}-9) = 2341 \cdot 16381$.) On the other hand, some nonprime values of n are definitely bad news for the primality test we have discussed, because it might happen that $x^{n-1} \bmod n = 1$ for all x relatively prime to n (see exercise 9). One such number is $n = 3 \cdot 11 \cdot 17 = 561$; here $\lambda(n) = \operatorname{lcm}(2, 10, 16) = 80$ in the notation of Eq. 3.2.1.2–9, so $x^{80} \bmod 561 = 1 = x^{560} \bmod 561$ whenever x is relatively prime to 561. Our procedure would repeatedly fail to show that such an n is prime, until we had stumbled across one of its divisors. To improve the method, we need a quick way to determine the nonprimality of nonprime n, even in such pathological cases.

The following surprisingly simple procedure is guaranteed to do the job with high probability:

Algorithm P (*Probabilistic primality test*). Given an odd integer n, this algorithm attempts to decide whether or not n is prime. By repeating the algorithm several times, as explained in the remarks below, it is possible to be extremely confident about the primality of n, in a precise sense, yet the primality will not be rigorously proved. Let $n = 1 + 2^k q$, where q is odd.

P1. [Generate x.] Let x be a random integer in the range $1 < x < n$.

P2. [Exponentiate.] Set $j \leftarrow 0$ and $y \leftarrow x^q \bmod n$. (As in our previous primality test, $x^q \bmod n$ should be calculated in $O(\log q)$ steps, cf. Section 4.6.3.)

P3. [Done?] (Now $y = x^{2^j q} \bmod n$.) If $j = 0$ and $y = 1$, or if $y = n - 1$, terminate the algorithm and say "n is probably prime." If $j > 0$ and $y = 1$, go to step P5.

P4. [Increase j.] Increase j by 1. If $j < k$, set $y \leftarrow y^2 \bmod n$ and return to step P3.

P5. [Not prime.] Terminate the algorithm and say that "n is definitely not prime." ∎

The idea underlying Algorithm P is that if $n = 1 + 2^k q$ is prime and $x^q \bmod n \neq 1$, the sequence of values

$$x^q \bmod n, \quad x^{2q} \bmod n, \quad x^{4q} \bmod n, \quad \ldots, \quad x^{2^k q} \bmod n$$

will end with 1, and the value just preceding the first appearance of 1 will be $n - 1$. (The only solutions to $y^2 \equiv 1$ (modulo p) are $y \equiv \pm 1$, when p is prime, since $(y - 1)(y + 1)$ must be a multiple of p.)

Exercise 22 proves the basic fact that Algorithm P will be wrong at most $\frac{1}{4}$ of the time, for all n. Actually it will rarely fail at all, for most n; but the crucial point is that the probability of failure is bounded *regardless* of the value of n.

Suppose we invoke Algorithm P repeatedly, choosing x independently and at random whenever we get to step P1. If the algorithm ever reports that n is nonprime, we can say that n definitely isn't prime. But if the algorithm reports 25 times in a row that n is "probably prime," we can say that n is "almost surely prime." For the probability is less than $(1/4)^{25}$ that such a 25-times-in-a-row procedure gives the wrong information about n. This is less than one chance in a quadrillion; even if we certified a billion different primes with such a procedure, the expected number of mistakes would be less than $\frac{1}{1000000}$. It's much more likely that our computer has dropped a bit in its calculations, due to hardware malfunctions or cosmic radiations, than that Algorithm P has repeatedly guessed wrong!

Probabilistic algorithms like this lead us to question our traditional standards of reliability. Do we really *need* to have a rigorous proof of primality? For people unwilling to abandon traditional notions of proof, Gary L. Miller

has demonstrated that if $\sqrt[r]{n}$ is not an integer for any integer $r \geq 2$ (this condition being easily checked), and if a certain well-known conjecture in number theory called the Generalized Riemann Hypothesis can be proved, then either n is prime or there is an $x < 4(\ln n)^2$ such that Algorithm P will discover the nonprimality of n. [See *J. Comp. System Sci.* **13** (1976), 300–317. The constant 4 in this upper bound is due to Peter Weinberger, whose paper on the subject is not yet published.] Thus, we would have a rigorous way to test primality in $O(\log n)^5$ elementary operations, as opposed to a probabilistic method whose running time is $O(\log n)^3$. But one might well ask whether any purported proof of the Generalized Riemann Hypothesis will ever be as reliable as repeated application of Algorithm P on random x's.

A probabilistic test for primality was first proposed in 1974 by R. Solovay and V. Strassen, who devised the interesting but more complicated test described in exercise 23(b). [See *SIAM J. Computing* **6** (1977), 84–85; **7** (1978), 118.] Algorithm P is a simplified version of a procedure due to M. O. Rabin, based in part on ideas of Gary L. Miller [cf. *Algorithms and Complexity*, ed. by J. F. Traub (New York: Academic Press, 1976), 35–36].

A completely different approach to primality testing was discovered in 1980 by Leonard M. Adleman. His highly interesting method is based on the theory of algebraic integers, so it is beyond the scope of this book; but it leads to a non-probabilistic procedure that will decide the primality of any number of up to, say, 250 digits, in a few hours at most. [See L. M. Adleman and R. S. Rumely, to appear.]

Factoring via continued fractions. The factorization procedures we have discussed so far will often balk at numbers of 30 digits or more, and another idea is needed if we are to go much further. Fortunately there is such an idea; in fact, there were two ideas, due respectively to A. M. Legendre and M. Kraitchik, that D. H. Lehmer and R. E. Powers used to devise a new technique many years ago [*Bull. Amer. Math. Soc.* **37** (1931), 770–776]. However, the method was not used at the time because it was comparatively unsuitable for desk calculators. This negative judgment prevailed until the late 1960s, when John Brillhart found that the Lehmer–Powers approach deserved to be resurrected, since it was quite well suited to computer programming. In fact, he and Michael A. Morrison later developed it into the champion of all known methods for factoring large numbers: Their program would handle typical 25-digit numbers in about 30 seconds, and 40-digit numbers in about 50 minutes, on an IBM 360/91 computer [see *Math. Comp.* **29** (1975), 183–205]. In 1970 the method had its first triumphant success, discovering that $2^{128} + 1 = 59649589127497217 \cdot 5704689200685129054721$.

The basic idea is to search for numbers x and y such that

$$x^2 \equiv y^2 \text{ (modulo } N), \qquad 0 < x, y < N, \qquad x \neq y, \qquad x + y \neq N. \tag{17}$$

Fermat's method imposes the stronger requirement $x^2 - y^2 = N$, but actually the congruence (17) is enough to split N into factors: It implies that N is a divisor of $x^2 - y^2 = (x-y)(x+y)$, yet N divides neither $x-y$ nor $x+y$; hence

$\gcd(N, x-y)$ and $\gcd(N, x+y)$ are proper factors of N that can be found by the efficient methods of Section 4.5.2.

One way to discover solutions of (17) is to look for values of x such that $x^2 \equiv a$ (modulo N), for small values of $|a|$. As we will see, it is often a simple matter to piece together solutions of this congruence to obtain solutions of (17). Now if $x^2 = a + kNd^2$ for some k and d, with small $|a|$, the fraction x/d is a good approximation to $\sqrt{kN}$; conversely, if x/d is an especially good approximation to $\sqrt{kN}$, the difference $|x^2 - kNd^2|$ will be small. This observation suggests looking at the continued fraction expansion of $\sqrt{kN}$, since we have seen (in Eq. 4.5.3–12 and exercise 4.5.3–42) that continued fractions yield good rational approximations.

Continued fractions for quadratic irrationalities have many pleasant properties, which are proved in exercise 4.5.3–12. The algorithm below makes use of these properties to derive solutions to the congruence

$$x^2 \equiv (-1)^{e_0} p_1^{e_1} p_2^{e_2} \ldots p_m^{e_m} \pmod{N}. \tag{18}$$

Here we use a fixed set of small primes $p_1 = 2$, $p_2 = 3$, ..., up to p_m; only primes p such that either $p = 2$ or $(kN)^{(p-1)/2} \bmod p \leq 1$ should appear in this list, since other primes will never be factors of the numbers generated by the algorithm (see exercise 14). If $(x_1, e_{01}, e_{11}, \ldots, e_{m1})$, ..., $(x_r, e_{0r}, e_{1r}, \ldots, e_{mr})$ are solutions of (18) such that the vector sum

$$(e_{01}, e_{11}, \ldots, e_{m1}) + \cdots + (e_{0r}, e_{1r}, \ldots, e_{mr}) = (2e'_0, 2e'_1, \ldots, 2e'_m) \tag{19}$$

is even in each component, then

$$x = (x_1 \ldots x_r) \bmod N, \qquad y = \big((-1)^{e'_0} p_1^{e'_1} \ldots p_m^{e'_m}\big) \bmod N \tag{20}$$

yields a solution to (17), except for the possibility that $x \equiv \pm y$. Condition (19) essentially says that the vectors are linearly dependent modulo 2, so we must have a solution to (19) if we have found at least $m + 2$ solutions to (18).

Algorithm E (*Factoring via continued fractions*). Given a positive integer N and a positive integer k such that kN is not a perfect square, this algorithm attempts to discover solutions to the congruence (18) for fixed m, by analyzing the convergents of the continued fraction for $\sqrt{kN}$. (Another algorithm, which uses the outputs to discover factors of N, is the subject of exercise 12.)

E1. [Initialize.] Set $D \leftarrow kN$, $R \leftarrow \lfloor\sqrt{D}\rfloor$, $R' \leftarrow 2R$, $U \leftarrow U' \leftarrow R'$, $V \leftarrow 1$, $V' \leftarrow D - R^2$, $P \leftarrow R$, $P' \leftarrow 1$, $A \leftarrow 0$, $S \leftarrow 0$. (This algorithm follows the general procedure of exercise 4.5.3–12, finding the continued fraction expansion of $\sqrt{kN}$. The variables U, U', V, V', P, P', A, and S represent, respectively, what that exercise calls $R + U_n$, $R + U_{n-1}$, V_n, V_{n-1}, $p_n \bmod N$, $p_{n-1} \bmod N$, A_n, and $n \bmod 2$. We will always have

$$0 < V \leq U \leq R',$$

so the highest precision is needed only for P and P'.)

E2. [Advance U, V, S.] Set $T \leftarrow V$, $V \leftarrow A(U'-U)+V'$, $V' \leftarrow T$, $A \leftarrow \lfloor U/V \rfloor$, $U' \leftarrow U$, $U \leftarrow R' - (U \bmod V)$, $S \leftarrow 1 - S$.

E3. [Factor V.] (Now we have $P^2 - kNQ^2 = (-1)^S V$, for some Q relatively prime to P, by exercise 4.5.3–12(c).) Set $(e_0, e_1, \ldots, e_m) \leftarrow (S, 0, \ldots, 0)$, $T \leftarrow V$. Now do the following, for $1 \le j \le m$: If $T \bmod p_j = 0$, set $T \leftarrow T/p_j$ and $e_j \leftarrow e_j + 1$, and repeat this process until $T \bmod p_j \neq 0$.

E4. [Solution?] If $T = 1$, output the values $(P, e_0, e_1, \ldots, e_m)$, which comprise a solution to (18). (If enough solutions have been generated, we may terminate the algorithm now.)

E5. [Advance P, P'.] If $V \neq 1$, set $T \leftarrow P$, $P \leftarrow (AP + P') \bmod N$, $P' \leftarrow T$, and return to step E2. Otherwise the continued fraction process has started to repeat its cycle, except perhaps for S, so the algorithm terminates. The cycle will usually be so long that this doesn't happen.) ∎

We can illustrate the application of Algorithm E to relatively small numbers by considering the case $N = 197209$, $k = 1$, $m = 3$, $p_1 = 2$, $p_2 = 3$, $p_3 = 5$. The computation proceeds as follows:

	U	V	A	P	S	T	Output
After E1:	888	1	0	444	0	—	
After E4:	876	73	12	444	1	73	
After E4:	882	145	6	5329	0	29	
After E4:	857	37	23	32418	1	37	
After E4:	751	720	1	159316	0	1	$159316^2 \equiv +2^4 \cdot 3^2 \cdot 5^1$
After E4:	852	143	5	191734	1	143	
After E4:	681	215	3	131941	0	43	
After E4:	863	656	1	193139	1	41	
After E4:	883	33	26	127871	0	11	
After E4:	821	136	6	165232	1	17	
After E4:	877	405	2	133218	0	1	$133218^2 \equiv +2^0 \cdot 3^4 \cdot 5^1$
After E4:	875	24	36	37250	1	1	$37250^2 \equiv -2^3 \cdot 3^1 \cdot 5^0$
After E4:	490	477	1	93755	0	53	

Continuing the computation gives 25 outputs in the first 100 iterations; in other words, the algorithm is finding solutions quite rapidly. But some of the solutions are trivial. For example, if the above computation were continued 13 more times, we would obtain the output $197197^2 \equiv 2^4 \cdot 3^2 \cdot 5^0$, which is of no interest since $197197 \equiv -12$. The first two solutions above are already enough to complete the factorization: We have found that

$$(159316 \cdot 133218)^2 \equiv (2^2 \cdot 3^3 \cdot 5^1)^2 \pmod{197209};$$

thus (17) holds with $x = (159316 \cdot 133218) \bmod 197209 = 126308$, $y = 540$. By Euclid's algorithm, $\gcd(126308-540, 197209) = 199$; hence we obtain the pretty factorization

$$197209 = 199 \cdot 991.$$

We can get some understanding of why Algorithm E factors large numbers so successfully by considering a heuristic analysis of its running time, following unpublished ideas that R. Schroeppel communicated to the author in 1975. Let us assume for convenience that $k = 1$. The number of outputs needed to produce a factorization of N will be roughly proportional to the number m of small primes being cast out. Each execution of step E3 takes about order $m \log N$ units of time, so the total running time will be roughly proportional to $m^2 \log N/P$, where P is the probability of a successful output per iteration. If we make the conservative assumption that V is randomly distributed between 0 and $2\sqrt{N}$, the probability P is $(2\sqrt{N})^{-1}$ times the number of integers $< 2\sqrt{N}$ whose prime factors are all in the set $\{p_1, \ldots, p_m\}$. Exercise 29 gives a lower bound for P, from which we conclude that the running time is at most of order

$$\frac{2\sqrt{N}\, m^2 \log N}{m^r/r!}, \qquad \text{where } r = \left\lfloor \frac{\log 2\sqrt{N}}{\log p_m} \right\rfloor. \tag{21}$$

If we let $\ln m = \frac{1}{2}\sqrt{\ln N \ln\ln N}$, we find that $r = \sqrt{\ln N/\ln\ln N} - 1 + O(\log\log\log N/\log\log N)$, assuming that $p_m = O(m \log m)$, so formula (21) reduces to

$$\exp\bigl(2\sqrt{(\ln N)(\ln\ln N)} + O\bigl((\log N)^{1/2}(\log\log N)^{-1/2}(\log\log\log N)\bigr)\bigr).$$

Stating this another way, the running time of Algorithm E is expected to be at most $N^{\epsilon(N)}$ under reasonably plausible assumptions, where the exponent $\epsilon(N) \approx 2\sqrt{\ln\ln N/\ln N}$ goes to 0 as $N \to \infty$.

When N is in a practical range, we should of course be careful not to take such asymptotic estimates too seriously. For example, if $N = 10^{50}$ we have $N^{1/\alpha} = (\lg N)^\alpha$ when $\alpha \approx 4.75$, and the same relation holds for $\alpha \approx 8.42$ when $N = 10^{200}$. The function $N^{\epsilon(N)}$ has an order of growth that is sort of a cross between $N^{1/\alpha}$ and $(\lg N)^\alpha$; but all three of these forms are about the same, unless N is intolerably large. Extensive computational experiments by M. C. Wunderlich have shown that a well-tuned version of Algorithm E performs much better than our estimate would indicate [cf. *Lecture Notes in Math.* **751** (1979), 328–342]; although $2\sqrt{\ln\ln N/\ln N} \approx .41$ when $N = 10^{50}$, he obtained running times of about $N^{.15}$ while factoring thousands of numbers in the range $10^{13} \le N \le 10^{42}$.

Algorithm E begins its attempt to factorize N by essentially replacing N by kN, and this is a rather curious way to proceed (if not downright stupid). Nevertheless, it turns out to be a good idea, since certain values of k will make the V numbers potentially divisible by more small primes, hence they will be more likely to factor completely in step E3. On the other hand, a large value of k will make the V numbers larger, hence they will be less likely to factor completely; we want to balance these tendencies by choosing k wisely. Consider, for example, the divisibility of V by powers of 5. We have $P^2 - kNQ^2 = (-1)^S V$ in step E3, so if 5 divides V we have $P^2 \equiv kNQ^2$ (modulo 5). In this

congruence Q cannot be a multiple of 5, since it is relatively prime to P, so we may write $(P/Q)^2 \equiv kN \pmod 5$. If we assume that P and Q are random relatively prime integers, so that the 24 possible pairs $(P \bmod 5, Q \bmod 5) \neq (0,0)$ are equally likely, the probability that 5 divides V is therefore $\frac{4}{24}$, $\frac{8}{24}$, 0, 0, or $\frac{8}{24}$ according as $kN \bmod 5$ is 0, 1, 2, 3, or 4. Similarly the probability that 25 divides V is 0, $\frac{40}{600}$, 0, 0, $\frac{40}{600}$ respectively, unless kN is a multiple of 25. In general, given an odd prime p with $(kN)^{(p-1)/2} \bmod p = 1$, we find that V is a multiple of p^e with probability $2/(p^{e-1}(p+1))$; and the average number of times p divides V comes to $2p/(p^2-1)$. This analysis, suggested by R. Schroeppel, suggests that the best choice of k is the value that maximizes

$$\sum_{p \text{ prime}} f(p, kN) \log p - \tfrac{1}{2} \log k, \tag{22}$$

where f is the function defined in exercise 28 and the sum is over all primes less than or equal to p_m, for this is essentially the expected value of the logarithm of $\sqrt{N}/T$ when we reach step E4.

Best results will be obtained with Algorithm E when both k and m are well chosen. The proper choice of m can only be made by experimental testing, since the asymptotic analysis we have made is too crude to give sufficiently precise information, and since a variety of refinements to the algorithm tend to have unpredictable effects. For example, we can make an important improvement by comparing step E3 with Algorithm A: The factoring of V can stop whenever we find $T \bmod p_j \neq 0$ and $\lfloor T/p_j \rfloor \leq p_j$, since T will then be either 1 or prime. If T is a prime greater than p_m (it will be at most $p_m^2 + p_m - 1$ in such a case), we can still output $(P, e_0, \ldots, e_m, T)$, since a complete factorization has been obtained. The second phase of the algorithm will use only those outputs whose prime T's have occurred at least twice. This modification gives the effect of a much longer list of primes, without increasing the factorization time. Wunderlich's experiments indicate that $m \approx 150$ works well in the presence of this refinement, when N is in the neighborhood of 10^{40}.

Since step E3 is by far the most time-consuming part of the algorithm, Morrison, Brillhart, and Schroeppel have suggested several ways to abort this step when success becomes improbable: (a) Whenever T changes to a single-precision value, continue only if $\lfloor T/p_j \rfloor > p_j$ and $3^{T-1} \bmod T \neq 1$. (b) Give up if T is still $> p_m^2$ after casting out factors $< \frac{1}{10} p_m$. (c) Cast out factors only up to p_5, say, for batches of 100 or so consecutive V's; continue the factorization later, but only on the V from each batch that has produced the smallest residual T. (Before casting out the factors up to p_5, it is wise to calculate $N \bmod p_1^{f_1} p_2^{f_2} p_3^{f_3} p_4^{f_4} p_5^{f_5}$, where the f's are small enough to make $p_1^{f_1} p_2^{f_2} p_3^{f_3} p_4^{f_4} p_5^{f_5}$ fit in single precision, but large enough to make $N \bmod p_i^{f_i+1} = 0$ unlikely. One single-precision remainder will therefore characterize the value of N modulo five small primes.)

For estimates of the cycle length in the output of Algorithm E, see D. R. Hickerson, *Pacific J. Math.* **46** (1973), 429–432; D. Shanks, *Proc. Boulder Number Theory Conference* (Univ. of Colorado: 1972), 217–224.

Other approaches. A completely different method of factorization, based on composition of binary quadratic forms, has been introduced by Daniel Shanks [*Proc. Symp. Pure Math.* **20** (1971), 415–440]. Like Algorithm B, it will factor a given N in $O(N^{(1/4)+\epsilon})$ steps except under wildly improbable circumstances.

Still another important technique has been suggested by John M. Pollard [*Proc. Cambridge Phil. Soc.* **76** (1974), 521–528]. He shows in essence that each prime factor $\leq m$ can be found in $O(\sqrt{m}(\log mN)^4)$ steps, with a probabilistic algorithm that uses a random x as in Algorithm P and performs suitable convolutions. In this paper Pollard also gives a practical algorithm for discovering prime factors p of N when $p-1$ has no large prime factors. The latter algorithm (see exercise 19) is probably the first thing to try after Algorithms A and B have run too long on a large N.

A survey paper by R. K. Guy, written in collaboration with J. H. Conway, *Proc. Fifth Manitoba Conf. Numer. Math.* (1975), 49–89, gives a unique perspective on these developments.

***A theoretical upper bound.** From the standpoint of computational complexity, we would like to know if there is any method of factorization whose expected running time can be proved to be $O(N^{\epsilon(N)})$, where $\epsilon(N) \to 0$ as $N \to \infty$. We showed that Algorithm E probably has such behavior, but it seems hopeless to find a rigorous proof, because continued fractions are not sufficiently well disciplined. The first proof that a good factorization algorithm exists in this sense was discovered by John Dixon in 1978; Dixon showed, in fact, that it suffices to consider a simplified version of Algorithm E, in which the continued fraction apparatus is removed but the basic idea of (17) remains.

Dixon's method is simply this, assuming that N is known to have at least two distinct prime factors, and that N is not divisible by the first m primes $p_1, p_2, \ldots, p_m$: Choose a random integer X in the range $0 < X < N$, and let $V = X^2 \bmod N$. If $V = 0$, the number $\gcd(X, N)$ is a proper factor of N. Otherwise cast out all of the small prime factors of V as in step E3; in other words, express V in the form

$$V = p_1^{e_1} \ldots p_m^{e_m} T,$$

where T is not divisible by any of the first m primes. If $T = 1$, the algorithm proceeds as in step E4 to output $(X, e_1, \ldots, e_m)$, which represents a solution to (18) with $e_0 = 0$. This process continues with new random values of X until there are sufficiently many outputs to discover a factor of N by the method of exercise 12.

In order to analyze this algorithm, we want to find bounds on (a) the probability that a random X will yield an output, and (b) the probability that a large number of outputs will be required before a factor is found. Let $P(m, N)$ be the probability (a), i.e., the probability that $T = 1$ when X is chosen at random. After M values of X have been tried, we will obtain $MP(m, N)$ outputs, on the average; and the number of outputs has a binomial distribution, so the standard deviation is less than the square root of the mean. The probability (b) is fairly

easy to deal with, since exercise 13 proves that the algorithm needs more than $m + k$ outputs with probability $\le 2^{-k}$.

Exercise 30 proves that $P(m, N) \ge m^r/(r!N)$ when $r = 2\lfloor \log N/(2 \log p_m) \rfloor$, so we can estimate the running time as we did in (21) but with the quantity $2\sqrt{N}$ replaced by N. This time we choose $\ln m = \sqrt{(\ln N \ln\ln N)/2}$, so that

$$r = \sqrt{\frac{2 \ln N}{\ln\ln N}} - 1 + O\left(\frac{\log\log\log N}{\log\log N}\right),$$

$$\frac{m^r}{r!\,N} = \exp\left(-\sqrt{2 \ln N \ln\ln N} + O(r \log\log\log N)\right).$$

We will choose M so that $Mm^r/(r!\,N) \ge 4m$; thus the expected number of outputs $MP(m, N)$ will be at least $4m$. The running time of the algorithm is of order $Mm \log N$, plus $O(m^3)$ steps for exercise 12; it turns out that $O(m^3)$ is less than $Mm \log N$, which is

$$\exp\left(\sqrt{8(\ln N)(\ln\ln N)} + O\big((\log N)^{1/2}(\log\log N)^{-1/2}(\log\log\log N)\big)\right).$$

The probability that this method fails to find a factor is negligibly small, since the probability is at most $e^{-m/2}$ that fewer than $2m$ outputs are obtained (see exercise 31), while the probability is at most 2^{-m} that no factors are found from the first $2m$ outputs, and $m \gg \ln N$. We have proved the following slight strengthening of Dixon's original theorem:

Theorem D. *There is an algorithm whose running time is $O(N^{\epsilon(N)})$, where $\epsilon(N) = c\sqrt{\ln\ln N/\ln N}$ and c is any constant greater than $\sqrt{8}$, that finds a nontrivial factor of N with probability $1 - O(1/N)$, whenever N has at least two distinct prime divisors.* ∎

Secret factors. Worldwide interest in the problem of factorization increased dramatically in 1977, when R. L. Rivest, A. Shamir, and L. Adleman discovered a way to encode messages that can apparently be decoded only by knowing the factors of a large number N, even though the method of encoding is known to everyone. Since a significant number of the world's greatest mathematicians have been unable to find efficient methods of factoring, this scheme [*CACM* **21** (1979), 120–126] almost certainly provides a secure way to protect confidential data and communications in computer networks.

Let us imagine a small electronic device called an *RSA box* that has two large prime numbers p and q stored in its memory. We will assume that $p-1$ and $q-1$ are not divisible by 3. The RSA box is connected somehow to a computer, and it has told the computer the product $N = pq$; however, no human being will be able to discover the values of p and q except by factoring N, since the RSA box is cleverly designed to self-destruct if anybody tries to tamper with it. In other words, it will erase its memory if it is jostled or if it is subjected to any radiation that could change or read out the data stored inside. Furthermore, the RSA box is sufficiently reliable that it never needs to be maintained; we simply

would discard it and buy another, if an emergency arose or if it wore out. The prime factors p and q were generated by the RSA box itself, using some scheme based on truly random phenomena in nature like cosmic rays. The important point is that *nobody* knows p or q, not even a person or organization that owns or has access to this RSA box; there is no point in bribing or blackmailing anyone or holding anybody hostage in order to discover N's factors.

To send a secret message to the owner of an RSA box whose product number is N, you break the message up into a sequence of numbers $(x_1, \ldots, x_k)$, where each x_i is between 0 and N, then you transmit the numbers

$$(x_1^3 \bmod N,\ \ldots,\ x_k^3 \bmod N).$$

The RSA box, knowing p and q, can decode the message, because it has precomputed a number $d < N$ such that $3d \equiv 1$ (modulo $(p-1)(q-1)$); it can now compute $(x^3)^d \bmod N = x$ in a reasonable amount of time, using the method of Section 4.6.3. Naturally the RSA box keeps this magic number d to itself; in fact, the RSA box might choose to remember only d instead of p and q, because its only duties after having computed N are to protect its secrets and to take cube roots mod N.

If $x < \sqrt[3]{N}$, the above encoding scheme is ineffective, since $x^3 \bmod N = x^3$ and the cube root will easily be found. The logarithmic law of leading digits in Section 4.2.4 implies that the leading place x_1 of a k-place message $(x_1, \ldots, x_k)$ will be less than $\sqrt[3]{N}$ about $\frac{1}{3}$ of the time, so this is a problem that needs to be resolved. Exercise 32 presents one way to do this.

The security of the RSA encoding scheme relies on the fact that nobody has been able to discover how to take cube roots mod N without knowing N's factors. It seems likely that no such method will be found, but we cannot be absolutely sure. So far all that can be said for certain is that all of the ordinary ways to discover cube roots will fail. For example, there is essentially no point in trying to compute the number d as a function of N; the reason is that if d is known, or in fact if any number m of reasonable size is known such that $x^m \bmod N = 1$ holds for a significant number of x's, then we can find the factors of N in a few more steps (see exercise 34). Thus, any method of attack based explicitly or implicitly on finding such an m can be no better than factoring.

The numbers p and q shouldn't merely be "random" primes in order to make the RSA scheme effective. We have mentioned that $p-1$ and $q-1$ should not be divisible by 3, since we want to ensure that unique cube roots exist modulo N. Another condition is that $p-1$ should have at least one very large prime factor, and so should $q-1$; otherwise N can be factored using the algorithm of exercise 19. In fact, that algorithm essentially relies on finding a fairly small number m with the property that $x^m \bmod N$ is frequently equal to 1, and we have just seen that such an m is dangerous. When $p-1$ and $q-1$ have large prime factors p_1 and q_1, the theory in exercise 34 implies that m is either a multiple of $p_1 q_1$ (hence m will be hard to discover) or the probability that $x^m \equiv 1$ will be less than $1/p_1 q_1$ (hence $x^m \bmod N$ will almost never be 1).

Besides this condition, we don't want p and q to be close to each other, lest Algorithm D succeed in discovering them; in fact, we don't want the ratio p/q to be near a simple fraction, otherwise Lehman's generalization of Algorithm C could find them.

The following procedure for generating p and q is almost surely unbreakable: Start with a truly random number p_0 between, say, 10^{80} and 10^{81}. Search for the first prime number p_1 greater than p_0; this will require testing about $\frac{1}{2}\ln p_0 \approx 90$ odd numbers, and it will be sufficient to have p_1 a "probable prime" with probability $> 1 - 2^{-100}$ after 50 trials of Algorithm P. Then choose another truly random number p_2 between, say, 10^{39} and 10^{40}. Search for the first prime number p of the form $kp_1 + 1$ where $k \geq p_2$, k is even, and $k \equiv p_1$ (modulo 3). This will require testing about $\frac{1}{6}\ln p_1 p_2 \approx 45$ numbers before a prime p is found. The prime p will be about 120 digits long; a similar construction can be used to find a prime q about 130 digits long. For extra security, it is probably advisable to check that neither $p + 1$ nor $q + 1$ consists entirely of rather small prime factors (see exercise 20). The product $N = pq$, whose order of magnitude will be about 10^{250}, now meets all of our requirements, and it is inconceivable at this time that such an N could be factored.

For example, suppose we knew a method that could factor a 250-digit number N in $N^{0.1}$ microseconds. This amounts to 10^{25} microseconds, and there are only 31,556,952,000,000 μs per year, so we would need more than 3×10^{11} years of CPU time to complete the factorization. Even if a government agency purchased 10 billion computers and set them all to working on this problem, it would take more than 31 years before one of them would crack N into factors; meanwhile the fact that the government had purchased so many specialized machines would leak out, and people would start using 300-digit N's.

Since the encoding method $x \mapsto x^3 \bmod N$ is known to everyone, there are additional advantages besides the fact that the code can be cracked only by the RSA box. Such "public key" systems were first considered by W. Diffie and M. E. Hellman in *IEEE Trans.* **IT-22** (1976), 644–651. As an example of what can be done when the encoding method is public knowledge, suppose that Alice wants to communicate with Bill via electronic mail, and suppose each of them wants the letters to be *signed* so that the receiver can be sure that nobody else is forging any messages. Let $E_A(M)$ be the encoding function for messages M sent to Alice, let $D_A(M)$ be the decoding done by Alice's RSA box, and let $E_B(M)$, $D_B(M)$ be the corresponding encoding and decoding functions for Bill's RSA box. Then Alice can send a signed message by affixing her name and the date to some confidential message, then transmitting $E_B\big(D_A(M)\big)$ to Bill, using her machine to compute $D_A(M)$. When Bill gets this message, his RSA box converts it to $D_A(M)$, and he knows E_A so he can compute $M = E_A\big(D_A(M)\big)$. This should convince him that the message did indeed come from Alice; nobody else could have sent the message $D_A(M)$.

We might ask, how do Alice and Bill know each other's encoding functions E_A and E_B? It wouldn't do simply to have them stored in a public file, since some Charlie could tamper with that file, substituting an N that he has computed

by himself; Charlie could then surreptitiously intercept and decode a private message before Alice or Bill would discover that something is amiss. The solution is to keep the product numbers N_A and N_B in a special public directory that has its own RSA box and its own widely publicized product number N_D. When Alice wants to know how to communicate with Bill, she asks the directory for Bill's product number; the directory computer sends her a *signed* message giving the value of N_B. Nobody can forge such a message, so it must be legitimate.

An interesting alternative to the RSA scheme has been proposed by Michael Rabin [MIT Lab. for Comp. Sci., report TR-212 (1979)], who suggests encoding by the function $x^2 \bmod N$ instead of $x^3 \bmod N$. In this case the decoding mechanism, which we can call a SQRT box, returns four different messages; the reason is that four different numbers have the same square modulo N, namely x, $-x$, $fx \bmod N$, and $(-fx) \bmod N$, where $f = (p^{q-1} - q^{p-1}) \bmod N$. If we agree in advance that x is even, or that $x < \frac{1}{2}N$, then the ambiguity drops to two messages, presumably only one of which makes any sense. The ambiguity can in fact be eliminated entirely, as shown in exercise 35. Rabin's scheme has the important property that it is provably as difficult to find square roots mod N as to find the factorization $N = pq$; for by taking the square root of $x^2 \bmod N$ when x is chosen at random, we have probability $\frac{1}{2}$ of finding a value y such that $x^2 \equiv y^2$ and $x \not\equiv \pm y$, after which $\gcd(x, y) = p$ or q. However, the system has a fatal flaw that does not seem to be present in the RSA scheme (see exercise 33): Anyone with access to a SQRT box can easily determine the factors of its N. This not only permits cheating by dishonest employees, or threats of extortion, it also allows people to reveal their p and q, after which they might claim that their "signature" on some transmitted document was a forgery. Thus it is clear that the goal of secure communication leads to subtle problems quite different from those we usually face in the design and analysis of algorithms.

The largest known primes. We have discussed several computational methods elsewhere in this book that require the use of large prime numbers, and the techniques just described can be used to discover primes of up to, say, 25 digits or fewer, with relative ease. Table 1 shows the ten largest primes that are less than the word size of typical computers. (Some other useful primes appear in the answer to exercise 4.6.4–57.)

Actually much larger primes of special forms are known, and it is occasionally important to find primes that are as large as possible. Let us therefore conclude this section by investigating the interesting manner in which the largest explicitly known primes have been discovered. Such primes are of the form $2^n - 1$, for various special values of n, and so they are especially suited to certain applications of binary computers.

A number of the form $2^n - 1$ cannot be prime unless n is prime, since $2^{uv} - 1$ is divisible by $2^u - 1$. In 1644, Marin Mersenne astonished his contemporaries by stating, in essence, that the numbers $2^p - 1$ are prime for $p = 2$, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257, and for no other p less than 257. (This statement appeared in connection with a discussion of perfect numbers in the preface to his *Cogitata*

Table 1

USEFUL PRIME NUMBERS

N	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
2^{15}	19	49	51	55	61	75	81	115	121	135
2^{16}	15	17	39	57	87	89	99	113	117	123
2^{17}	1	9	13	31	49	61	63	85	91	99
2^{18}	5	11	17	23	33	35	41	65	75	93
2^{19}	1	19	27	31	45	57	67	69	85	87
2^{20}	3	5	17	27	59	69	129	143	153	185
2^{21}	9	19	21	55	61	69	105	111	121	129
2^{22}	3	17	27	33	57	87	105	113	117	123
2^{23}	15	21	27	37	61	69	135	147	157	159
2^{24}	3	17	33	63	75	77	89	95	117	167
2^{25}	39	49	61	85	91	115	141	159	165	183
2^{26}	5	27	45	87	101	107	111	117	125	135
2^{27}	39	79	111	115	135	187	199	219	231	235
2^{28}	57	89	95	119	125	143	165	183	213	273
2^{29}	3	33	43	63	73	75	93	99	121	133
2^{30}	35	41	83	101	105	107	135	153	161	173
2^{31}	1	19	61	69	85	99	105	151	159	171
2^{32}	5	17	65	99	107	135	153	185	209	267
2^{33}	9	25	49	79	105	285	301	303	321	355
2^{34}	41	77	113	131	143	165	185	207	227	281
2^{35}	31	49	61	69	79	121	141	247	309	325
2^{36}	5	17	23	65	117	137	159	173	189	233
2^{37}	25	31	45	69	123	141	199	201	351	375
2^{38}	45	87	107	131	153	185	191	227	231	257
2^{39}	7	19	67	91	135	165	219	231	241	301
2^{40}	87	167	195	203	213	285	293	299	389	437
2^{41}	21	31	55	63	73	75	91	111	133	139
2^{42}	11	17	33	53	65	143	161	165	215	227
2^{43}	57	67	117	175	255	267	291	309	319	369
2^{44}	17	117	119	129	143	149	287	327	359	377
2^{45}	55	69	81	93	121	133	139	159	193	229
2^{46}	21	57	63	77	167	197	237	287	305	311
2^{47}	115	127	147	279	297	339	435	541	619	649
2^{48}	59	65	89	93	147	165	189	233	243	257
2^{59}	55	99	225	427	517	607	649	687	861	871
2^{60}	93	107	173	179	257	279	369	395	399	453
2^{63}	25	165	259	301	375	387	391	409	457	471
2^{64}	59	83	95	179	189	257	279	323	353	363
10^6	17	21	39	41	47	69	83	93	117	137
10^7	9	27	29	57	63	69	71	93	99	111
10^8	11	29	41	59	69	153	161	173	179	213
10^9	63	71	107	117	203	239	243	249	261	267
10^{10}	33	57	71	119	149	167	183	213	219	231
10^{11}	23	53	57	93	129	149	167	171	179	231
10^{12}	11	39	41	63	101	123	137	143	153	233
10^{16}	63	83	113	149	183	191	329	357	359	369

The ten largest primes less than N are $N - a_1, \ldots, N - a_{10}$.

Physico-Mathematics. Curiously, he also made the following remark: "To tell if a given number of 15 or 20 digits is prime or not, all time would not suffice for the test, whatever use is made of what is already known.") Mersenne, who had corresponded frequently with Fermat, Descartes, and others about similar topics in previous years, gave no proof of his assertions, and for over 200 years nobody knew whether he was correct or not. Euler showed that $2^{31} - 1$ is prime in 1772, after having tried unsuccessfully to prove this in previous years. About 100 years later, É. Lucas discovered that $2^{127} - 1$ is prime, but $2^{67} - 1$ is not; therefore Mersenne was not completely accurate. Then I. M. Pervushin proved in 1883 that $2^{61} - 1$ is prime [cf. *Istoriko-Mat. Issledovaniĭa* **6** (1953), 559], and this touched off speculation that Mersenne had only made a copying error, writing 67 for 61. Eventually other errors in Mersenne's statement were discovered; R. E. Powers [*AMM* **18** (1911), 195] found that $2^{89} - 1$ is prime, as had been conjectured by some earlier writers, and three years later he proved that $2^{107} - 1$ also is prime. M. Kraitchik showed in 1922 that $2^{257} - 1$ is *not* prime [cf. *Recherches sur la Théorie des Nombres* (Paris: 1924), 21].

At any rate, numbers of the form $2^p - 1$ are now called *Mersenne numbers*, and it is known that the first 27 Mersenne primes are obtained for p equal to

$$\begin{aligned} &2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203\,2281, \\ &3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, \end{aligned} \tag{23}$$

The 24th of these was found by Bryant Tuckerman [*Proc. Nat. Acad. Sci.* **68** (1971), 2319–2320], and the 25th was found in 1978 by Laura Nickel and Curt Noll. Shortly afterwards, Noll found the 26th, and David Slowinski harnessed a CRAY-I computer to the task of discovering the 27th; see *J. Recreational Math.* **11** (1979), 258–261. Note that the prime $8191 = 2^{13} - 1$ does not occur in (23); Mersenne had stated that $2^{8191} - 1$ is prime, and others had conjectured that any Mersenne prime could perhaps be used in the exponent.

Since $2^{44497} - 1$ is a 13,395-digit number, it is clear that some special techniques have been used to prove that it is prime. An efficient way to test the primality of a given Mersenne number $2^p - 1$ was first devised by É. Lucas [*Amer. J. Math.* **1** (1878), 184–239, 289–321, especially p. 316] and improved by D. H. Lehmer [*Annals of Math.* **31** (1930), 419–448, especially p. 443]. The Lucas–Lehmer test, which is a special case of the method now used for testing the primality of n when the factors of $n + 1$ are known, is the following:

Theorem L. *Let q be an odd prime, and define the sequence $\langle L_n \rangle$ by the rule*

$$L_0 = 4, \qquad L_{n+1} = (L_n^2 - 2) \bmod (2^q - 1). \tag{24}$$

Then $2^q - 1$ is prime if and only if $L_{q-2} = 0$.

For example, $2^3 - 1$ is prime since $L_1 = (4^2 - 2) \bmod 7 = 0$. This test is particularly well suited to binary computers, using multiple-precision arithmetic when q is large, since calculation mod $(2^q - 1)$ is so convenient; cf. Section 4.3.2.

Proof. We will prove Theorem L using only very simple principles of number theory, by investigating several features of recurring sequences that are of independent interest. Consider the sequences $\langle U_n \rangle$ and $\langle V_n \rangle$ defined by

$$\begin{aligned} U_0 &= 0, \quad U_1 = 1, \quad U_{n+1} = 4U_n - U_{n-1}; \\ V_0 &= 2, \quad V_1 = 4, \quad V_{n+1} = 4V_n - V_{n-1}. \end{aligned} \tag{25}$$

The following equations are readily proved by induction:

$$V_n = U_{n+1} - U_{n-1}; \tag{26}$$

$$U_n = \big((2+\sqrt{3})^n - (2-\sqrt{3})^n\big)/\sqrt{12}; \tag{27}$$

$$V_n = (2+\sqrt{3})^n + (2-\sqrt{3})^n; \tag{28}$$

$$U_{m+n} = U_m U_{n+1} - U_{m-1} U_n. \tag{29}$$

Let us now prove an auxiliary result, when p is prime and $e \geq 1$:

$$\text{if} \quad U_n \equiv 0 \text{ (modulo } p^e) \quad \text{then} \quad U_{np} \equiv 0 \text{ (modulo } p^{e+1}). \tag{30}$$

This follows from the more general considerations of exercise 3.2.2–11, but a direct proof for sequence (25) can be given. Assume that $U_n = bp^e$, $U_{n+1} = a$. By (29) and (25), $U_{2n} = bp^e(2a - 4bp^e) \equiv (2a)U_n$ (modulo p^{e+1}), while we have $U_{2n+1} = U_{n+1}^2 - U_n^2 \equiv a^2$. Similarly, $U_{3n} = U_{2n+1}U_n - U_{2n}U_{n-1} \equiv (3a^2)U_n$ and $U_{3n+1} = U_{2n+1}U_{n+1} - U_{2n}U_n \equiv a^3$. In general,

$$U_{kn} \equiv (ka^{k-1})U_n \quad \text{and} \quad U_{kn+1} \equiv a^k \quad \text{(modulo } p^{e+1}),$$

so (30) follows if we take $k = p$.

From formulas (27) and (28) we can obtain other expressions for U_n and V_n, expanding $(2 \pm \sqrt{3})^n$ by the binomial theorem:

$$U_n = \sum_k \binom{n}{2k+1} 2^{n-2k-1} 3^k, \qquad V_n = \sum_k \binom{n}{2k} 2^{n-2k+1} 3^k. \tag{31}$$

Now if we set $n = p$, where p is an odd prime, and if we use the fact that $\binom{p}{k}$ is a multiple of p except when $k = 0$ or $k = p$, we find that

$$U_p \equiv 3^{(p-1)/2}, \qquad V_p \equiv 4 \qquad \text{(modulo } p). \tag{32}$$

If $p \neq 3$, Fermat's theorem tells us that $3^{p-1} \equiv 1$; hence $(3^{(p-1)/2} - 1) \times (3^{(p-1)/2} + 1) \equiv 0$, and $3^{(p-1)/2} \equiv \pm 1$. When $U_p \equiv -1$, we have $U_{p+1} = 4U_p - U_{p-1} = 4U_p + V_p - U_{p+1} \equiv -U_{p+1}$; hence $U_{p+1} \bmod p = 0$. When $U_p \equiv +1$, we have $U_{p-1} = 4U_p - U_{p+1} = 4U_p - V_p - U_{p-1} \equiv -U_{p-1}$; hence $U_{p-1} \bmod p = 0$. We have proved that, for all primes p, there is an integer $\epsilon(p)$ such that

$$U_{p+\epsilon(p)} \bmod p = 0, \qquad |\epsilon(p)| \leq 1. \tag{33}$$

Now if N is any positive integer, and if $m = m(N)$ is the smallest positive integer such that $U_{m(N)} \bmod N = 0$, we have

$$U_n \bmod N = 0 \qquad \text{if and only if} \qquad n \text{ is a multiple of } m(N). \tag{34}$$

(This number $m(N)$ is called the "rank of apparition" of N in the sequence.) To prove (34), observe that the sequence U_m, U_{m+1}, U_{m+2}, ... is congruent (modulo N) to aU_0, aU_1, aU_2, ..., where $a = U_{m+1} \bmod N$ is relatively prime to N because $\gcd(U_n, U_{n+1}) = 1$.

With these preliminaries out of the way, we are ready to prove Theorem L. By (24) and induction,

$$L_n = V_{2^n} \bmod (2^q - 1). \tag{35}$$

Furthermore, the identity $2U_{n+1} = 4U_n + V_n$ implies that $\gcd(U_n, V_n) \le 2$, since any common factor of U_n and V_n must divide U_n and $2U_{n+1}$, while $\gcd(U_n, U_{n+1}) = 1$. So U_n and V_n have no odd factor in common, and if $L_{q-2} = 0$ we must have

$$U_{2^{q-1}} = U_{2^{q-2}} V_{2^{q-2}} \equiv 0 \pmod{2^q - 1},$$
$$U_{2^{q-2}} \not\equiv 0 \pmod{2^q - 1}.$$

Now if $m = m(2^q - 1)$ is the rank of apparition of $2^q - 1$, it must be a divisor of 2^{q-1} but not of 2^{q-2}; thus $m = 2^{q-1}$. We will prove that $n = 2^q - 1$ must therefore be prime: Let the factorization of n be $p_1^{e_1} \ldots p_r^{e_r}$. All primes p_j are greater than 3, since n is odd and congruent to $(-1)^q - 1 = -2$ (modulo 3). From (30), (33), and (34) we know that $U_t \equiv 0$ (modulo $2^q - 1$), where

$$t = \operatorname{lcm}\bigl(p_1^{e_1-1}(p_1 + \epsilon_1), \ldots, p_r^{e_r-1}(p_r + \epsilon_r)\bigr),$$

and each ϵ_j is ± 1. It follows that t is a multiple of $m = 2^{q-1}$. Let $n_0 = \prod_{1 \le j \le r} p_j^{e_j-1}(p_j + \epsilon_j)$; we have $n_0 \le \prod_{1 \le j \le r} p_j^{e_j-1}(p_j + \frac{1}{5}p_j) = (\frac{6}{5})^r n$. Also, because $p_j + \epsilon_j$ is even, $t \le n_0/2^{r-1}$, since a factor of two is lost each time the least common multiple of two even numbers is taken. Combining these results, we have $m \le t \le 2(\frac{3}{5})^r n < 4(\frac{3}{5})^r m < 3m$; hence $r \le 2$ and $t = m$ or $t = 2m$, a power of 2. Therefore $e_1 = 1$, $e_r = 1$, and if n is not prime we must have $n = 2^q - 1 = (2^k + 1)(2^l - 1)$ where $2^k + 1$ and $2^l - 1$ are prime. The latter factorization is obviously impossible when q is odd, so n is prime.

Conversely, suppose that $n = 2^q - 1$ is prime; we must show that $V_{2^{q-2}} \equiv 0$ (modulo n). For this purpose it suffices to prove that $V_{2^{q-1}} \equiv -2$ (modulo n), since $V_{2^{q-1}} = (V_{2^{q-2}})^2 - 2$. Now

$$V_{2^{q-1}} = \bigl((\sqrt{2} + \sqrt{6})/2\bigr)^{n+1} + \bigl((\sqrt{2} - \sqrt{6})/2\bigr)^{n+1}$$
$$= 2^{-n} \sum_k \binom{n+1}{2k} \sqrt{2}^{\,n+1-2k} \sqrt{6}^{\,2k} = 2^{(1-n)/2} \sum_k \binom{n+1}{2k} 3^k.$$

Since n is prime, the binomial coefficient

$$\binom{n+1}{2k} = \binom{n}{2k} + \binom{n}{2k-1}$$

is divisible by n except when $k = 0$ and $k = (n+1)/2$; hence

$$2^{(n-1)/2} V_{2q-1} \equiv 1 + 3^{(n+1)/2} \text{ (modulo } n).$$

Here $2 \equiv (2^{(q+1)/2})^2$, so $2^{(n-1)/2} \equiv (2^{(q+1)/2})^{(n-1)} \equiv 1$ by Fermat's theorem. Finally, by a simple case of the law of quadratic reciprocity (cf. exercise 23), $3^{(n-1)/2} \equiv -1$, since $n \bmod 3 = 1$ and $n \bmod 4 = 3$. This means $V_{2q-1} \equiv -2$, so $V_{2q-2} \equiv 0$. ∎

The world's largest explicitly known prime numbers have always been Mersenne primes, at least from 1772 until 1980 when this book was written. But the situation will probably change soon, since Mersenne primes are getting harder to find, and since exercise 27 presents an efficient test for primes of other forms.

EXERCISES

1. [*10*] If the sequence d_0, d_1, d_2, ... of trial divisors in Algorithm A contains a number that is not prime, why will it never appear in the output?

2. [*15*] If it is known that the input N to Algorithm A is equal to 3 or more, could step A2 be eliminated?

3. [*M20*] Show that there is a number P with the following property: If $1000 \le n \le 1000000$, then n is prime if and only if $\gcd(n, P) = 1$.

4. [*M29*] In the notation of exercise 3.1–7 and Section 1.2.11.3, prove that the average value of the least n such that $X_n = X_{l(n)-1}$ lies between $1.5\,Q(m) - 0.5$ and $1.625\,Q(m) - 0.5$.

5. [*21*] Use Fermat's method (Algorithm D) to find the factors of 10541 by hand, when the moduli are 3, 5, 7, and 8.

6. [*M24*] If p is an odd prime and if N is not a multiple of p, prove that the number of integers x such that $0 \le x < p$ and $x^2 - N \equiv y^2$ (modulo p) has a solution y is equal to $(p \pm 1)/2$.

7. [*25*] Discuss the problems of programming the sieve of Algorithm D on a binary computer when the table entries for modulus m_i do not exactly fill an integral number of memory words.

▶ **8.** [*23*] (*The "sieve of Eratosthenes,"* 3rd century B.C.) The following procedure evidently discovers all odd prime numbers less than a given integer N, since it removes all the nonprime numbers: Start with all the odd numbers less than N; then successively strike out the multiples p_k^2, $p_k(p_k+2)$, $p_k(p_k+4)$, ..., of the kth prime p_k, for $k = 2$, 3, 4, ..., until reaching a prime p_k with $p_k^2 > N$.

Show how to adapt the procedure just described into an algorithm that is directly suited to efficient computer calculation, using no multiplication.

9. [*M25*] Let n be an odd number, $n \geq 3$. Show that if the number $\lambda(n)$ of Theorem 3.2.1.2B is a divisor of $n-1$ but not equal to $n-1$, then n must have the form $p_1 p_2 \ldots p_t$ where the p's are distinct primes and $t \geq 3$.

▶ **10.** [*M26*] (John Selfridge.) Prove that if, for each prime divisor p of $n-1$, there is a number x_p such that $x_p^{(n-1)/p} \bmod n \neq 1$ but $x_p^{n-1} \bmod n = 1$, then n is prime.

11. [*M20*] What outputs does Algorithm E give when $N = 197209$, $k = 5$, $m = 1$? [*Hint:* $\sqrt{5 \cdot 197209} = 992 + /\overline{1, 495, 2, 495, 1, 1984}/$.]

▶ **12.** [*M28*] Design an algorithm that uses the outputs of Algorithm E to find a proper factor of N, provided that Algorithm E has produced enough outputs to deduce a solution of (17).

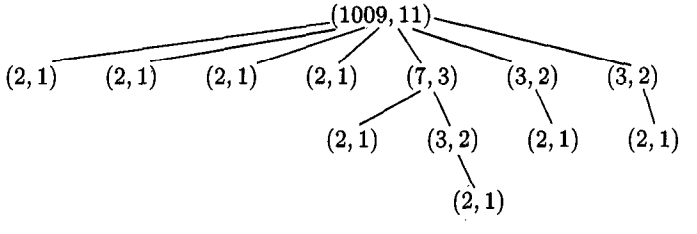
13. [*HM25*] (J. D. Dixon.) Prove that whenever the algorithm of exercise 12 is presented with a solution $(x, e_0, \ldots, e_m)$ whose exponents are linearly dependent modulo 2 on the exponents of previous solutions, the probability is 2^{1-d} that a factorization will be found, when n has d distinct prime factors and x is chosen at random.

14. [*M20*] Prove that the number T in step E3 of Algorithm E will never be a multiple of an odd prime p for which $(kN)^{(p-1)/2} \bmod p > 1$.

▶ **15.** [*M34*] (Lucas and Lehmer.) Let P and Q be relatively prime integers, and let $U_0 = 0$, $U_1 = 1$, $U_{n+1} = PU_n - QU_{n-1}$ for $n \geq 1$. Prove that if N is a positive integer relatively prime to $2P^2 - 8Q$, and if $U_{N+1} \bmod N = 0$, while $U_{(N+1)/p} \bmod N \neq 0$ for each prime p dividing $N+1$, then N is prime. (This gives a test for primality when the factors of $N+1$ are known instead of the factors of $N-1$. We can evaluate U_m in $O(\log m)$ steps; cf. exercise 4.6.3–26.) [*Hint:* See the proof of Theorem L.]

16. [*M50*] Are there infinitely many Mersenne primes?

17. [*M25*] (V. R. Pratt.) A complete proof of primality by the converse of Fermat's theorem takes the form of a tree whose nodes have the form (q, x), where q and x are positive integers satisfying the following arithmetic conditions: (i) If $(q_1, x_1), \ldots, (q_t, x_t)$ are the sons of (q, x) then $q = q_1 \ldots q_t + 1$. [In particular, if (q, x) has no sons, then $q = 2$.] (ii) If (r, y) is a son of (q, x), then $x^{(q-1)/r} \bmod q \neq 1$. (iii) For each node (q, x), we have $x^{q-1} \bmod q = 1$. From these conditions it follows that q is prime and x is a primitive root modulo q, for all nodes (q, x). [For example, the tree



demonstrates that 1009 is prime.] Prove that such a tree with root (q, x) has at most $f(q)$ nodes, where f is a rather slowly growing function.

▶ **18.** [*HM23*] Give a heuristic proof of (7), analogous to the text's derivation of (6). What is the approximate probability that $p_{t-1} \leq \sqrt{p_t}$?

▸ **19.** [*M25*] (J. M. Pollard.) Show how to compute a number M that is divisible by all primes p such that $p - 1$ is a divisor of some given number D. [*Hint:* Consider numbers of the form $a^n - 1$.] Such an M is useful in factorization, for by computing $\gcd(M, N)$ we may discover a factor of N. Extend this idea to an efficient method that has high probability of discovering prime factors p of a given large number N, when all prime power factors of $p - 1$ are less than 10^3 except for at most one prime factor less than 10^5. [For example, the second-largest prime dividing (14) would be detected by this method, since it is $1 + 2^4 \cdot 5^2 \cdot 67 \cdot 107 \cdot 199 \cdot 41231$.]

20. [*M40*] Consider exercise 19 with $p + 1$ replacing $p - 1$.

21. [*M49*] (R. K. Guy.) Let $m(p)$ be the number of iterations required by Algorithm B to cast out the prime factor p. Is $m(p) = O(\sqrt{p \log p})$ as $p \to \infty$?

▸ **22.** [*M30*] (M. O. Rabin.) Let p_n be the probability that Algorithm P guesses wrong, given n. Show that $p_n < \frac{1}{4}$ for all n.

23. [*M35*] The *Jacobi symbol* $(\frac{p}{q})$ is defined to be -1, 0, or $+1$ for all integers $p \geq 0$ and all odd integers $q > 1$ by the rules $(\frac{p}{q}) \equiv p^{(q-1)/2}$ (modulo q) when q is prime; $(\frac{p}{q}) = (\frac{p}{q_1}) \ldots (\frac{p}{q_t})$ when q is the product $q_1 \ldots q_t$ of t primes (not necessarily distinct).

a) Prove that $(\frac{p}{q})$ satisfies the following relationships, hence it can be computed efficiently: $(\frac{0}{q}) = 0$; $(\frac{1}{q}) = 1$; $(\frac{p}{q}) = (\frac{p \bmod q}{q})$; $(\frac{2}{q}) = (-1)^{(q^2-1)/8}$; $(\frac{p_1 p_2}{q}) = (\frac{p_1}{q})(\frac{p_2}{q})$; $(\frac{p}{q}) = (-1)^{(p-1)(q-1)/4}(\frac{q}{p})$ if both p and q are odd. [The latter law, which is a reciprocity relation reducing the evaluation of $(\frac{p}{q})$ to the evaluation of $(\frac{q}{p})$, has been proved in exercise 1.2.4–47(d) when both p and q are prime, so you may assume its validity in that special case.]

b) (Solovay and Strassen.) Prove that if n is odd but not prime, the number of integers x such that $1 \leq x < n$ and $0 \neq (\frac{x}{n}) \equiv x^{(n-1)/2}$ (modulo n) is at most $\frac{1}{2}\varphi(n)$. (Thus, the following testing procedure correctly determines whether or not a given n is prime, with probability $\geq \frac{1}{2}$ for all fixed n: "Generate x at random with $1 \leq x < n$. If $0 \neq (\frac{x}{n}) \equiv x^{(n-1)/2}$ (modulo n), say that n is probably prime, otherwise say that n is definitely not prime.")

c) (L. Monier.) Prove that if n and x are numbers for which Algorithm P concludes that "n is probably prime", then $0 \neq (\frac{x}{n}) \equiv x^{(n-1)/2}$ (modulo n). [Hence Algorithm P is always superior to the test in (b).]

▸ **24.** [*M25*] (L. Adleman.) When $n > 1$ and $x > 1$ are integers, n odd, let us say that n "passes the x test of Algorithm P" if either $x = n$ or if steps P2–P5 lead to the conclusion that n is probably prime. Prove that, for any N, there exists a set of positive integers $x_1, \ldots, x_m \leq N$ with $m \leq \lfloor \lg N \rfloor$ such that a positive odd integer in the range $1 < n \leq N$ is prime if and only if it passes the x test of Algorithm P for $x = x_1 \bmod n, \ldots, x = x_m \bmod n$. Thus, the probabilistic test for primality can in principle be converted into an efficient test that leaves nothing to chance. (You need not show how to compute the x_j efficiently; just prove that they exist.)

25. [*HM41*] (B. Riemann.) Prove that

$$\pi(x) = \int_2^x dt/\ln t + \sum \int_{-\infty}^{\sigma} e^{(t+i\tau)\ln x} dt/(t + i\tau) + O(1),$$

where the sum is over all complex $\sigma + i\tau$ such that $\tau \neq 0$ and $\zeta(\sigma + i\tau) = 0$.

▶ **26.** [*M25*] (H. C. Pocklington, 1914.) Let $N = fr + 1 > 1$, where $0 < r \leq f + 1$. Prove that N is prime if, for every prime divisor p of f, there is an integer x such that $x^{N-1} \bmod N = \gcd(x^{(N-1)/p} - 1, N) = 1$.

▶ **27.** [*M30*] Show that there is a way to test numbers of the form $5 \cdot 2^n + 1$ for primality, using the same amount of computer time as the Lucas–Lehmer test for Mersenne primes in Theorem L, except for an additional $O(n \log n)$ seconds.

28. [*M27*] Given a prime p and a positive integer d, what is the value of $f(p, d)$, the average number of times p divides $A^2 - dB^2$, when A and B are random integers that are independent except for the condition $\gcd(A, B) = 1$?

29. [*M25*] Prove that the number of positive integers $\leq n$ whose prime factors are all contained in a given set of primes $\{p_1, \ldots, p_m\}$ is at least $m^r/r!$, when $r = \lfloor \log n / \log p_m \rfloor$ and $p_1 < \cdots < p_m$.

30. [*HM35*] (J. D. Dixon and Claus-Peter Schnorr.) Let $p_1 < \cdots < p_m$ be primes that do not divide the odd number N, and let r be an even integer $\leq \log N / \log p_m$. Prove that the number of integers X in the range $0 \leq X < N$ such that $X^2 \bmod N = p_1^{e_1} \ldots p_m^{e_m}$ is at least $m^r/r!$. [*Hint:* Let the prime factorization of N be $q_1^{f_1} \ldots q_d^{f_d}$. Show that a sequence of exponents $(e_1, \ldots, e_m)$ leads to 2^d solutions X whenever we have $e_1 + \cdots + e_m \leq r$ and $p_1^{e_1} \ldots p_m^{e_m}$ is a quadratic residue modulo q_i for $1 \leq i \leq d$. Such exponent sequences can be obtained as ordered pairs $(e'_1, \ldots, e'_m; e''_1, \ldots, e''_m)$ where $e'_1 + \cdots + e'_m \leq \frac{1}{2}r$ and $e''_1 + \cdots + e''_m \leq \frac{1}{2}r$ and

$$(p_1^{e'_1} \ldots p_m^{e'_m})^{(q_i-1)/2} \equiv (p_1^{e''_1} \ldots p_m^{e''_m})^{(q_i-1)/2} \text{ (modulo } q_i)$$

for $1 \leq i \leq d$.]

31. [*M20*] Use formula 3.5–33 to show that the probability is less than $e^{-m/2}$ that Dixon's factorization algorithm (as described preceding Theorem D) obtains fewer than $2m$ outputs.

▶ **32.** [*M21*] Show how to modify the RSA encoding scheme so that there is no problem with messages $< \sqrt[3]{N}$, in such a way that the length of messages is not substantially increased.

33. [*M50*] Prove or disprove: If a reasonably efficient algorithm exists that has a nonnegligible probability of being able to find $x \bmod N$, given a number $N = pq$ whose prime factors satisfy $p \equiv q \equiv 2$ (modulo 3) and given the value of $x^3 \bmod N$, then there is a reasonably efficient algorithm that has a nonnegligible probability of being able to find the factors of N. [If this could be proved, it would not only show that the cube root problem is as difficult as factoring, it would also show that the RSA scheme has the same fatal flaw as the SQRT scheme.]

34. [*M30*] (Peter Weinberger.) Suppose $N = pq$ in the RSA scheme, and suppose you know a number m such that $x^m \bmod N = 1$ for at least 10^{-12} of all positive integers x. Explain how you would go about factoring N without great difficulty, if m is not too large.

▶ **35.** [*M25*] (H. C. Williams, 1979.) Let N be the product of two primes p and q, where $p \bmod 8 = 3$ and $q \bmod 8 = 7$. Prove that the Jacobi symbol satisfies $(\frac{-x^2}{N}) = (\frac{x}{N}) = -(\frac{2x}{N})$, and use this to design an encoding/decoding scheme analogous to Rabin's SQRT box but with no ambiguity of messages.

36. [*HM24*] The asymptotic analysis following (21) is too coarse to give meaningful values unless N is extremely large, since $\ln\ln N$ is always rather small when N is in a practical range. Carry out a more precise analysis that gives insight into the behavior of (21) for reasonable values of N; also explain how to choose a value of $\ln m$ that minimizes (21) except for a factor of size at most $\exp(O(\log\log N))$.

37. [*M27*] Prove that the square root of every positive integer D has a periodic continued fraction of the form

$$\sqrt{D} = R + /a_1, \ldots, a_n, 2R, a_1, \ldots, a_n, 2R, a_1, \ldots, a_n, 2R, \ldots /,$$

unless D is a perfect square, where $R = \lfloor\sqrt{D}\rfloor$ and $(a_1, \ldots, a_n)$ is a *palindrome* (i.e., $a_i = a_{n+1-i}$ for $1 \le i \le n$).

▶ **38.** [*M36*] (A. Shamir.) Consider an abstract computer that can perform the operations $x + y$, $x - y$, $x \cdot y$, and $\lfloor x/y \rfloor$ on integers x and y of arbitrary length, in just one unit of time, no matter how large those integers are. The machine stores integers in a random-access memory and it can select different program steps depending on whether or not $x = y$, given x and y. The purpose of this exercise is to demonstrate that there is an amazingly fast way to factorize numbers on such a computer. (Therefore it will probably be quite difficult to show that factorization is inherently complicated on *real* machines, although we suspect that it is.)

a) Find a way to compute $n!$ in $O(\log n)$ steps on such a computer, given an integer value $n \ge 2$. [*Hint:* If A is a sufficiently large integer, the binomial coefficients $\binom{m}{k} = m!/(m-k)!\,k!$ can be computed readily from the value of $(A+1)^m$.]

b) Show how to compute a number $f(n)$ in $O(\log n)$ steps on such a computer, given an integer value $n \ge 2$, having the following properties: $f(n) = n$ if n is prime, otherwise $f(n)$ is a proper (but not necessarily prime) divisor of n. [*Hint:* If $n \neq 4$, one such function $f(n)$ is $\gcd(m(n), n)$, where $m(n) = \min\{\, m \mid m! \bmod n = 0 \,\}$.]

(As a consequence of (b), we can completely factor a given number n by doing only $O(\log n)^2$ arithmetic operations on arbitrarily large integers: Given a partial factorization $n = n_1 \ldots n_r$, each nonprime n_i can be replaced by $f(n_i) \cdot (n_i / f(n_i))$ in a total of $\sum O(\log n_i) = O(\log n)$ steps, and this refinement operation can be repeated until all n_i are prime.)

The problem of distinguishing prime numbers from composites,
and of resolving composite numbers into their prime factors,
is one of the most important and useful in all of arithmetic.
... The dignity of science seems to demand that every aid to the solution
of such an elegant and celebrated problem be zealously cultivated.

—K. F. GAUSS, *Disquisitiones Arithmeticæ*, Art. 329 (1801)

4.6. POLYNOMIAL ARITHMETIC

THE TECHNIQUES we have been studying apply in a natural way to many different types of mathematical quantities, not simply to numbers. In this section we shall deal with polynomials, which are the next step up from numbers. Formally speaking, a *polynomial over* S is an expression of the form

$$u(x) = u_n x^n + \cdots + u_1 x + u_0, \tag{1}$$

where the "coefficients" $u_n, \ldots, u_1, u_0$ are elements of some algebraic system S, and the "variable" x may be regarded as a formal symbol with an indeterminate meaning. We will assume that the algebraic system S is a *commutative ring with identity*; this means that S admits the operations of addition, subtraction, and multiplication, satisfying the customary properties: Addition and multiplication are associative and commutative binary operations defined on S, where multiplication distributes over addition; and subtraction is the inverse of addition. There is an additive identity element 0 such that $a + 0 = a$, and a multiplicative identity element 1 such that $a \cdot 1 = a$, for all a in S. The polynomial $0x^{n+m} + \cdots + 0x^{n+1} + u_n x^n + \cdots + u_1 x + u_0$ is regarded as the same polynomial as (1), although its expression is formally different.

We say that (1) is a polynomial of *degree* n and *leading coefficient* u_n if $u_n \neq 0$; and in this case we write

$$\deg(u) = n, \qquad \ell(u) = u_n. \tag{2}$$

By convention, we also set

$$\deg(0) = -\infty, \qquad \ell(0) = 0, \tag{3}$$

where "0" denotes the zero polynomial whose coefficients are all zero. We say that $u(x)$ is a *monic polynomial* if its leading coefficient $\ell(u)$ is 1.

Arithmetic on polynomials consists primarily of addition, subtraction, and multiplication; in some cases, further operations such as division, exponentiation, factoring, and taking the greatest common divisor are important. The processes of addition, subtraction, and multiplication are defined in a natural way, as though the variable x were an element of S: Addition and subtraction are done by adding or subtracting the coefficients of like powers of x. Multiplication is done by the rule

$$(u_r x^r + \cdots + u_0)(v_s x^s + \cdots + v_0) = (w_{r+s} x^{r+s} + \cdots + w_0),$$

where

$$w_k = u_0 v_k + u_1 v_{k-1} + \cdots + u_{k-1} v_1 + u_k v_0. \tag{4}$$

In the latter formula u_i or v_j are treated as zero if $i > r$ or $j > s$.

The algebraic system S is usually the set of integers, or the rational numbers; or it may itself be a set of polynomials (in variables other than x); in the latter

situation (1) is a *multivariate* polynomial, a polynomial in several variables. Another important case occurs when the algebraic system S consists of the integers $0, 1, \ldots, m-1$, with addition, subtraction, and multiplication performed mod m (cf. Eq. 4.3.2–11); this is called *polynomial arithmetic modulo* m. The special case of polynomial arithmetic modulo 2, when each of the coefficients is 0 or 1, is especially important.

The reader should note the similarity between polynomial arithmetic and multiple-precision arithmetic (Section 4.3.1), where the radix b is substituted for x. The chief difference is that the coefficient u_k of x^k in polynomial arithmetic bears little or no essential relation to its neighboring coefficients $u_{k\pm1}$, so the idea of "carrying" from one place to the next is absent. In fact, polynomial arithmetic modulo b is essentially identical to multiple-precision arithmetic with radix b, except that all carries are suppressed. For example, compare the multiplication of $(1101)_2$ by $(1011)_2$ in the binary number system with the analogous multiplication of x^3+x^2+1 by x^3+x+1 modulo 2:

Binary system	Polynomials modulo 2
1101	1101
$\times$ 1011	$\times$ 1011
1101	1101
1101	1101
1101	1101
10001111	1111111

The product of these polynomials modulo 2 is obtained by suppressing all carries, and it is $x^6+x^5+x^4+x^3+x^2+x+1$. If we had multiplied the same polynomials over the integers, without taking residues modulo 2, the result would have been $x^6+x^5+x^4+3x^3+x^2+x+1$; again carries are suppressed, but in this case the coefficients can get arbitrarily large.

In view of this strong analogy with multiple-precision arithmetic, it is unnecessary to discuss polynomial addition, subtraction, and multiplication much further in this section. However, we should point out some factors that often make polynomial arithmetic somewhat different from multiple-precision arithmetic in practice: There is often a tendency to have a large number of zero coefficients, and polynomials of huge degrees, so special forms of representation are desirable; see Section 2.2.4. Furthermore, arithmetic on polynomials in several variables leads to routines that are best understood in a recursive framework; this situation is discussed in Chapter 8.

Although the techniques of polynomial addition, subtraction, and multiplication are comparatively straightforward, there are several other important aspects of polynomial arithmetic that deserve special examination. The following subsections therefore discuss *division* of polynomials, with associated techniques such as finding greatest common divisors and factoring. We shall also discuss the problem of efficient *evaluation* of polynomials, i.e., the task of finding the value of $u(x)$ when x is a given element of S, using as few operations as possible.

The special case of evaluating x^n rapidly when n is large is quite important by itself, so it is discussed in detail in Section 4.6.3.

The first major set of computer subroutines for doing polynomial arithmetic was the ALPAK system [W. S. Brown, J. P. Hyde, and B. A. Tague, *Bell System Tech. J.* **42** (1963), 2081–2119; **43** (1964), 785–804, 1547–1562]. Another early landmark in this field was the PM system of George Collins [*CACM* **9** (1966), 578–589]; see also C. L. Hamblin, *Comp. J.* **10** (1967), 168–171.

EXERCISES

1. [*10*] If we are doing polynomial arithmetic modulo 10, what is $7x + 2$ minus $x^2 + 3$? What is $6x^2 + x + 3$ times $5x^2 + 2$?

2. [*17*] True or false: (a) The product of monic polynomials is monic. (b) The product of polynomials of respective degrees m and n has degree $m + n$. (c) The sum of polynomials of respective degrees m and n has degree $\max(m, n)$.

3. [*M20*] If each of the coefficients $u_\tau, \ldots, u_0, v_s, \ldots, v_0$ in (4) is an integer satisfying the conditions $|u_i| \leq m_1$, $|v_j| \leq m_2$, what is the maximum absolute value of the product coefficients w_k?

▶ **4.** [*21*] Can the multiplication of polynomials modulo 2 be facilitated by using the ordinary arithmetic operations on a binary computer, if coefficients are packed into computer words?

▶ **5.** [*M21*] Show how to multiply two polynomials of degree $\leq n$, modulo 2, with an execution time proportional to $O(n^{\lg 3})$ when n is large, by adapting Karatsuba's method (cf. Section 4.3.3).

4.6.1. Division of Polynomials

It is possible to divide one polynomial by another in essentially the same way that we divide one multiple-precision integer by another, when arithmetic is being done on polynomials over a "field." A field S is a commutative ring with identity, in which exact division is possible as well as the operations of addition, subtraction, and multiplication; this means as usual that whenever u and v are elements of S, and $v \neq 0$, there is an element w in S such that $u = vw$. The most important fields of coefficients that arise in applications are

a) the rational numbers (represented as fractions, see Section 4.5.1);
b) the real or complex numbers (represented within a computer by means of floating point approximations; see Section 4.2);
c) the integers modulo p where p is prime (where division can be implemented as suggested in exercise 4.5.2–15);
d) "rational functions" over a field (namely, quotients of two polynomials whose coefficients are in that field, the denominator being monic).

Of special importance is the field of integers modulo 2, when the two values 0 and 1 are the only elements of the field. Polynomials over this field (namely polynomials modulo 2) have many analogies to integers expressed in binary notation; and rational functions over this field have striking analogies to rational numbers whose numerator and denominator are represented in binary notation.

Given two polynomials $u(x)$ and $v(x)$ over a field, with $v(x) \neq 0$, we can divide $u(x)$ by $v(x)$ to obtain a quotient polynomial $q(x)$ and a remainder polynomial $r(x)$ satisfying the conditions

$$u(x) = q(x) \cdot v(x) + r(x), \qquad \deg(r) < \deg(v). \tag{1}$$

It is easy to see that there is at most one pair of polynomials $\big(q(x), r(x)\big)$ satisfying these relations; for if (1) holds for both $\big(q_1(x), r_1(x)\big)$ and $\big(q_2(x), r_2(x)\big)$ and for the same polynomials $u(x), v(x)$, then $q_1(x)v(x) + r_1(x) = q_2(x)v(x) + r_2(x)$, so $\big(q_1(x) - q_2(x)\big)v(x) = r_2(x) - r_1(x)$. Now if $q_1(x) - q_2(x)$ is nonzero, we have $\deg\big((q_1 - q_2) \cdot v\big) = \deg(q_1 - q_2) + \deg(v) \geq \deg(v) > \deg(r_2 - r_1)$, a contradiction; hence $q_1(x) - q_2(x) = 0$ and $r_1(x) = 0$. ·

The following algorithm, which is essentially the same as Algorithm 4.3.1D for multiple-precision division but without any concerns of carries, may be used to determine $q(x)$ and $r(x)$:

Algorithm D (*Division of polynomials over a field*). Given polynomials

$$u(x) = u_m x^m + \cdots + u_1 x + u_0, \qquad v(x) = v_n x^n + \cdots + v_1 x + v_0$$

over a field S, where $v_n \neq 0$ and $m \geq n \geq 0$, this algorithm finds the polynomials

$$q(x) = q_{m-n} x^{m-n} + \cdots + q_0, \qquad r(x) = r_{n-1} x^{n-1} + \cdots + r_0$$

over S that satisfy (1).

D1. [Iterate on k.] Do step D2 for $k = m - n, m - n - 1, \ldots, 0$; then the algorithm terminates with $(r_{n-1}, \ldots, r_0) \leftarrow (u_{n-1}, \ldots, u_0)$.

D2. [Division loop.] Set $q_k \leftarrow u_{n+k}/v_n$, and then set $u_j \leftarrow u_j - q_k v_{j-k}$ for $j = n+k-1, n+k-2, \ldots, k$. (The latter operation amounts to replacing $u(x)$ by $u(x) - q_k x^k v(x)$, a polynomial of degree $< n + k$.) ∎

An example of Algorithm D appears below in (5). The number of arithmetic operations is essentially proportional to $n(m - n + 1)$. For some reason this procedure has become known as "synthetic division" of polynomials. Note that explicit division of coefficients is done only at the beginning of step D2, and the divisor is always v_n; thus if $v(x)$ is a monic polynomial (with $v_n = 1$), there is no actual division at all. If multiplication is easier to perform than division it will be preferable to compute $1/v_n$ at the beginning of the algorithm and to multiply by this quantity in step D2.

We shall often write $u(x) \bmod v(x)$ for the remainder $r(x)$ in (1).

Unique factorization domains. If we restrict consideration to polynomials over a field, we are not coming to grips with many important cases, such as polynomials over the integers or polynomials in several variables. Let us therefore now consider the more general situation that the algebraic system S of coefficients is a *unique factorization domain*, not necessarily a field. This means that S is a commutative ring with identity, and that

i) $uv \neq 0$, whenever u and v are nonzero elements of S;

ii) every nonzero element u of S is either a "unit" or has a "unique" representation of the form

$$u = p_1 \dots p_t, \qquad t \geq 1, \tag{2}$$

where $p_1, \dots, p_t$ are "primes."

Here a "unit" u is an element that has a reciprocal, i.e., an element such that $uv = 1$ for some v in S; and a "prime" p is a nonunit element such that the equation $p = qr$ can be true only if either q or r is a unit. The representation (2) is to be unique in the sense that if $p_1 \dots p_t = q_1 \dots q_s$, where all the p's and q's are primes, then $s = t$ and there is a permutation $\pi_1 \dots \pi_t$ of $\{1, \dots, t\}$ such that $p_1 = a_1 q_{\pi_1}, \dots, p_t = a_t q_{\pi_t}$ for some units $a_1, \dots, a_t$. In other words, factorization into primes is unique, except for unit multiples and except for the order of the factors.

Any field is a unique factorization domain, in which each nonzero element is a unit and there are no primes. The integers form a unique factorization domain in which the units are $+1$ and -1, and the primes are $\pm 2, \pm 3, \pm 5, \pm 7, \pm 11$, etc. The case that S is the set of all integers is of principal importance, because it is often preferable to work with integer coefficients instead of arbitrary rational coefficients.

One of the key facts about polynomials (see exercise 10) is that *the polynomials over a unique factorization domain form a unique factorization domain.* A polynomial that is "prime" in this domain is usually called an *irreducible polynomial.* By using the unique factorization theorem repeatedly, we can prove that multivariate polynomials over the integers, or over any field, in any number of variables, can be uniquely factored into irreducible polynomials. For example, the multivariate polynomial $90x^3 - 120x^2y + 18x^2yz - 24xy^2z$ over the integers is the product of five irreducible polynomials $2 \cdot 3 \cdot x \cdot (3x - 4y) \cdot (5x + yz)$. The same polynomial, as a polynomial over the rationals, is the product of three irreducible polynomials $(6x) \cdot (3x - 4y) \cdot (5x + yz)$; this factorization can also be written $x \cdot (90x - 120y) \cdot (x + \frac{1}{6}yz)$ and in infinitely many other ways, although the factorization is essentially unique.

As usual, we say that $u(x)$ is a multiple of $v(x)$, and that $v(x)$ is a divisor of $u(x)$, if $u(x) = v(x)q(x)$ for some polynomial $q(x)$. If we have an algorithm to tell whether or not u is a multiple of v for arbitrary nonzero elements u and v of a unique factorization domain S, and to determine w if $u = v \cdot w$, then Algorithm D gives us a method to tell whether or not $u(x)$ is a multiple of $v(x)$ for arbitrary nonzero polynomials $u(x)$ and $v(x)$ over S. For if $u(x)$ is a multiple

of $v(x)$, it is easy to see that u_{n+k} must be a multiple of v_n each time we get to step D2, hence the quotient $u(x)/v(x)$ will be found. (Applying this observation repeatedly, we obtain an algorithm that decides if a given polynomial over S, in any number of variables, is a multiple of another given polynomial over S, and the algorithm will find the quotient when it exists.)

A set of elements of a unique factorization domain is said to be *relatively prime* if no prime of that unique factorization domain divides all of them. A polynomial over a unique factorization domain is called *primitive* if its coefficients are relatively prime. (This concept should not be confused with the quite different idea of "primitive polynomials modulo p" discussed in Section 3.2.2.) The following fact, introduced for the case of polynomials over the integers by K. F. Gauss in article 42 of his celebrated book *Disquisitiones Arithmeticæ* (Leipzig: 1801), is of prime importance:

Lemma G (Gauss's Lemma). *The product of primitive polynomials over a unique factorization domain is primitive.*

Proof. Let $u(x) = u_m x^m + \cdots + u_0$ and $v(x) = v_n x^n + \cdots + v_0$ be primitive polynomials. If p is any prime of the domain, we must show that p does not divide all the coefficients of $u(x)v(x)$. By assumption, there is an index j such that u_j is not divisible by p, and an index k such that v_k is not divisible by p. Let j and k be as small as possible; then the coefficient of x^{j+k} in $u(x)v(x)$ is $u_j v_k + u_{j+1}v_{k-1} + \cdots + u_{j+k}v_0 + u_{j-1}v_{k+1} + \cdots + u_0 v_{k+j}$, and it is easy to see that this is not a multiple of p (since its first term isn't, but all of its other terms are). ∎

If a nonzero polynomial $u(x)$ over a unique factorization domain S is not primitive, we can write $u(x) = p_1 \cdot u_1(x)$, where p_1 is a prime of S dividing all the coefficients of $u(x)$, and where $u_1(x)$ is another nonzero polynomial over S. All of the coefficients of $u_1(x)$ have one less prime factor than the corresponding coefficients of $u(x)$. Now if $u_1(x)$ is not primitive, we can write $u_1(x) = p_2 \cdot u_2(x)$, etc.; this process must ultimately terminate in a representation $u(x) = c \cdot u_k(x)$, where c is an element of S and $u_k(x)$ is primitive. In fact, we have the following companion to Lemma G:

Lemma H. *Any nonzero polynomial $u(x)$ over a unique factorization domain S can be factored in the form $u(x) = c \cdot v(x)$, where c is in S and $v(x)$ is primitive. Furthermore, this representation is unique, in the sense that if $u = c_1 \cdot v_1(x) = c_2 \cdot v_2(x)$, then $c_1 = ac_2$ and $v_2(x) = av_1(x)$ where a is a unit of S.*

Proof. We have shown that such a representation exists, so only the uniqueness needs to be proved. Assume that $c_1 \cdot v_1(x) = c_2 \cdot v_2(x)$, where $v_1(x)$ and $v_2(x)$ are primitive and c_1 is not a unit multiple of c_2. By unique factorization there is a prime p of S and an exponent k such that p^k divides one of $\{c_1, c_2\}$ but not the other, say p^k divides c_1 but not c_2. Then p^k divides all of the coefficients of $c_2 \cdot v_2(x)$, so p divides all the coefficients of $v_2(x)$, contradicting the assumption that $v_2(x)$ is primitive. Hence $c_1 = ac_2$, where a is a unit; and $0 = ac_2 \cdot v_1(x) - c_2 \cdot v_2(x) = c_2 \cdot \big(av_1(x) - v_2(x)\big)$, so $av_1(x) - v_2(x) = 0$. ∎

Therefore we may write any nonzero polynomial $u(x)$ as

$$u(x) = \operatorname{cont}(u) \cdot \operatorname{pp}(u(x)), \tag{3}$$

where $\operatorname{cont}(u)$, the "content" of u, is an element of S, and $\operatorname{pp}(u(x))$, the "primitive part" of $u(x)$, is a primitive polynomial over S. When $u(x) = 0$, it is convenient to define $\operatorname{cont}(u) = \operatorname{pp}(u(x)) = 0$. Combining Lemmas G and H gives us the relations

$$\begin{aligned} \operatorname{cont}(u \cdot v) &= a \operatorname{cont}(u) \operatorname{cont}(v), \\ \operatorname{pp}(u(x) \cdot v(x)) &= b \operatorname{pp}(u(x)) \operatorname{pp}(v(x)), \end{aligned} \tag{4}$$

where a and b are units, depending on u and v, with $ab = 1$. When we are working with polynomials over the integers, the only units are $+1$ and -1, and it is conventional to define $\operatorname{pp}(u(x))$ so that its leading coefficient is positive; then (4) is true with $a = b = 1$. When working with polynomials over a field we may take $\operatorname{cont}(u) = \ell(u)$, so that $\operatorname{pp}(u(x))$ is monic; in this case again (4) holds with $a = b = 1$, for all $u(x)$ and $v(x)$.

For example, if we are dealing with polynomials over the integers, let $u(x) = -26x^2 + 39$ and $v(x) = 21x + 14$. Then

$$\begin{aligned} \operatorname{cont}(u) &= -13, & \operatorname{pp}(u(x)) &= 2x^2 - 3, \\ \operatorname{cont}(v) &= +7, & \operatorname{pp}(v(x)) &= 3x + 2, \\ \operatorname{cont}(u \cdot v) &= -91, & \operatorname{pp}(u(x) \cdot v(x)) &= 6x^3 + 4x^2 - 9x - 6. \end{aligned}$$

Greatest common divisors. When there is unique factorization, it makes sense to speak of a "greatest common divisor" of two elements; this is a common divisor that is divisible by as many primes as possible. (Cf. Eq. 4.5.2–6.) Since a unique factorization domain may have many units, however, there is a certain amount of ambiguity in this definition of greatest common divisor; if w is a greatest common divisor of u and v, so is $a \cdot w$, when a is any unit. Conversely, the assumption of unique factorization implies that if w_1 and w_2 are both greatest common divisors of u and v, then $w_1 = a \cdot w_2$ for some unit a. In other words it does not make sense, in general, to speak of "the" greatest common divisor of u and v; there is a set of greatest common divisors, each one being a unit multiple of the others.

Let us now consider the problem of finding a greatest common divisor of two given polynomials over an algebraic system S. If S is a field, the problem is relatively simple; our division algorithm, Algorithm D, can be extended to an algorithm that computes greatest common divisors, just as Euclid's algorithm (Algorithm 4.5.2A) yields the greatest common divisor of two given integers based on a division algorithm for integers: If $v(x) = 0$, then $\gcd(u(x), v(x)) = u(x)$; otherwise $\gcd(u(x), v(x)) = \gcd(v(x), r(x))$, where $r(x)$ is given by (1). This procedure is called Euclid's algorithm for polynomials over a field; it was first used by Simon Stevin in 1585 [*Les œuvres mathématiques de Simon Stevin*, ed. by A. Girard, **1** (Leyden, 1634), 56].

For example, let us determine the gcd of $x^8 + x^6 + 10x^4 + 10x^3 + 8x^2 + 2x + 8$ and $3x^6 + 5x^4 + 9x^2 + 4x + 8$, mod 13, by using Euclid's algorithm for polynomials over the integers modulo 13. First, writing only the coefficients to show the steps of Algorithm D, we have

$$\begin{array}{rrrrrrrrrr}
 & & & & & & & 9 & 0 & 7 \\
3 & 0 & 5 & 0 & 9 & 4 & 8 \;)\; & 1 & 0 & 1 & 0 & 10 & 10 & 8 & 2 & 8 \\
 & & & & & & & 1 & 0 & 6 & 0 & 3 & 10 & 7 & & \\
\cline{4-14}
 & & & & & & & 0 & 8 & 0 & 7 & 0 & 1 & 2 & 8 & \\
 & & & & & & & 8 & 0 & 9 & 0 & 11 & 2 & 4 & & \\
\cline{5-14}
 & & & & & & & & 0 & 11 & 0 & 3 & 0 & 4 & &
\end{array} \tag{5}$$

and hence

$$\begin{aligned}
x^8 + x^6 + 10x^4 + 10x^3 + 8x^2 + 2x + 8 \\
\qquad = (9x^2 + 7)(3x^6 + 5x^4 + 9x^2 + 4x + 8) \;+\; (11x^4 + 3x^2 + 4).
\end{aligned}$$

Similarly,

$$\begin{aligned}
3x^6 + 5x^4 + 9x^2 + 4x + 8 &= (5x^2 + 5)(11x^4 + 3x^2 + 4) \;+\; (4x + 1); \\
11x^4 + 3x^2 + 4 &= (6x^3 + 5x^2 + 6x + 5)(4x + 1) \;+\; 12; \\
4x + 1 &= (9x + 12) \cdot 12 \;+\; 0.
\end{aligned} \tag{6}$$

(The equality sign here means congruence modulo 13, since all arithmetic on the coefficients has been done mod 13.) This computation shows that 12 is a greatest common divisor of the two original polynomials. Now any nonzero element of a field is a unit of the domain of polynomials over that field, so it is conventional in the case of fields to divide the result of the algorithm by its leading coefficient, producing a *monic* polynomial that is called *the* greatest common divisor of the two given polynomials. The gcd computed in (6) is accordingly taken to be 1, not 12. The last step in (6) could have been omitted, for if $\deg(v) = 0$, then $\gcd(u(x), v(x)) = 1$, no matter what polynomial is chosen for $u(x)$. Exercise 4 determines the average running time for Euclid's algorithm on random polynomials modulo p.

Let us now turn to the more general situation in which our polynomials are given over a unique factorization domain that is not a field. From Eqs. (4) we can deduce the important relations

$$\begin{aligned}
\operatorname{cont}(\gcd(u, v)) &= a \cdot \gcd(\operatorname{cont}(u), \operatorname{cont}(v)), \\
\operatorname{pp}(\gcd(u(x), v(x))) &= b \cdot \gcd(\operatorname{pp}(u(x)), \operatorname{pp}(v(x))),
\end{aligned} \tag{7}$$

where a and b are units. Here $\gcd(u(x), v(x))$ denotes any particular polynomial in x that is a greatest common divisor of $u(x)$ and $v(x)$. Equations (7) reduce the problem of finding greatest common divisors of arbitrary polynomials to the problem of finding greatest common divisors of *primitive* polynomials.

Algorithm D for division of polynomials over a field can be generalized to a pseudo-division of polynomials over any algebraic system that is a commutative ring with identity. We can observe that Algorithm D requires explicit division only by $\ell(v)$, the leading coefficient of $v(x)$, and that step D2 is carried out exactly $m-n+1$ times; thus if $u(x)$ and $v(x)$ start with integer coefficients, and if we are working over the rational numbers, then the only denominators that appear in the coefficients of $q(x)$ and $r(x)$ are divisors of $\ell(v)^{m-n+1}$. This suggests that we can always find polynomials $q(x)$ and $r(x)$ such that

$$\ell(v)^{m-n+1}u(x) = q(x)v(x) + r(x), \qquad \deg(r) < n, \tag{8}$$

where $m = \deg(u)$ and $n = \deg(v)$, for any polynomials $u(x)$ and $v(x) \neq 0$, provided that $m \geq n$.

Algorithm R (*Pseudo-division of polynomials*). Given polynomials

$$u(x) = u_m x^m + \cdots + u_1 x + u_0, \qquad v(x) = v_n x^n + \cdots + v_1(x) + v_0,$$

where $v_n \neq 0$ and $m \geq n \geq 0$, this algorithm finds polynomials $q(x) = q_{m-n}x^{m-n} + \cdots + q_0$ and $r(x) = r_{n-1}x^{n-1} + \cdots + r_0$ satisfying (8).

R1. [Iterate on k.] Do step R2 for $k = m-n, m-n-1, \ldots, 0$; then the algorithm terminates with $(r_{n-1}, \ldots, r_0) = (u_{n-1}, \ldots, u_0)$.

R2. [Multiplication loop.] Set $q_k \leftarrow u_{n+k}v_n^k$, and set $u_j \leftarrow v_n u_j - u_{n+k}v_{j-k}$ for $j = n+k-1, n+k-2, \ldots, 0$. (When $j < k$ this means that $u_j \leftarrow v_n u_j$, since we treat $v_{-1}, v_{-2}, \ldots$ as zero. These multiplications could have been avoided if we had started the algorithm by replacing u_t by $v_n^{m-n-t}u_t$, for $0 \leq t < m-n$.) ∎

An example calculation appears below in (10). It is easy to prove the validity of Algorithm R by induction on $m-n$, since each execution of step R2 essentially replaces $u(x)$ by $\ell(v)u(x) - \ell(u)x^k v(x)$, where $k = \deg(u) - \deg(v)$. Note that no division whatever is used in this algorithm; the coefficients of $q(x)$ and $r(x)$ are themselves certain polynomial functions of the coefficients of $u(x)$ and $v(x)$. If $v_n = 1$, the algorithm is identical to Algorithm D. If $u(x)$ and $v(x)$ are polynomials over a unique factorization domain, we can prove as before that the polynomials $q(x)$ and $r(x)$ are unique; therefore another way to do the pseudo-division over a unique factorization domain is to multiply $u(x)$ by v_n^{m-n+1} and apply Algorithm D, knowing that all the quotients in step D2 will exist.

Algorithm R can be extended to a "generalized Euclidean algorithm" for primitive polynomials over a unique factorization domain, in the following way: Let $u(x)$ and $v(x)$ be primitive polynomials with $\deg(u) \geq \deg(v)$, and determine the polynomial $r(x)$ satisfying (8) by means of Algorithm R. Now we can prove that $\gcd\big(u(x), v(x)\big) = \gcd\big(v(x), r(x)\big)$: Any common divisor of $u(x)$ and $v(x)$ divides $v(x)$ and $r(x)$; conversely, any common divisor of $v(x)$ and $r(x)$ divides $\ell(v)^{m-n+1}u(x)$, and it must be primitive (since $v(x)$ is primitive) so it divides $u(x)$. If $r(x) = 0$, we therefore have $\gcd\big(u(x), v(x)\big) = v(x)$; on the other hand if $r(x) \neq 0$, we have $\gcd\big(v(x), r(x)\big) = \gcd\big(v(x), \mathrm{pp}\big(r(x)\big)\big)$ since $v(x)$ is primitive, so the process can be iterated.

Algorithm E (*Generalized Euclidean algorithm*). Given nonzero polynomials $u(x)$ and $v(x)$ over a unique factorization domain S, this algorithm calculates a greatest common divisor of $u(x)$ and $v(x)$. We assume that auxiliary algorithms exist to calculate greatest common divisors of elements of S, and to divide a by b in S when $b \neq 0$ and a is a multiple of b.

E1. [Reduce to primitive.] Set $d \leftarrow \gcd(\text{cont}(u), \text{cont}(v))$, using the assumed algorithm for calculating greatest common divisors in S. (Note that $\text{cont}(u)$ is a greatest common divisor of the coefficients of $u(x)$.) Replace $u(x)$ by the polynomial $u(x)/\text{cont}(u) = \text{pp}(u(x))$; similarly, replace $v(x)$ by $\text{pp}(v(x))$.

E2. [Pseudo-division.] Calculate $r(x)$ using Algorithm R. (It is unnecessary to calculate the quotient polynomial $q(x)$.) If $r(x) = 0$, go to E4. If $\deg(r) = 0$, replace $v(x)$ by the constant polynomial "1" and go to E4.

E3. [Make remainder primitive.] Replace $u(x)$ by $v(x)$ and replace $v(x)$ by $\text{pp}(r(x))$. Go back to step E2. (This is the "Euclidean step," analogous to the other instances of Euclid's algorithm that we have seen.)

E4. [Attach the content.] The algorithm terminates, with $d \cdot v(x)$ as the desired answer. ∎

As an example of Algorithm E, let us calculate the gcd of the polynomials

$$\begin{aligned} u(x) &= x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5, \\ v(x) &= 3x^6 + 5x^4 - 4x^2 - 9x + 21, \end{aligned} \tag{9}$$

over the integers. These polynomials are primitive, so step E1 sets $d \leftarrow 1$. In step E2 we have the pseudo-division

$$\begin{array}{rrrrrrrrrr}
 & & & & & & & 1 & 0 & -6 \\
3\ 0\ 5\ 0\ -4\ -9\ 21\ ) & 1 & 0 & 1 & 0 & -3 & -3 & 8 & 2 & -5 \\
 & 3 & 0 & 3 & 0 & -9 & -9 & 24 & 6 & -15 \\
\cline{2-10}
 & 3 & 0 & 5 & 0 & -4 & -9 & 21 & & \\
\cline{2-10}
 & & 0 & -2 & 0 & -5 & 0 & 3 & 6 & -15 \\
 & & 0 & -6 & 0 & -15 & 0 & 9 & 18 & -45 \\
 & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\cline{2-10}
 & & & -6 & 0 & -15 & 0 & 9 & 18 & -45 \\
 & & & -18 & 0 & -45 & 0 & 27 & 54 & -135 \\
 & & & -18 & 0 & -30 & 0 & 24 & 54 & -126 \\
\cline{2-10}
 & & & & & -15 & 0 & 3 & 0 & -9
\end{array} \tag{10}$$

Here the quotient $q(x)$ is $1 \cdot 3^2x^2 + 0 \cdot 3^1x + -6 \cdot 3^0$; we have

$$27u(x) = v(x)(9x^2 - 6) + (-15x^4 + 3x^2 - 9). \tag{11}$$

Now step E3 replaces $u(x)$ by $v(x)$ and $v(x)$ by $\text{pp}(r(x)) = 5x^4 - x^2 + 3$. The subsequent calculation is summarized in the following table, where only the

coefficients are shown:

$u(x)$	$v(x)$	$r(x)$	
$1,0,1,0,-3,-3,8,2,-5$	$3,0,5,0,-4,-9,21$	$-15,0,3,0,-9$	
$3,0,5,0,-4,-9,21$	$5,0,-1,0,3$	$-585,-1125,2205$	
$5,0,-1,0,3$	$13,25,-49$	$-233150,307500$	
$13,25,-49$	$4663,-6150$	143193869	(12)

It is instructive to compare this calculation with the computation of the same greatest common divisor over the *rational* numbers, instead of over the integers, by using Euclid's algorithm for polynomials over a field as described earlier in this section. The following surprisingly complicated sequence appears:

$u(x)$	$v(x)$	
$1,0,1,0,-3,-3,8,2,-5$	$3,0,5,0,-4,-9,21$	
$3,0,5,0,-4,-9,21$	$-\frac{5}{9},0,\frac{1}{9},0,-\frac{1}{3}$	
$-\frac{5}{9},0,\frac{1}{9},0,-\frac{1}{3}$	$-\frac{117}{25},-9,\frac{441}{25}$	
$-\frac{117}{25},-9,\frac{441}{25}$	$\frac{233150}{19773},-\frac{102500}{6591}$	
$\frac{233150}{19773},-\frac{102500}{6591}$	$-\frac{1288744821}{543589225}$	(13)

To improve that algorithm, we can reduce $u(x)$ and $v(x)$ to monic polynomials at each step, since this removes "unit" factors that make the coefficients more complicated than necessary; this is actually Algorithm E over the rationals:

$u(x)$	$v(x)$	
$1,0,1,0,-3,-3,8,2,-5$	$1,0,\frac{5}{3},0,-\frac{4}{3},-3,7$	
$1,0,\frac{5}{3},0,-\frac{4}{3},-3,7$	$1,0,-\frac{1}{5},0,\frac{3}{5}$	
$1,0,-\frac{1}{5},0,\frac{3}{5}$	$1,\frac{25}{13},-\frac{49}{13}$	
$1,\frac{25}{13},-\frac{49}{13}$	$1,-\frac{6150}{4663}$	
$1,-\frac{6150}{4663}$	1	(14)

In both (13) and (14) the sequence of polynomials is essentially the same as (12), which was obtained by Algorithm E over the integers; the only difference is that the polynomials have been multiplied by certain rational numbers. Whether we have $5x^4-x^2+3$ or $-\frac{5}{9}x^4+\frac{1}{9}x^2-\frac{1}{3}$ or $x^4-\frac{1}{5}x^2+\frac{3}{5}$, the computations are essentially the same. But either algorithm using rational arithmetic will run noticeably slower than the all-integer Algorithm E, since rational arithmetic requires many more evaluations of gcd's of integers within each step. Therefore it is definitely better to use the all-integer algorithm.

It is also instructive to compare the above calculations with (6) above, where we determined the gcd of the same polynomials $u(x)$ and $v(x)$ modulo 13 with considerably less labor. Since $\ell(u)$ and $\ell(v)$ are not multiples of 13, the fact that $\gcd(u(x),v(x))=1$ modulo 13 is sufficient to prove that $u(x)$ and $v(x)$ are relatively prime over the integers (and therefore over the rational numbers); we will return to this time-saving observation at the close of Section 4.6.2.

The subresultant algorithm. An ingenious algorithm that is generally superior to Algorithm E, and that gives us further information about Algorithm E's behavior, was discovered by George E. Collins [*JACM* **14** (1967), 128–142] and subsequently improved by W. S. Brown and J. F. Traub [*JACM* **18** (1971), 505–514; see also W. S. Brown, *ACM Trans. Math. Software* **4** (1978), 237–249]. This algorithm avoids the calculation of primitive parts in step E3, dividing instead by an element of S that is known to be a factor of $r(x)$:

Algorithm C (*Greatest common divisor over a unique factorization domain*). This algorithm has the same input and output assumptions as Algorithm E, and has the advantage that fewer calculations of greatest common divisors of coefficients are needed.

C1. [Reduce to primitive.] As in step E1 of Algorithm E, set $d \leftarrow \gcd(\mathrm{cont}(u), \mathrm{cont}(v))$, and replace $\big(u(x), v(x)\big)$ by $\big(\mathrm{pp}(u(x)), \mathrm{pp}(v(x))\big)$. Set $g \leftarrow h \leftarrow 1$.

C2. [Pseudo-division.] Set $\delta \leftarrow \deg(u) - \deg(v)$. Calculate $r(x)$ using Algorithm R. If $r(x) = 0$, go to C4. If $\deg(r) = 0$, replace $v(x)$ by the constant polynomial "1" and go to C4.

C3. [Adjust remainder.] Replace the polynomial $u(x)$ by $v(x)$, and replace $v(x)$ by $r(x)/gh^{\delta}$. (At this point all coefficients of $r(x)$ are multiples of gh^{δ}.) Then set $g \leftarrow \ell(u)$, $h \leftarrow h^{1-\delta}g^{\delta}$ and return to C2. (The new value of h will be in the domain S, even if $\delta > 1$.)

C4. [Attach the content.] Return $d \cdot \mathrm{pp}(v(x))$ as the answer. ∎

If we apply this algorithm to the polynomials (9) considered earlier, the following sequence of results is obtained at the beginning of step C2:

$u(x)$	$v(x)$	g	h	
$1, 0, 1, 0, -3, -3, 8, 2, -5$	$3, 0, 5, 0, -4, -9, 21$	1	1	
$3, 0, 5, 0, -4, -9, 21$	$-15, 0, 3, 0, -9$	3	9	
$-15, 0, 3, 0, -9$	$65, 125, -245$	-15	25	
$65, 125, -245$	$-9326, 12300$	65	169	(15)

At the conclusion of the algorithm, $r(x)/gh^{\delta} = 260708$.

The sequence of polynomials consists of integral multiples of the polynomials in the sequence produced by Algorithm E. In spite of the fact that the polynomials are not reduced to primitive form, the coefficients are kept to a reasonable size because of the reduction factor in step C3.

In order to analyze Algorithm C and to prove that it is valid, let us call the sequence of polynomials it produces $u_1(x)$, $u_2(x)$, $u_3(x)$, ..., where $u_1(x) = u(x)$ and $u_2(x) = v(x)$. Let $\delta_j = n_j - n_{j+1}$ for $j \geq 1$, where $n_j = \deg(u_j)$; and let $g_1 = h_1 = 1$, $g_j = \ell(u_j)$, $h_j = h_{j-1}^{1-\delta_{j-1}} g_j^{\delta_{j-1}}$ for $j \geq 2$. Then we have

$$\begin{aligned} g_2^{\delta_1+1}u_1(x) &= u_2(x)q_1(x) + g_1h_1^{\delta_1}u_3(x), && n_3 < n_2; \\ g_3^{\delta_2+1}u_2(x) &= u_3(x)q_2(x) + g_2h_2^{\delta_2}u_4(x), && n_4 < n_3; \\ g_4^{\delta_3+1}u_3(x) &= u_4(x)q_3(x) + g_3h_3^{\delta_3}u_5(x), && n_5 < n_4; \end{aligned} \tag{16}$$

Table 1

COEFFICIENTS IN ALGORITHM C

Row name	Row														Multiply by	Replace by row
A_5	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	0	0	0	0	0	b_6^3	C_5
A_4	0	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	0	0	0	0	b_6^3	C_4
A_3	0	0	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	0	0	0	b_6^3	C_3
A_2	0	0	0	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	0	0	b_6^3	C_2
A_1	0	0	0	0	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	0	b_6^3	C_1
A_0	0	0	0	0	0	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0	b_6^3	C_0
B_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	0	0	0	0	0	0	0		
B_6	0	b_6	b_5	b_4	b_3	b_2	b_1	b_0	0	0	0	0	0	0		
B_5	0	0	b_6	b_5	b_4	b_3	b_2	b_1	b_0	0	0	0	0	0		
B_4	0	0	0	b_6	b_5	b_4	b_3	b_2	b_1	b_0	0	0	0	0		
B_3	0	0	0	0	b_6	b_5	b_4	b_3	b_2	b_1	b_0	0	0	0	c_4^3/b_6^5	D_3
B_2	0	0	0	0	0	b_6	b_5	b_4	b_3	b_2	b_1	b_0	0	0	c_4^3/b_6^5	D_2
B_1	0	0	0	0	0	0	b_6	b_5	b_4	b_3	b_2	b_1	b_0	0	c_4^3/b_6^5	D_1
B_0	0	0	0	0	0	0	0	b_6	b_5	b_4	b_3	b_2	b_1	b_0	c_4^3/b_6^5	D_0
C_5	0	0	0	0	c_4	c_3	c_2	c_1	c_0	0	0	0	0	0		
C_4	0	0	0	0	0	c_4	c_3	c_2	c_1	c_0	0	0	0	0		
C_3	0	0	0	0	0	0	c_4	c_3	c_2	c_1	c_0	0	0	0		
C_2	0	0	0	0	0	0	0	c_4	c_3	c_2	c_1	c_0	0	0		
C_1	0	0	0	0	0	0	0	0	c_4	c_3	c_2	c_1	c_0	0	$d_2^2b_6^4/c_4^5$	E_1
C_0	0	0	0	0	0	0	0	0	0	c_4	c_3	c_2	c_1	c_0	$d_2^2b_6^4/c_4^5$	E_0
D_3	0	0	0	0	0	0	0	0	d_2	d_1	d_0	0	0	0		
D_2	0	0	0	0	0	0	0	0	0	d_2	d_1	d_0	0	0		
D_1	0	0	0	0	0	0	0	0	0	0	d_2	d_1	d_0	0		
D_0	0	0	0	0	0	0	0	0	0	0	0	d_2	d_1	d_0	$e_2^2c_4^2/d_2^3b_6^2$	F_0
E_1	0	0	0	0	0	0	0	0	0	0	0	e_1	e_0	0		
E_0	0	0	0	0	0	0	0	0	0	0	0	0	e_1	e_0		
F_0	0	0	0	0	0	0	0	0	0	0	0	0	0	f_0		

and so on. The process terminates when $n_{k+1} = \deg(u_{k+1}) \leq 0$. We must show that $u_3(x)$, $u_4(x)$, ..., have coefficients in S, i.e., that the factors $g_j h_j^{\delta_j}$ evenly divide the remainders, and we must also show that the h_j values all belong to S. The proof is rather involved, and it can be most easily understood by considering an example.

Suppose, as in (15), that $n_1 = 8$, $n_2 = 6$, $n_3 = 4$, $n_4 = 2$, $n_5 = 1$, $n_6 = 0$, so that $\delta_1 = \delta_2 = \delta_3 = 2$, $\delta_4 = \delta_5 = 1$. Let us write $u_1(x) = a_8x^8 + a_7x^7 + \cdots + a_0$, $u_2(x) = b_6x^6 + b_5x^5 + \cdots + b_0$, ..., $u_5(x) = e_1x + e_0$, $u_6(x) = f_0$, so that $h_1 = 1$, $h_2 = b_6^2$, $h_3 = c_4^2/b_6^2$, $h_4 = d_2^2b_6^2/c_4^2$. In these terms it is helpful to consider the array shown in Table 1. For concreteness, let us assume that the coefficients of the polynomials are integers. We have $b_6^3u_1(x) = u_2(x)q_1(x) + u_3(x)$; so if we multiply row A_5 by b_6^3 and subtract appropriate multiples of rows B_7, B_6, and B_5 (corresponding to the coefficients of $q_1(x)$) we will get row C_5. If we also multiply row A_4 by b_6^3 and subtract multiples of rows B_6, B_5, and B_4, we get row C_4. In a similar way, we have $c_4^3u_2(x) = u_3(x)q_2(x) + b_6^5u_4(x)$; so we

can multiply row B_3 by c_4^3, subtract integer multiples of rows C_5, C_4, and C_3, then divide by b_6^5 to obtain row D_3.

In order to prove that $u_4(x)$ has integer coefficients, let us consider the matrix

$$\begin{matrix} A_2 \\ A_1 \\ A_0 \\ B_4 \\ B_3 \\ B_2 \\ B_1 \\ B_0 \end{matrix} \begin{pmatrix} a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & 0 & 0 \\ 0 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & 0 \\ 0 & 0 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\ b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 & 0 & 0 & 0 \\ 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 & 0 & 0 \\ 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 & 0 \\ 0 & 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 \\ 0 & 0 & 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \end{pmatrix} = M. \tag{17}$$

The indicated row operations and a permutation of rows will transform M into

$$\begin{matrix} B_4 \\ B_3 \\ B_2 \\ B_1 \\ C_2 \\ C_1 \\ C_0 \\ D_0 \end{matrix} \begin{pmatrix} b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 & 0 & 0 & 0 \\ 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 & 0 & 0 \\ 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 & 0 \\ 0 & 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 \\ 0 & 0 & 0 & 0 & c_4 & c_3 & c_2 & c_1 & c_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c_4 & c_3 & c_2 & c_1 & c_0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & c_4 & c_3 & c_2 & c_1 & c_0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & d_2 & d_1 & d_0 \end{pmatrix} = M'. \tag{18}$$

Because of the way M' has been derived from M, we must have

$$b_6^3 \cdot b_6^3 \cdot b_6^3 \cdot (c_4^3/b_6^5) \cdot \det M_0 = \pm \det M_0',$$

if M_0 and M_0' represent any square matrices obtained by selecting eight corresponding columns from M and M'. For example, let us select the first seven columns and the column containing d_1; then

$$b_6^3 \cdot b_6^3 \cdot b_6^3 \cdot (c_4^3/b_6^5) \cdot \det \begin{pmatrix} a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & 0 \\ 0 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_0 \\ 0 & 0 & a_8 & a_7 & a_6 & a_5 & a_4 & a_1 \\ b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 \\ 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & 0 \\ 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 & 0 \\ 0 & 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_0 \\ 0 & 0 & 0 & 0 & b_6 & b_5 & b_4 & b_1 \end{pmatrix} = \pm b_6^4 \cdot c_4^3 \cdot d_1.$$

Since $b_6 c_4 \neq 0$, this proves that d_1 is an integer. Similarly, d_2 and d_0 are integers.

In general, we can show that $u_{j+1}(x)$ has integer coefficients in a similar manner. If we start with the matrix M consisting of rows $A_{n_2-n_j}$ through A_0 and $B_{n_1-n_j}$ through B_0, and if we perform the row operations indicated in Table 1, we will obtain a matrix M' consisting in some order of rows $B_{n_1-n_j}$

through $B_{n_3-n_j+1}$, $C_{n_2-n_j}$ through $C_{n_4-n_j+1}$, $\ldots$, $P_{n_{j-2}-n_j}$ through P_1, $Q_{n_{j-1}-n_j}$ through Q_0, and finally R_0 (a row containing the coefficients of $u_{j+1}(x)$). Extracting appropriate columns shows that

$$(g_2^{\delta_1+1}/g_1h_1^{\delta_1})^{n_2-n_j+1}(g_3^{\delta_2+1}/g_2h_2^{\delta_2})^{n_3-n_j+1}\ldots(g_j^{\delta_{j-1}+1}/g_{j-1}h_{j-1}^{\delta_{j-1}})^{n_j-n_j+1}$$
$$\times \det M_0 \;=\; \pm g_2^{n_1-n_3}g_3^{n_2-n_4}\ldots g_{j-1}^{n_{j-2}-n_j}g_j^{n_{j-1}-n_j+1}r_t, \quad (19)$$

where r_t is a given coefficient of $u_{j+1}(x)$ and M_0 is a submatrix of M. The h's have been chosen very cleverly so that this equation simplifies to

$$\det M_0 = \pm r_t \quad (20)$$

(see exercise 24). Therefore *every coefficient of $u_{j+1}(x)$ can be expressed as the determinant of an $(n_1+n_2-2n_j+2)\times(n_1+n_2-2n_j+2)$ matrix whose elements are coefficients of $u(x)$ and $v(x)$.*

It remains to be shown that the cleverly chosen h's also are integers. A similar technique applies: Let's look, for example, at the matrix

$$\begin{array}{c} A_1 \\ A_0 \\ B_3 \\ B_2 \\ B_1 \\ B_0 \end{array}\begin{pmatrix} a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & 0 \\ 0 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 \\ b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 & 0 & 0 \\ 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 & 0 \\ 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 \\ 0 & 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \end{pmatrix} = M. \quad (21)$$

Row operations as specified in Table 1, and permutation of rows, leads to

$$\begin{array}{c} B_3 \\ B_2 \\ B_1 \\ B_0 \\ C_1 \\ C_0 \end{array}\begin{pmatrix} b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 & 0 & 0 \\ 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 & 0 \\ 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 \\ 0 & 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ 0 & 0 & 0 & 0 & c_4 & c_3 & c_2 & c_1 & c_0 & 0 \\ 0 & 0 & 0 & 0 & 0 & c_4 & c_3 & c_2 & c_1 & c_0 \end{pmatrix} = M'; \quad (22)$$

hence if we consider any submatrices M_0 and M_0' obtained by selecting six corresponding columns of M and M' we have

$$b_6^3 \cdot b_6^3 \cdot b_6^3 \cdot \det M_0 = \pm \det M_0'.$$

When M_0 is chosen to be the first six columns of M, we find that $\det M_0 = \pm c_4^2/b_6^2 = \pm h_3$, so h_3 is an integer.

In general, to show that h_j is an integer for $j \geq 3$, we start with the matrix M consisting of rows $A_{n_2-n_j-1}$ through A_0 and $B_{n_1-n_j-1}$ through B_0; then we perform appropriate row operations until obtaining a matrix M' consisting of rows $B_{n_1-n_j-1}$ through $B_{n_3-n_j}$, $C_{n_2+n_j-1}$ through $C_{n_4-n_j}$, $\ldots$, $P_{n_{j-2}-n_j-1}$

through P_0, $Q_{n_{j-1}-n_j-1}$ through Q_0. Letting M_0 be the first $n_1+n_2-2n_j$ columns of M, we obtain

$$(g_2^{\delta_1+1}/g_1h_1^{\delta_1})^{n_2-n_j}(g_3^{\delta_2+1}/g_2h_2^{\delta_2})^{n_3-n_j}\dots(g_j^{\delta_{j-1}+1}/g_{j-1}h_{j-1}^{\delta_{j-1}})^{n_j-n_j}\det M_0 = \pm g_2^{n_1-n_3}g_3^{n_2-n_4}\dots g_{j-1}^{n_{j-2}-n_j}g_j^{n_{j-1}-n_j}, \quad (23)$$

an equation that neatly simplifies to

$$\det M_0 = \pm h_j. \quad (24)$$

(This proof, although stated for the domain of integers, obviously applies to any unique factorization domain.)

In the process of verifying Algorithm C, we have also learned that every element of S dealt with by the algorithm can be expressed as a determinant whose entries are the coefficients of the primitive parts of the original polynomials. A well-known theorem of Hadamard (see exercise 15) states that

$$|\det(a_{ij})| \le \prod_{1\le i\le n}\left(\sum_{1\le j\le n}a_{ij}^2\right)^{1/2}; \quad (25)$$

therefore an upper bound for the maximum coefficient appearing in the polynomials computed by Algorithm C is

$$N^{m+n}(m+1)^{n/2}(n+1)^{m/2}, \quad (26)$$

if all coefficients of the given polynomials $u(x)$ and $v(x)$ are bounded by N in absolute value. This same upper bound applies to the coefficients of all polynomials $u(x)$ and $v(x)$ computed during the execution of Algorithm E, since the polynomials obtained in Algorithm E are always divisors of the polynomials obtained in Algorithm C.

This upper bound on the coefficients is extremely gratifying, because it is much better than we would ordinarily have a right to expect. For example, consider what happens if we avoid the corrections in steps E3 and C3, merely replacing $v(x)$ by $r(x)$. This is the simplest gcd algorithm, and it is the one that traditionally appears in textbooks on algebra (for theoretical purposes, not intended for practical calculations). If we suppose that $\delta_1=\delta_2=\cdots=1$, we find that the coefficients of $u_3(x)$ are bounded by N^3, the coefficients of $u_4(x)$ are bounded by N^7, those of $u_5(x)$ by N^{17}, ...; the coefficients of $u_k(x)$ are bounded by N^{a_k}, where $a_k=2a_{k-1}+a_{k-2}$. Thus the upper bound, in place of (25) for $m=n+1$, would be approximately

$$N^{0.5(2.414)^n}, \quad (27)$$

and experiments show that the simple algorithm does in fact have this behavior; the number of digits in the coefficients grows exponentially at each step! In

Algorithm E, by contrast, the growth in the number of digits is only slightly more than linear at most.

Another byproduct of our proof of Algorithm C is the fact that the degrees of the polynomials will almost always decrease by 1 at each step, so that the number of iterations of step C2 (or E2) will usually be $\deg(v)$ if the given polynomials are "random." In order to see why this happens, note for example that we could have chosen the first eight columns of M and M' in (16) and (17), and then we would have found that $u_4(x)$ has degree less than 3 if and only if $d_3 = 0$, that is, if and only if

$$\det\begin{pmatrix} a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 & a_1 \\ 0 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 & a_2 \\ 0 & 0 & a_8 & a_7 & a_6 & a_5 & a_4 & a_3 \\ b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 \\ 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 \\ 0 & 0 & 0 & b_6 & b_5 & b_4 & b_3 & b_2 \\ 0 & 0 & 0 & 0 & b_6 & b_5 & b_4 & b_3 \end{pmatrix} = 0.$$

In general, δ_j will be greater than 1 for $j > 1$ if and only if a similar determinant in the coefficients of $u(x)$ and $v(x)$ is zero. Since such a determinant is a nonzero multivariate polynomial in the coefficients, it will be nonzero "almost always," or "with probability 1." (See exercise 16 for a more precise formulation of this statement, and see exercise 4 for a related proof.) The example polynomials in (15) have both δ_2 and δ_3 equal to 2, so they are exceptional indeed.

The considerations above can be used to derive the well-known fact that two polynomials are relatively prime if and only if their "resultant" is nonzero; the resultant is a determinant having the form of rows A_5 through A_0 and B_7 through B_0 in Table 1. (This is "Sylvester's determinant"; see exercise 12. Further properties of resultants are discussed in B. L. van der Waerden, *Modern Algebra*, tr. by Fred Blum (New York: Ungar, 1949), Sections 27–28.) From the standpoint discussed above, we could say that the gcd is "almost always" of degree zero, since Sylvester's determinant is almost never zero. But many calculations of practical interest would never be undertaken if there weren't some reasonable chance that the gcd would be a polynomial of positive degree.

We can see exactly what happens during Algorithms E and C when the gcd is not 1 by considering $u(x) = w(x)u_1(x)$ and $v(x) = w(x)u_2(x)$, where $u_1(x)$ and $u_2(x)$ are relatively prime and $w(x)$ is primitive. Then if the polynomials $u_1(x)$, $u_2(x)$, $u_3(x)$, ... are obtained when Algorithm E works on $u(x) = u_1(x)$ and $v(x) = u_2(x)$, it is easy to show that the sequence obtained for $u(x) = w(x)u_1(x)$ and $v(x) = w(x)u_2(x)$ is simply $w(x)u_1(x)$, $w(x)u_2(x)$, $w(x)u_3(x)$, $w(x)u_4(x)$, ... With Algorithm C the behavior is different; if the polynomials $u_1(x)$, $u_2(x)$, $u_3(x)$, ... are obtained when Algorithm C is applied to $u(x) = u_1(x)$ and $v(x) = u_2(x)$, and if we assume that $\deg(u_{j+1}) = \deg(u_j) - 1$ (which is almost always true when $j > 1$), then the sequence

$$w(x)u_1(x),\ w(x)u_2(x),\ \ell^2 w(x)u_3(x),\ \ell^4 w(x)u_4(x),\ \ell^6 w(x)u_5(x),\ \ldots \tag{28}$$

is obtained when Algorithm C is applied to $u(x) = w(x)u_1(x)$ and $v(x) = w(x)u_2(x)$, where $\ell = \ell(w)$. (See exercise 13.) Even though these additional ℓ-factors are present, Algorithm C will be superior to Algorithm E, because it is easier to deal with slightly larger polynomials than to calculate primitive parts repeatedly.

Polynomial remainder sequences such as those in Algorithms C and E are not useful merely for finding greatest common divisors; another important application is to the enumeration of real roots, for a given polynomial in a given interval, according to the famous theorem of J. Sturm [*Mém. présentes par divers savants* **6** (Paris, 1835), 271–318]. Let $u(x)$ be a polynomial over the real numbers, having distinct roots. We shall see in the next section that this is the same as saying $\gcd(u(x), u'(x)) = 1$, where $u'(x)$ is the derivative of $u(x)$; accordingly, there is a polynomial remainder sequence proving that $u(x)$ is relatively prime to $u'(x)$. We set $u_0(x) = u(x)$, $u_1(x) = u'(x)$, and (following Sturm) we negate the sign of all remainders:

$$\begin{aligned} c_1 u_0(x) &= u_1(x)q_1(x) - d_1 u_2(x), \\ c_2 u_1(x) &= u_2(x)q_2(x) - d_2 u_3(x), \\ &\vdots \\ c_k u_{k-1}(x) &= u_k(x)q_k(x) - d_k u_{k+1}(x), \end{aligned} \tag{29}$$

for some positive constants c_j and d_j, where $\deg(u_{k+1}) = 0$. We say that the *variation* $V(u, a)$ of $u(x)$ at a is the number of changes of sign in the sequence $u_0(a)$, $u_1(a)$, $\ldots$, $u_{k+1}(a)$, not counting zeros. For example, if the sequence of signs is 0, +, −, −, 0, +, +, −, we have $V(u, a) = 3$. Sturm's theorem asserts that the number of roots of $u(x)$ in the interval $a < x \le b$ is $V(u, a) - V(u, b)$; and the proof is surprisingly short (see exercise 22).

Although Algorithms C and E are interesting, they aren't the whole story. Important alternative ways to calculate polynomial gcd's over the integers are discussed at the end of Section 4.6.2. There is also a general determinant-evaluation algorithm that may be said to include Algorithm C as a special case; see E. H. Bareiss, *Math. Comp.* **22** (1968), 565–578.

EXERCISES

1. [*10*] Compute the pseudo-quotient $q(x)$ and pseudo-remainder $r(x)$, namely, the polynomials satisfying (8), when $u(x) = x^6 + x^5 - x^4 + 2x^3 + 3x^2 - x + 2$ and $v(x) = 2x^3 + 2x^2 - x + 3$, over the integers.

2. [*15*] What is the greatest common divisor of $3x^6 + x^5 + 4x^4 + 4x^3 + 3x^2 + 4x + 2$ and its "reverse" $2x^6 + 4x^5 + 3x^4 + 4x^3 + 4x^2 + x + 3$, modulo 7?

▶ **3.** [*M25*] Show that Euclid's algorithm for polynomials over a field S can be extended to find polynomials $U(x)$ and $V(x)$ over S such that

$$u(x)V(x) + U(x)v(x) = \gcd(u(x), v(x)).$$

(Cf. Algorithm 4.5.2X.) What are the degrees of the polynomials $U(x)$ and $V(x)$ that are computed by this extended algorithm? Prove that if S is the field of rational numbers, and if $u(x) = x^m - 1$ and $v(x) = x^n - 1$, then the extended algorithm yields polynomials $U(x)$ and $V(x)$ having *integer* coefficients. Find $U(x)$ and $V(x)$ when $u(x) = x^{21} - 1$ and $v(x) = x^{13} - 1$.

▶ **4.** [*M30*] Let p be prime, and suppose that Euclid's algorithm applied to the polynomials $u(x)$ and $v(x)$ modulo p yields a sequence of polynomials having respective degrees $m, n, n_1, \ldots, n_t, -\infty$, where $m = \deg(u)$, $n = \deg(v)$, and $n_t \geq 0$. Assume that $m \geq n$. If $u(x)$ and $v(x)$ are monic polynomials, independently and uniformly distributed over all the p^{m+n} pairs of monic polynomials having respective degrees m and n, what are the average values of the three quantities t, $n_1 + \cdots + n_t$, and $(n - n_1)n_1 + \cdots + (n_{t-1} - n_t)n_t$, as functions of m, n, and p? (These three quantities are the fundamental factors in the running time of Euclid's algorithm applied to polynomials modulo p, assuming that division is done by Algorithm D.) [*Hint:* Show that $u(x) \bmod v(x)$ is uniformly distributed and independent of $v(x)$.]

5. [*M22*] What is the probability that $u(x)$ and $v(x)$ are relatively prime modulo p, if $u(x)$ and $v(x)$ are independently and uniformly distributed monic polynomials of degree n?

6. [*M23*] We have seen that Euclid's Algorithm 4.5.2A for integers can be directly adapted to an algorithm for the greatest common divisor of polynomials. Can the "binary gcd algorithm," Algorithm 4.5.2B, be adapted in an analogous way to an algorithm that applies to polynomials?

7. [*M10*] What are the units in the domain of all polynomials over a unique factorization domain S?

▶ **8.** [*M22*] Show that if a polynomial with integer coefficients is irreducible over the domain of integers, it is irreducible when considered as a polynomial over the field of rational numbers.

9. [*M25*] Let $u(x)$ and $v(x)$ be primitive polynomials over a unique factorization domain S. Prove that $u(x)$ and $v(x)$ are relatively prime if and only if there are polynomials $U(x)$ and $V(x)$ over S such that $u(x)V(x) + U(x)v(x)$ is a polynomial of degree zero. [*Hint:* Extend Algorithm E, as Algorithm 4.5.2E is extended in exercise 3.]

10. [*M28*] Prove that the polynomials over a unique factorization domain form a unique factorization domain. [*Hint:* Use the result of exercise 9 to help show that there is at most one kind of factorization possible.]

11. [*M22*] What row names would have appeared in Table 1 if the sequence of degrees had been 9, 6, 5, 2, $-\infty$ instead of 8, 6, 4, 2, 1, 0?

▶ **12.** [*M24*] Let $u_1(x), u_2(x), u_3(x), \ldots$ be a sequence of polynomials obtained during a run of Algorithm C. "Sylvester's matrix" is the square matrix formed from rows A_{n_2-1} through A_0 and B_{n_1-1} through B_0 (in a notation analogous to that of Table 1). Show that if $u_1(x)$ and $u_2(x)$ have a common factor of positive degree, then the determinant of Sylvester's matrix is zero; conversely, given that $\deg(u_k) = 0$ for some k, show that the determinant of Sylvester's matrix is nonzero by deriving a formula for its absolute value in terms of $\ell(u_j)$ and $\deg(u_j)$, $1 \leq j \leq k$.

13. [*M22*] Show that the leading coefficient ℓ of the primitive part of $\gcd(u(x), v(x))$ enters into Algorithm C's polynomial sequence as shown in (28), when $\delta_1 = \delta_2 = \cdots = \delta_{k-1} = 1$. What is the behavior for general δ_j?

14. [*M29*] Let $r(x)$ be the pseudo-remainder when $u(x)$ is pseudo-divided by $v(x)$. If $\deg(u) \geq \deg(v) + 2$ and $\deg(v) \geq \deg(r) + 2$, show that $r(x)$ is a multiple of $\ell(v)$.

15. [*M26*] Prove Hadamard's inequality (25). [*Hint:* Consider the matrix AA^T.]

16. [*HM22*] Let $f(x_1, \ldots, x_n)$ be a multivariate polynomial with real coefficients not all zero, and let a_N be the number of solutions to the equation $f(x_1, \ldots, x_n) = 0$ such that $|x_1| \leq N$, ..., $|x_n| \leq N$, and such that each x_j is an integer. Prove that the roots have zero density, i.e., that $\lim_{N\to\infty} a_N/(2N+1)^n = 0$.

17. [*M32*] (*P. M. Cohn's algorithm for division of string polynomials.*) Let A be an "alphabet," i.e., a set of symbols. A *string* α on A is a sequence of $n \geq 0$ symbols, $\alpha = a_1 \ldots a_n$, where each a_j is in A. The length of α, denoted by $|\alpha|$, is the number n of symbols. A *string polynomial* on A is a finite sum $U = \sum_k r_k \alpha_k$, where each r_k is a nonzero rational number and each α_k is a string on A. The *degree* of U, $\deg(U)$, is defined to be $-\infty$ if $U = 0$ (i.e., if the sum is empty), otherwise $\deg(u) = \max|\alpha_k|$. The sum and product of string polynomials are defined in an obvious manner, e.g., $(\sum_j r_j\alpha_j)(\sum_k s_k\beta_k) = \sum_{j,k} r_j s_k \alpha_j \beta_k$, where the product of two strings is obtained by simply juxtaposing them. For example, if $A = \{a, b\}$, $U = ab + ba - 2a - 2b$, and $V = a + b - 1$, then $\deg(U) = 2$, $\deg(V) = 1$, $V^2 = aa + ab + ba + bb - 2a - 2b + 1$, and $V^2 - U = aa + bb + 1$. Clearly, $\deg(UV) = \deg(U) + \deg(V)$, and $\deg(U+V) \leq \max(\deg(U), \deg(V))$, with equality in the latter formula if $\deg(U) \neq \deg(V)$. (String polynomials may be regarded as ordinary multivariate polynomials over the field of rational numbers, except that the variables are *not commutative* under multiplication. In the conventional language of pure mathematics, the set of string polynomials with the operations defined here is the "free associative algebra" generated by A over the rationals.)

a) Let Q_1, Q_2, U, V be string polynomials with $\deg(U) \geq \deg(V)$ and such that $\deg(Q_1U - Q_2V) < \deg(Q_1U)$. Give an algorithm to find a string polynomial Q such that $\deg(U - QV) < \deg(U)$. (Thus if we are given U and V such that $Q_1U = Q_2V + R$ and $\deg(R) < \deg(Q_1U)$, for some Q_1 and Q_2, then there is a solution to these conditions with $Q_1 = 1$.)

b) Given that U and V are string polynomials with $\deg(V) > \deg(Q_1U - Q_2V)$ for some Q_1 and Q_2, show that the result of (a) can be improved to find a quotient Q such that $U = QV + R$, $\deg(R) < \deg(V)$. (This is the analog of (1) for string polynomials; part (a) showed that we can make $\deg(R) < \deg(U)$, under weaker hypotheses.)

c) A "homogeneous" polynomial is one whose terms all have the same degree (length). If U_1, U_2, V_1, V_2 are homogeneous string polynomials with $U_1V_1 = U_2V_2$ and $\deg(V_1) \geq \deg(V_2)$, show that there is a homogeneous string polynomial U such that $U_2 = U_1U$ and $V_1 = UV_2$.

d) Given that U and V are homogeneous string polynomials with $UV = VU$, prove that there is a homogeneous string polynomial W such that $U = rW^m$, $V = sW^n$ for some integers m, n and rational numbers r, s. Give an algorithm to compute such a W having the largest possible degree. (This algorithm is of interest, for example, when $U = \alpha$ and $V = \beta$ are strings satisfying $\alpha\beta = \beta\alpha$; then W is simply a string γ. When $U = x^m$ and $V = x^n$, the solution of largest degree is

the string $W = x^{\gcd(m,n)}$, so this algorithm includes a gcd algorithm for integers as a special case.)

▶ **18.** [*M24*] (*Euclidean algorithm for string polynomials.*) Let V_1 and V_2 be string polynomials, not both zero, having a "common left multiple." (This means that there exist string polynomials U_1 and U_2, not both zero, such that $U_1V_1 = U_2V_2$.) The purpose of this exercise is to find an algorithm to compute their "greatest common right divisor" $\operatorname{gcrd}(V_1, V_2)$ as well as their "least common left multiple" $\operatorname{lclm}(V_1, V_2)$. The latter quantities are defined as follows: $\operatorname{gcrd}(V_1, V_2)$ is a common right divisor of V_1 and V_2 (that is, $V_1 = W_1 \operatorname{gcrd}(V_1, V_2)$ and $V_2 = W_2 \operatorname{gcrd}(V_1, V_2)$ for some W_1 and W_2), and any common right divisor of V_1 and V_2 is a right divisor of $\operatorname{gcrd}(V_1, V_2)$; $\operatorname{lclm}(V_1, V_2) = Z_1V_1 = Z_2V_2$ for some Z_1 and Z_2, and any common left multiple of V_1 and V_2 is a left multiple of $\operatorname{lclm}(V_1, V_2)$.

For example, let $U_1 = abbbab + abbab - bbab + ab - 1$, $V_1 = babab + abab + ab - b$; $U_2 = abb + ab - b$, $V_2 = babbabab + bababab + babab + abab - babb - 1$. Then we have $U_1V_1 = U_2V_2 = abbbabbabab + abbabbabab + abbbababab + abbababab - bbabbabab + abbbabab - bbababab + 2abbabab - abbbabb + ababab - abbabb - bbabab - babab + bbabb - abb - ab + b$. For these string polynomials it can be shown that $\operatorname{gcrd}(V_1, V_2) = ab + 1$, and $\operatorname{lclm}(V_1, V_2) = U_1V_1$.

The division algorithm of exercise 17 may be restated thus: If V_1 and V_2 are string polynomials, with $V_2 \neq 0$, and if $U_1 \neq 0$ and U_2 satisfy the equation $U_1V_1 = U_2V_2$, then there exist string polynomials Q and R such that

$$V_1 = QV_2 + R, \qquad \text{where } \deg(R) < \deg(V_2).$$

It follows readily that Q and R are uniquely determined; they do not depend on the given U_1 and U_2. Furthermore the result is right-left symmetric, in the sense that

$$U_2 = U_1Q + R', \qquad \text{where } \deg(R') = \deg(U_1) - \deg(V_2) + \deg(R) < \deg(U_1).$$

Show that this division algorithm can be extended to an algorithm that computes $\operatorname{lclm}(V_1, V_2)$ and $\operatorname{gcrd}(V_1, V_2)$; in fact, the extended algorithm finds string polynomials Z_1 and Z_2 such that $Z_1V_1 + Z_2V_2 = \operatorname{gcrd}(V_1, V_2)$. [*Hint:* Use auxiliary variables u_1, u_2, v_1, v_2, w_1, w_2, w_1', w_2', z_1, z_2, z_1', z_2', whose values are string polynomials; start by setting $u_1 \leftarrow U_1$, $u_2 \leftarrow U_2$, $v_1 \leftarrow V_1$, $v_2 \leftarrow V_2$, and throughout the algorithm maintain the conditions

$$\begin{aligned} U_1w_1 + U_2w_2 &= u_1, & z_1V_1 + z_2V_2 &= v_1, \\ U_1w_1' + U_2w_2' &= u_2, & z_1'V_1 + z_2'V_2 &= v_2, \\ u_1z_1 - u_2z_1' &= (-1)^nU_1, & w_1v_1 - w_1'v_2 &= (-1)^nV_1, \\ -u_1z_2 + u_2z_2' &= (-1)^nU_2, & -w_2v_1 + w_2'v_2 &= (-1)^nV_2 \end{aligned}$$

at the nth iteration. This might be regarded as the "ultimate" extension of Euclid's algorithm.]

19. [*M39*] (*Common divisors of square matrices.*) Exercise 18 shows that the concept of greatest common right divisor can be meaningful when multiplication is not commutative. Prove that any two $n \times n$ matrices A and B of integers have a greatest common right matrix divisor D. [*Suggestion:* Design an algorithm whose inputs are A and B, and whose outputs are integer matrices D, P, Q, X, Y, where $A = PD$, $B = QD$, and $D = XA + YB$.] Find a greatest common right divisor of the matrices $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right)$ and $\left(\begin{smallmatrix} 4 & 3 \\ 2 & 1 \end{smallmatrix}\right)$.

20. [*M40*] Investigate the accuracy of Euclid's algorithm: What can be said about calculation of the greatest common divisor of polynomials whose coefficients are floating point numbers?

21. [*M25*] Prove that the computation time required by Algorithm C to compute the gcd of two nth degree polynomials over the integers is $O(n^4(\log Nn)^2)$, if the coefficients of the given polynomials are bounded by N in absolute value.

22. [*M23*] Prove Sturm's theorem. [*Hint:* Some sign sequences are impossible.]

23. [*M22*] Prove that if $u(x)$ in (29) has $\deg(u)$ real roots, then we have $\deg(u_{j+1}) = \deg(u_j) - 1$ for $0 \le j \le k$.

24. [*M21*] Show that (19) simplifies to (20) and (23) simplifies to (24).

25. [*M24*] (W. S. Brown.) Prove that all the polynomials $u_j(x)$ in (16) for $j \ge 3$ are multiples of $\gcd(\ell(u), \ell(v))$, and explain how to improve Algorithm C accordingly.

▶ **26.** [*M26*] The purpose of this exercise is to give an analog for polynomials of the fact that continued fractions with positive integer entries give the best approximations to real numbers (exercise 4.5.3-42).

Let $u(x)$ and $v(x)$ be polynomials over a field, with $\deg(u) > \deg(v)$, and let $a_1(x)$, $a_2(x)$, ... be the quotient polynomials when Euclid's algorithm is applied to $u(x)$ and $v(x)$. For example, the sequence of quotients in (5) and (6) is $9x^2 + 7$, $5x^2 + 5$, $6x^3 + 5x^2 + 6x + 5$, $9x + 12$. We wish to show that the convergents $p_n(x)/q_n(x)$ of the continued fraction $/a_1(x), a_2(x), \ldots/$ are the "best approximations" of low degree to the rational function $v(x)/u(x)$, where we have $p_n(x) = Q_{n-1}(a_2(x), \ldots, a_n(x))$ and $q_n(x) = Q_n(a_1(x), \ldots, a_n(x))$ in terms of the continuant polynomials of Eq. 4.5.3–4. By convention, we let $p_0(x) = q_{-1}(x) = 0$, $p_{-1}(x) = q_0(x) = 1$.

Prove that if $p(x)$ and $q(x)$ are polynomials such that $\deg(q) < \deg(q_n)$ and $\deg(pu - qv) \le \deg(p_{n-1}u - q_{n-1}v)$, for some $n \ge 1$, then $p(x) = cp_{n-1}(x)$ and $q(x) = cq_{n-1}(x)$ for some constant c. In particular, each $q_n(x)$ is a "record-breaking" polynomial in the sense that no nonzero polynomial $q(x)$ of smaller degree can make the quantity $p(x)u(x) - q(x)v(x)$, for any polynomial $p(x)$, achieve a degree as small as $p_n(x)u(x) - q_n(x)v(x)$.

*4.6.2. Factorization of Polynomials

Let us now consider the problem of *factoring* polynomials, not merely finding the greatest common divisor of two or more of them.

Factoring modulo p. As in the case of integer numbers (Sections 4.5.2, 4.5.4), the problem of factoring seems to be more difficult than finding the greatest common divisor. But factorization of polynomials modulo a prime integer p is not as hard to do as we might expect. It is much easier to find the factors of an arbitrary polynomial of degree n, modulo 2, than to use any known method to find the factors of an arbitrary n-bit binary number. This surprising situation is a consequence of an instructive factorization algorithm discovered in 1967 by Elwyn R. Berlekamp [*Bell System Technical J.* **46** (1967), 1853–1859].

Let p be a prime number; all arithmetic on polynomials in the following discussion will be done modulo p. Suppose that someone has given us a polynomial $u(x)$, whose coefficients are chosen from the set $\{0, 1, \ldots, p-1\}$; we may assume

that $u(x)$ is monic. Our goal is to express $u(x)$ in the form

$$u(x) = p_1(x)^{e_1} \dots p_r(x)^{e_r}, \tag{1}$$

where $p_1(x), \dots, p_r(x)$ are distinct, monic, irreducible polynomials.

As a first step, we can use a standard technique to determine whether any of the exponents $e_1, \dots, e_r$ are greater than unity. If

$$u(x) = u_n x^n + \cdots + u_0 = v(x)^2 w(x), \tag{2}$$

then its "derivative" formed in the usual way (but modulo p) is

$$u'(x) = n u_n x^{n-1} + \cdots + u_1 = 2v(x)v'(x)w(x) + v(x)^2 w'(x), \tag{3}$$

and this is a multiple of the squared factor $v(x)$. Therefore our first step in factoring $u(x)$ is to form

$$\gcd\big(u(x), u'(x)\big) = d(x). \tag{4}$$

If $d(x)$ is equal to 1, we know that $u(x)$ is "squarefree," the product of distinct primes $p_1(x) \dots p_r(x)$. If $d(x)$ is not equal to 1 and $d(x) \neq u(x)$, then $d(x)$ is a proper factor of $u(x)$; the relation between the factors of $d(x)$ and the factors of $u(x)/d(x)$ speeds up the factorization process nicely in this case (see exercise 34). Finally, if $d(x) = u(x)$, we must have $u'(x) = 0$; hence the coefficient u_k of x^k is nonzero only when k is a multiple of p. This means that $u(x)$ can be written as a polynomial of the form $v(x^p)$, and in such a case we have

$$u(x) = v(x^p) = \big(v(x)\big)^p; \tag{5}$$

the factorization process can be completed by finding the irreducible factors of $v(x)$ and raising them to the pth power.

Identity (5) may appear somewhat strange to the reader; it is an important fact that is basic to Berlekamp's algorithm and to several other methods we shall discuss. We can prove it as follows: If $v_1(x)$ and $v_2(x)$ are any polynomials modulo p, then

$$\begin{aligned}\big(v_1(x) + v_2(x)\big)^p &= v_1(x)^p + \tbinom{p}{1} v_1(x)^{p-1} v_2(x) \\ &\qquad + \cdots + \tbinom{p}{p-1} v_1(x) v_2(x)^{p-1} + v_2(x)^p \\ &= v_1(x)^p + v_2(x)^p,\end{aligned}$$

since the binomial coefficients $\binom{p}{1}, \dots, \binom{p}{p-1}$ are all multiples of p. Furthermore if a is any integer, we have $a^p \equiv a$ (modulo p) by Fermat's theorem. Therefore when $v(x) = v_m x^m + v_{m-1} x^{m-1} + \cdots + v_0$, we find that

$$\begin{aligned}v(x)^p &= (v_m x^m)^p + (v_{m-1} x^{m-1})^p + \cdots + (v_0)^p \\ &= v_m x^{mp} + v_{m-1} x^{(m-1)p} + \cdots + v_0 \;=\; v(x^p).\end{aligned}$$

The above remarks show that the problem of factoring a polynomial reduces to the problem of factoring a squarefree polynomial. Let us therefore assume that

$$u(x) = p_1(x)p_2(x)\dots p_r(x) \tag{6}$$

is the product of distinct primes. How can we be clever enough to discover the $p_j(x)$'s when only $u(x)$ is given? Berlekamp's idea is to make use of the Chinese remainder theorem, which is valid for polynomials just as it is valid for integers (see exercise 3). If $(s_1, s_2, \dots, s_r)$ is any r-tuple of integers mod p, the Chinese remainder theorem implies that *there is a unique polynomial* $v(x)$ *such that*

$$\begin{gathered} v(x) \equiv s_1 \ \big(\text{modulo } p_1(x)\big), \quad \dots, \quad v(x) \equiv s_r \ \big(\text{modulo } p_r(x)\big), \\ \deg(v) < \deg(p_1) + \deg(p_2) + \cdots + \deg(p_r) = \deg(u). \end{gathered} \tag{7}$$

The notation $g(x) \equiv h(x) \ \big(\text{modulo } f(x)\big)$ that appears here is the same as "$g(x) \equiv h(x)$ (modulo $f(x)$ and p)" in exercise 3.2.2–11, since we are considering polynomial arithmetic modulo p. The polynomial $v(x)$ in (7) gives us a way to get at the factors of $u(x)$, for if $r \geq 2$ and $s_1 \neq s_2$, we will have $\gcd\big(u(x), v(x) - s_1\big)$ divisible by $p_1(x)$ but not by $p_2(x)$.

Since this observation shows that we can get information about the factors of $u(x)$ from appropriate solutions $v(x)$ of (7), let us analyze (7) more closely. In the first place we can observe that the polynomial $v(x)$ satisfies the condition $v(x)^p \equiv s_j^p = s_j \equiv v(x) \ \big(\text{modulo } p_j(x)\big)$ for $1 \leq j \leq r$, therefore

$$v(x)^p \equiv v(x) \ \big(\text{modulo } u(x)\big), \qquad \deg(v) < \deg(u). \tag{8}$$

In the second place we have the basic polynomial identity

$$x^p - x \equiv (x-0)(x-1)\dots\big(x-(p-1)\big) \quad (\text{modulo } p) \tag{9}$$

(see exercise 6); hence

$$v(x)^p - v(x) = \big(v(x)-0\big)\big(v(x)-1\big)\dots\big(v(x)-(p-1)\big) \tag{10}$$

is an identity for any polynomial $v(x)$, when we are working modulo p. If $v(x)$ satisfies (8), it follows that $u(x)$ divides the left-hand side of (10), so every irreducible factor of $u(x)$ must divide one of the p relatively prime factors of the right-hand side of (10). In other words, *all* solutions of (8) must have the form of (7), for some $s_1, s_2, \dots, s_r$; *there are exactly* p^r *solutions of* (8).

The solutions $v(x)$ to congruence (8) therefore provide a key to the factorization of $u(x)$. It may seem harder to find all solutions to (8) than to factor $u(x)$ in the first place, but in fact this is not true, since the set of solutions to (8) is closed under addition. Let $\deg(u) = n$; we can construct the $n \times n$ matrix

$$Q = \begin{pmatrix} q_{0,0} & q_{0,1} & \cdots & q_{0,n-1} \\ \vdots & \vdots & & \vdots \\ q_{n-1,0} & q_{n-1,1} & \cdots & q_{n-1,n-1} \end{pmatrix} \tag{11}$$

where

$$x^{pk} \equiv q_{k,n-1}x^{n-1} + \cdots + q_{k,1}x + q_{k,0} \pmod{u(x)}. \tag{12}$$

Then $v(x) = v_{n-1}x^{n-1} + \cdots + v_1x + v_0$ is a solution to (8) if and only if

$$(v_0, v_1, \ldots, v_{n-1})Q = (v_0, v_1, \ldots, v_{n-1}); \tag{13}$$

for the latter equation holds if and only if

$$v(x) = \sum_j v_j x^j = \sum_j \sum_k v_k q_{k,j} x^j \equiv \sum_k v_k x^{pk} = v(x^p) \equiv v(x)^p \pmod{u(x)}.$$

Berlekamp's factoring algorithm therefore proceeds as follows:

B1. Ensure that $u(x)$ is squarefree; i.e., if $\gcd(u(x), u'(x)) \neq 1$, reduce the problem of factoring $u(x)$, as stated earlier in this section.

B2. Form the matrix Q defined by (11) and (12). This can be done in one of two ways, depending on whether or not p is very large, as explained below.

B3. "Triangularize" the matrix $Q - I$, where $I = (\delta_{ij})$ is the $n \times n$ identity matrix, finding its rank $n - r$ and finding linearly independent vectors $v^{[1]}$, $\ldots$, $v^{[r]}$ such that $v^{[j]}(Q-I) = (0, 0, \ldots, 0)$ for $1 \leq j \leq r$. (The first vector $v^{[1]}$ may always be taken as $(1, 0, \ldots, 0)$, representing the trivial solution $v^{[1]}(x) = 1$ to (8). The "triangularization" needed in this step can be done using appropriate column operations, as explained in Algorithm N below.) *At this point, r is the number of irreducible factors of $u(x)$,* because the solutions to (8) are the p^r polynomials corresponding to the vectors $t_1v^{[1]} + \cdots + t_rv^{[r]}$ for all choices of integers $0 \leq t_1, \ldots, t_r < p$. Therefore if $r = 1$ we know that $u(x)$ is irreducible, and the procedure terminates.

B4. Calculate $\gcd(u(x), v^{[2]}(x)-s)$ for $0 \leq s < p$, where $v^{[2]}(x)$ is the polynomial represented by vector $v^{[2]}$. The result will be a nontrivial factorization of $u(x)$, because $v^{[2]}(x) - s$ is nonzero and has degree less than $\deg(u)$, and by exercise 7 we have

$$u(x) = \prod_{0 \leq s < p} \gcd(v(x) - s, u(x)) \tag{14}$$

whenever $v(x)$ satisfies (8).

If the use of $v^{[2]}(x)$ does not succeed in splitting $u(x)$ into r factors, further factors can be obtained by calculating $\gcd(v^{[k]}(x) - s, w(x))$ for $0 \leq s < p$ and all factors $w(x)$ found so far, for $k = 3, 4, \ldots$, until r factors are obtained. (If we choose $s_i \neq s_j$ in (7), we obtain a solution $v(x)$ to (8) that distinguishes $p_i(x)$ from $p_j(x)$; some $v^{[k]}(x) - s$ will be divisible by $p_i(x)$ and not by $p_j(x)$, so this procedure will eventually find all of the factors.)

If p is 2 or 3, the calculations of this step are quite efficient; but if p is more than 25, say, there is a much better way to proceed, as we shall see later. ∎

As an example of this procedure, let us now determine the factorization of

$$u(x) = x^8 + x^6 + 10x^4 + 10x^3 + 8x^2 + 2x + 8 \tag{15}$$

modulo 13. (This polynomial appears in several of the examples in Section 4.6.1.) A quick calculation using Algorithm 4.6.1E shows that $\gcd(u(x), u'(x)) = 1$; therefore $u(x)$ is squarefree, and we turn to step B2. Step B2 involves calculating the Q matrix, which in this case is an 8×8 array. The first row of Q is always $(1, 0, 0, \ldots, 0)$, representing the polynomial $x^0 \bmod u(x) = 1$. The second row represents $x^{13} \bmod u(x)$, and, in general, $x^k \bmod u(x)$ may readily be determined as follows (for relatively small values of k): If

$$u(x) = x^n + u_{n-1}x^{n-1} + \cdots + u_1 x + u_0$$

and if

$$x^k \equiv a_{k,n-1}x^{n-1} + \cdots + a_{k,1}x + a_{k,0} \ \ (\text{modulo } u(x)),$$

then

$$\begin{aligned} x^{k+1} &\equiv a_{k,n-1}x^n + \cdots + a_{k,1}x^2 + a_{k,0}x \\ &\equiv a_{k,n-1}(-u_{n-1}x^{n-1} - \cdots - u_1 x - u_0) + a_{k,n-2}x^{n-1} + \cdots + a_{k,0}x \\ &= a_{k+1,n-1}x^{n-1} + \cdots + a_{k+1,1}x + a_{k+1,0}, \end{aligned}$$

where

$$a_{k+1,j} = a_{k,j-1} - a_{k,n-1}u_j. \tag{16}$$

In this formula $a_{k,-1}$ is treated as zero, so that $a_{k+1,0} = -a_{k,n-1}u_0$. The simple "shift register" recurrence (16) makes it easy to calculate x^1, x^2, x^3, ... mod $u(x)$. Inside a computer, this calculation is of course generally done by maintaining a one-dimensional array $(a_{n-1}, \ldots, a_1, a_0)$ and repeatedly setting $t \leftarrow a_{n-1}$, $a_{n-1} \leftarrow (a_{n-2} - tu_{n-1}) \bmod p$, ..., $a_1 \leftarrow (a_0 - tu_1) \bmod p$, and $a_0 \leftarrow (-tu_0) \bmod p$. (We have seen similar procedures in connection with random number generation; cf. Eq. 3.2.2–10.) For the example polynomial $u(x)$ in (15), we obtain the following sequence of coefficients of $x^k \bmod u(x)$, using arithmetic modulo 13:

k	$a_{k,7}$	$a_{k,6}$	$a_{k,5}$	$a_{k,4}$	$a_{k,3}$	$a_{k,2}$	$a_{k,1}$	$a_{k,0}$
0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	1	0
2	0	0	0	0	0	1	0	0
3	0	0	0	0	1	0	0	0
4	0	0	0	1	0	0	0	0
5	0	0	1	0	0	0	0	0
6	0	1	0	0	0	0	0	0
7	1	0	0	0	0	0	0	0
8	0	12	0	3	3	5	11	5
9	12	0	3	3	5	11	5	0
10	0	4	3	2	8	0	2	8
11	4	3	2	8	0	2	8	0
12	3	11	8	12	1	2	5	7
13	11	5	12	10	11	7	1	2

Therefore the second row of Q is $(2,1,7,11,10,12,5,11)$. Similarly we may determine $x^{26} \bmod u(x)$, ..., $x^{91} \bmod u(x)$, and we find that

$$Q = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 7 & 11 & 10 & 12 & 5 & 11 \\ 3 & 6 & 4 & 3 & 0 & 4 & 7 & 2 \\ 4 & 3 & 6 & 5 & 1 & 6 & 2 & 3 \\ 2 & 11 & 8 & 8 & 3 & 1 & 3 & 11 \\ 6 & 11 & 8 & 6 & 2 & 7 & 10 & 9 \\ 5 & 11 & 7 & 10 & 0 & 11 & 7 & 12 \\ 3 & 3 & 12 & 5 & 0 & 11 & 9 & 12 \end{pmatrix},$$

$$Q - I = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 7 & 11 & 10 & 12 & 5 & 11 \\ 3 & 6 & 3 & 3 & 0 & 4 & 7 & 2 \\ 4 & 3 & 6 & 4 & 1 & 6 & 2 & 3 \\ 2 & 11 & 8 & 8 & 2 & 1 & 3 & 11 \\ 6 & 11 & 8 & 6 & 2 & 6 & 10 & 9 \\ 5 & 11 & 7 & 10 & 0 & 11 & 6 & 12 \\ 3 & 3 & 12 & 5 & 0 & 11 & 9 & 11 \end{pmatrix}. \tag{17}$$

That finishes step B2; the next step of Berlekamp's procedure requires finding the "null space" of $Q - I$. In general, suppose that A is an $n \times n$ matrix over a field, whose rank $n - r$ is to be determined; suppose further that we wish to determine linearly independent vectors $v^{[1]}$, $v^{[2]}$, ..., $v^{[r]}$ such that $v^{[1]}A = v^{[2]}A = \cdots = v^{[r]}A = (0, \ldots, 0)$. An algorithm for this calculation can be based on the observation that any column of A may be multiplied by a nonzero quantity, and any multiple of one of its columns may be added to a different column, without changing the rank or the vectors $v^{[1]}, \ldots, v^{[r]}$. (These transformations amount to replacing A by AB, where B is a nonsingular matrix.) The following well-known "triangularization" procedure may therefore be used.

Algorithm N (*Null space algorithm*). Let A be an $n \times n$ matrix, whose elements a_{ij} belong to a field and have subscripts in the range $0 \le i, j < n$. This algorithm outputs r vectors $v^{[1]}$, ..., $v^{[r]}$, which are linearly independent over the field and satisfy $v^{[j]}A = (0, \ldots, 0)$, where $n - r$ is the rank of A.

N1. [Initialize.] Set $c_0 \leftarrow c_1 \leftarrow \cdots \leftarrow c_{n-1} \leftarrow -1$, $r \leftarrow 0$. (During the calculation we will have $c_j \ge 0$ only if $a_{c_j j} = -1$ and all other entries of row c_j are zero.)

N2. [Loop on k.] Do step N3 for $k = 0$, 1, ..., $n - 1$, and then terminate the algorithm.

N3. [Scan row for dependence.] If there is some j in the range $0 \le j < n$ such that $a_{kj} \ne 0$ and $c_j < 0$, then do the following: Multiply column j of A by $-1/a_{kj}$ (so that a_{kj} becomes equal to -1); then add a_{ki} times column j to column i for all $i \ne j$; finally set $c_j \leftarrow k$. (Since it is not difficult to show that $a_{sj} = 0$ for all $s < k$, these operations have no effect on rows 0, 1, ..., $k - 1$ of A.)

On the other hand, if there is no j in the range $0 \le j < n$ such that $a_{kj} \neq 0$ and $c_j < 0$, then set $r \leftarrow r + 1$ and output the vector

$$v^{[r]} = (v_0, v_1, \ldots, v_{n-1})$$

defined by the rule

$$v_j = \begin{cases} a_{ks}, & \text{if } c_s = j \ge 0; \\ 1, & \text{if } j = k; \\ 0, & \text{otherwise.} \end{cases} \tag{18}$$

∎

An example will reveal the mechanism of this algorithm. Let A be the matrix $Q - I$ of (17) over the field of integers modulo 13. When $k = 0$, we output the vector $v^{[1]} = (1, 0, 0, 0, 0, 0, 0, 0)$. When $k = 1$, we may take j in step N3 to be either 0, 2, 3, 4, 5, 6, or 7; the choice here is completely arbitrary, although it affects the particular vectors that are chosen to be output by the algorithm. For hand calculation, it is most convenient to pick $j = 5$, since $a_{15} = 12 = -1$ already; the column operations of step N3 then change A to the matrix

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \textcircled{12} & 0 & 0 \\ 11 & 6 & 5 & 8 & 1 & 4 & 1 & 7 \\ 3 & 3 & 9 & 5 & 9 & 6 & 6 & 4 \\ 4 & 11 & 2 & 6 & 12 & 1 & 8 & 9 \\ 5 & 11 & 11 & 7 & 10 & 6 & 1 & 10 \\ 1 & 11 & 6 & 1 & 6 & 11 & 9 & 3 \\ 12 & 3 & 11 & 9 & 6 & 11 & 12 & 2 \end{pmatrix}.$$

(The circled element in column "5", row "1", is used here to indicate that $c_5 = 1$. Remember that Algorithm N numbers the rows and columns of the matrix starting with 0, not 1.) When $k = 2$, we may choose $j = 4$ and proceed in a similar way, obtaining the following matrices, which all have the same null space as $Q - I$:

$k = 2$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \textcircled{12} & 0 & 0 \\ 0 & 0 & 0 & 0 & \textcircled{12} & 0 & 0 & 0 \\ 8 & 1 & 3 & 11 & 4 & 9 & 10 & 6 \\ 2 & 4 & 7 & 1 & 1 & 5 & 9 & 3 \\ 12 & 3 & 0 & 5 & 3 & 5 & 4 & 5 \\ 0 & 1 & 2 & 5 & 7 & 0 & 3 & 0 \\ 11 & 6 & 7 & 0 & 7 & 0 & 6 & 12 \end{pmatrix}$$

$k = 3$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \textcircled{12} & 0 & 0 \\ 0 & 0 & 0 & 0 & \textcircled{12} & 0 & 0 & 0 \\ 0 & \textcircled{12} & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 9 & 8 & 9 & 11 & 8 & 8 & 5 \\ 1 & 10 & 4 & 11 & 4 & 4 & 0 & 0 \\ 5 & 12 & 12 & 7 & 3 & 4 & 6 & 7 \\ 2 & 7 & 2 & 12 & 9 & 11 & 11 & 2 \end{pmatrix}$$

$k = 4$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & (12) & 0 & 0 \\ 0 & 0 & 0 & 0 & (12) & 0 & 0 & 0 \\ 0 & (12) & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & (12) \\ 1 & 10 & 4 & 11 & 4 & 4 & 0 & 0 \\ 8 & 2 & 6 & 10 & 11 & 11 & 0 & 9 \\ 1 & 6 & 4 & 11 & 2 & 0 & 0 & 10 \end{pmatrix}$$

$k = 5$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & (12) & 0 & 0 \\ 0 & 0 & 0 & 0 & (12) & 0 & 0 & 0 \\ 0 & (12) & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & (12) \\ (12) & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 5 & 5 & 0 & 9 \\ 12 & 9 & 0 & 0 & 11 & 9 & 0 & 10 \end{pmatrix}$$

Now every column that has no circled entry is completely zero; so when $k = 6$ and $k = 7$ the algorithm outputs two more vectors, namely

$$v^{[2]} = (0, 5, 5, 0, 9, 5, 1, 0), \qquad v^{[3]} = (0, 9, 11, 9, 10, 12, 0, 1).$$

From the form of matrix A after $k = 5$, it is evident that these vectors satisfy the equation $vA = (0, \ldots, 0)$. Since the computation has produced three linearly independent vectors, $u(x)$ must have exactly three irreducible factors.

Finally we can go to step B4 of the factoring procedure. The calculation of $\gcd\big(u(x), v^{[2]}(x) - s\big)$ for $0 \le s < 13$, where $v^{[2]}(x) = x^6 + 5x^5 + 9x^4 + 5x^2 + 5x$, gives $x^5 + 5x^4 + 9x^3 + 5x + 5$ as the answer when $s = 0$, and $x^3 + 8x^2 + 4x + 12$ when $s = 2$; the gcd is unity for other values of s. Therefore $v^{[2]}(x)$ gives us only two of the three factors. Turning to $\gcd\big(v^{[3]}(x) - s,\ x^5 + 5x^4 + 9x^3 + 5x + 5\big)$, where $v^{[3]}(x) = x^7 + 12x^5 + 10x^4 + 9x^3 + 11x^2 + 9x$, we obtain the value $x^4 + 2x^3 + 3x^2 + 4x + 6$ when $s = 6$, $x + 3$ when $s = 8$, and unity otherwise. Thus the complete factorization is

$$u(x) = (x^4 + 2x^3 + 3x^2 + 4x + 6)(x^3 + 8x^2 + 4x + 12)(x + 3). \tag{19}$$

Let us now estimate the running time of Berlekamp's method when an nth degree polynomial is factored modulo p. First assume that p is relatively small, so that the four arithmetic operations can be done modulo p in essentially a fixed length of time. (Division modulo p can be converted to multiplication, by storing a table of reciprocals as suggested in exercise 9; for example, when working modulo 13, we have $\frac{1}{2} = 7$, $\frac{1}{3} = 9$, etc.) The computation in step B1 takes $O(n^2)$ units of time; step B2 takes $O(pn^2)$. For step B3 we use Algorithm N, which requires $O(n^3)$ units of time at most. Finally, in step B4 we can observe that the calculation of $\gcd\big(f(x), g(x)\big)$ by Euclid's algorithm takes $O\big(\deg(f)\deg(q)\big)$ units of time; hence the calculation of $\gcd\big(v^{[j]}(x) - s,\ w(x)\big)$ for fixed j and s and for all factors $w(x)$ of $u(x)$ found so far takes $O(n^2)$ units. Step B4 therefore requires $O(prn^2)$ units of time at most. *Berlekamp's procedure factors an arbitrary polynomial of degree n, modulo p, in $O(n^3 + prn^2)$ steps*, when p is a small prime; and exercise 5 shows that the average number of factors, r, is approximately $\ln n$.

Thus the algorithm is much faster than any known methods of factoring n-digit numbers in the p-ary number system.

Of course, when n and p are small, a trial-and-error factorization procedure analogous to Algorithm 4.5.4A will be even faster than Berlekamp's method. Exercise 1 implies that it is a good idea to cast out factors of small degree first when p is small, before going to any more complicated procedure, even when n is large.

When p is large, a different implementation of Berlekamp's procedure would be used for the calculations. Division modulo p would not be done with an auxiliary table of reciprocals; instead the method of exercise 4.5.2–15, which takes $O((\log p)^2)$ steps, would probably be used. Then step B1 would take $O(n^2(\log p)^2)$ units of time; similarly, step B3 takes $O(n^3(\log p)^2)$. In step B2, we can form $x^p \bmod u(x)$ in a more efficient way than (16) when p is large: Section 4.6.3 shows that this value can essentially be obtained by using $O(\log p)$ operations of "squaring $\bmod u(x)$," i.e., going from $x^k \bmod u(x)$ to $x^{2k} \bmod u(x)$. The squaring operation is relatively easy to perform if we first make an auxiliary table of $x^m \bmod u(x)$ for $m = n, n+1, \ldots, 2n-2$; if

$$x^k \bmod u(x) = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0,$$

then

$$x^{2k} \bmod u(x) = \left(c_{n-1}^2 x^{2n-2} + \cdots + (c_1c_0 + c_1c_0)x + c_0^2\right) \bmod u(x),$$

where $x^{2n-2}, \ldots, x^n$ can be replaced by polynomials in the auxiliary table. The total time to compute $x^p \bmod u(x)$ comes to $O(n^2(\log p)^3)$ units, and we obtain the second row of Q. To get further rows of Q, we can compute $x^{2p} \bmod u(x)$, $x^{3p} \bmod u(x)$, ..., simply by multiplying repeatedly by $x^p \bmod u(x)$, in a fashion analogous to squaring $\bmod u(x)$; step B2 is completed in $O(n^2(\log p)^3)$ units of time. Thus steps B1, B2, and B3 take a total of $O(n^2(\log p)^3 + n^3(\log p)^2)$ units of time; these three steps tell us the number of factors of $u(x)$.

But when p is large and we get to step B4, we are asked to calculate a greatest common divisor for p different values of s, and that is out of the question if p is even moderately large. This hurdle was first surmounted by Hans Zassenhaus [*J. Number Theory* **1** (1969), 291–311], who showed how to determine all of the "useful" values of s (see exercise 14); but an even better way to proceed was found by Zassenhaus and Cantor in 1980. If $v(x)$ is any solution to (8), we know that $u(x)$ divides $v(x)^p - v(x) = v(x) \cdot \left(v(x)^{(p-1)/2} + 1\right) \cdot \left(v(x)^{(p-1)/2} - 1\right)$. This suggests that we calculate

$$\gcd\left(u(x), v(x)^{(p-1)/2} - 1\right); \tag{20}$$

with a little bit of luck, (20) will be a nontrivial factor of $u(x)$. In fact, we can determine exactly how much luck is involved, by considering (7). Let $v(x) \equiv s_j$ (modulo $p_j(x)$) for $1 \le j \le r$; then $p_j(x)$ divides $v(x)^{(p-1)/2} - 1$ if and only if

$s_j^{(p-1)/2} \equiv 1$ (modulo p). We know that exactly $(p-1)/2$ of the integers s in the range $0 \le s < p$ satisfy $s^{(p-1)/2} \equiv 1$ (modulo p), hence about half of the $p_j(x)$ will appear in the gcd (20). More precisely, if $v(x)$ is a random solution of (8), where all p^r solutions are equally likely, the probability that the gcd (20) equals $u(x)$ is exactly

$$\big((p-1)/2p\big)^r,$$

and the probability that it equals 1 is $\big((p+1)/2p\big)^r$. The probability that a nontrivial factor will be obtained is therefore

$$1-\left(\frac{p-1}{2p}\right)^r-\left(\frac{p+1}{2p}\right)^r = 1-\frac{1}{2^{r-1}}\left(1+\binom{r}{2}p^{-2}+\binom{r}{4}p^{-4}+\cdots\right) \ge \frac{4}{9},$$

for all $r \ge 2$ and $p \ge 3$.

It is therefore a good idea to replace step B4 by the following procedure, unless p is quite small: Set $v(x) \leftarrow a_1 v^{[1]}(x) + a_2 v^{[2]}(x) + \cdots + a_r v^{[r]}(x)$, where the coefficients a_j are randomly chosen in the range $0 \le a_j < p$. Let the current partial factorization of $u(x)$ be $u_1(x)\ldots u_t(x)$ where t is initially 1. Compute

$$g_i(x) = \gcd\big(u_i(x), v(x)^{(p-1)/2} - 1\big)$$

for all i such that $\deg(u_i) > 1$; replace $u_i(x)$ by $g_i(x)\cdot\big(u_i(x)/g_i(x)\big)$ and increase the value of t, whenever a nontrivial gcd is found. Repeat this process for different choices of $v(x)$ until $t = r$.

If we assume (as we may) that only $O(\log r)$ random solutions $v(x)$ to (8) will be needed, we can give an upper bound on the time required to perform this alternative to step B4. It takes $O\big(r(\log p)^2\big)$ steps to compute $v(x)$; and if $\deg(u_i) = d$, it takes $O\big(d^2(\log p)^3\big)$ steps to compute $v(x)^{(p-1)/2} \bmod u_i(x)$ and $O\big(d^2(\log p)^2\big)$ further steps to compute $\gcd\big(u_i(x), v(x)^{(p-1)/2} - 1\big)$. Thus the total time is $O(n^2(\log p)^3 \log r)$.

For further discussion, see the articles by E. R. Berlekamp, *Math. Comp.* **24** (1970), 713–735, and Robert T. Moenck, *Math. Comp.* **31** (1977), 235–250.

Distinct-degree factorization. We shall now turn to a somewhat simpler way to find factors modulo p. The ideas we have studied so far in this section involve many instructive insights into computational algebra, so the author does not apologize to the reader for presenting them; but it turns out that the problem of factorization modulo p can actually be solved without relying on so many concepts.

In the first place we can make use of the fact that an irreducible polynomial $q(x)$ of degree d is a divisor of $x^{p^d} - x$, and it is not a divisor of $x^{p^c} - x$ for $c < d$; see exercise 16. We can therefore cast out the irreducible factors of each degree separately, by adopting the following strategy.

D1. Rule out squared factors, as in Berlekamp's method. Also set $v(x) \leftarrow u(x)$, $w(x) \leftarrow$ "x", and $d \leftarrow 0$. (Here $v(x)$ and $w(x)$ are variables that have polynomials as values.)

D2. (At this point $w(x) = x^{p^d} \bmod v(x)$; all of the irreducible factors of $v(x)$ are distinct and have degree $> d$.) If $d+1 > \frac{1}{2}\deg(v)$, the procedure terminates since we either have $v(x) = 1$ or $v(x)$ is irreducible. Otherwise increase d by 1 and replace $w(x)$ by $w(x)^p \bmod v(x)$.

D3. Find $g_d(x) = \gcd\big(w(x) - x, v(x)\big)$. (This is the product of all the irreducible factors of $u(x)$ whose degree is d.) If $g_d(x) \neq 1$, replace $v(x)$ by $v(x)/g_d(x)$ and $w(x)$ by $w(x) \bmod v(x)$; and if the degree of $g_d(x)$ is greater than d, use the algorithm below to find its factors. Return to step D2. ∎

This procedure determines the product of all irreducible factors of each degree d, and therefore it tells us how many factors there are of each degree. Since the three factors of our example polynomial (19) have different degrees, they would all be discovered without any need to factorize the polynomials $g_d(x)$.

The distinct degree factorization technique was known to several people in 1960 [cf. S. W. Golomb, L. R. Welch, A. Hales, "On the factorization of trinomials over GF(2)," Jet Propulsion Laboratory memo 20–189 (July 14, 1959)], but there seem to be no references to it in the "open literature." Previous work by Š. Schwarz, *Quart. J. Math.*, Oxford (2) **7** (1956), 110–124, had shown how to determine the number of irreducible factors of each degree, but not their product, using the matrix Q.

To complete the method, we need a way to split the polynomial $g_d(x)$ into its irreducible factors when $\deg(g_d) > d$. Michael Rabin pointed out in 1976 that this can be done by doing arithmetic in the field of p^d elements. David G. Cantor and Hans Zassenhaus discovered in 1979 that there is an even simpler way to proceed, based on the following identity: If p is any odd prime, we have

$$g_d(x) = \gcd\big(g_d(x), t(x)\big) \cdot \gcd\big(g_d(x), t(x)^{(p^d-1)/2} + 1\big) \cdot \gcd\big(g_d(x), t(x)^{(p^d-1)/2} - 1\big) \tag{21}$$

for all polynomials $t(x)$, since $t(x)^{p^d} - t(x)$ is a multiple of all irreducible polynomials of degree d. (We may regard $t(x)$ as an element of the field of size p^d, when that field consists of all polynomials modulo an irreducible $f(x)$ as in exercise 16.) Now exercise 29 shows that $\gcd\big(g_d(x), t(x)^{(p^d-1)/2}\big)$ will be a nontrivial factor of $g_d(x)$ about 50 per cent of the time, when $t(x)$ is a random polynomial of degree $\leq 2d - 1$; hence it will not take long to discover all of the factors. We may assume without loss of generality that $t(x)$ is monic, since integer multiples of $t(x)$ make no difference except possibly to change $t(x)^{(p^d-1)/2}$ into its negative. Thus in the case $d = 1$, we can take $t(x) = x + s$, where s is chosen at random. [See *SIAM J. Computing* **9** (1980), 273–280; *Math. Comp.*, to appear.]

Sometimes this procedure will in fact succeed for $d > 1$ when only linear polynomials $t(x)$ are used. For example, there are eight irreducible polynomials $f(x)$ of degree 3, modulo 3, and they will all be distinguished by calculating

$\gcd\big(f(x), (x+s)^{13}-1\big)$ for $0 \le s < 3$:

$f(x)$	$s=0$	$s=1$	$s=2$
x^3+2x+1	1	1	1
x^3+2x+2	$f(x)$	$f(x)$	$f(x)$
x^3+x^2+2	$f(x)$	$f(x)$	1
x^3+x^2+x+2	$f(x)$	1	$f(x)$
x^3+x^2+2x+1	1	$f(x)$	$f(x)$
x^3+2x^2+1	1	$f(x)$	1
x^3+2x^2+x+1	1	1	$f(x)$
x^3+2x^2+2x+2	$f(x)$	1	1

Exercise 31 contains a partial explanation of why linear polynomials can be effective; however, when the number of irreducible polynomials of degree d exceeds 2^p, it is clear that there will exist irreducibles that cannot be distinguished by linear choices of $t(x)$.

An alternative to (21) that works when $p = 2$ is discussed in exercise 30.

Factoring over the integers. It is somewhat more difficult to find the complete factorization of polynomials with integer coefficients when we are *not* working modulo p, but some reasonably efficient methods are available for this purpose.

Isaac Newton gave a method for finding linear and quadratic factors of polynomials with integer coefficients in his *Arithmetica Universalis* (1707). This method was extended by an astronomer named Friedrich von Schubert in 1793, who showed how to find all factors of degree n in a finite number of steps; see M. Cantor, *Geschichte der Mathematik* **4** (Leipzig: Teubner, 1908), 136–137. L. Kronecker rediscovered von Schubert's method independently about 90 years later; but unfortunately the method is very inefficient when n is five or more. Much better results can be obtained with the help of the "mod p" factorization methods presented above.

Suppose that we want to find the irreducible factors of a given polynomial

$$u(x) = u_n x^n + u_{n-1}x^{n-1} + \cdots + u_0, \qquad u_n \neq 0,$$

over the integers. As a first step, we can divide by the greatest common divisor of the coefficients; this leaves us with a *primitive* polynomial. We may also assume that $u(x)$ is squarefree, by dividing out $\gcd(u(x), u'(x))$ as in exercise 34.

Now if $u(x) = v(x)w(x)$, where each of these polynomials has integer coefficients, we obviously have $u(x) \equiv v(x)w(x)$ (modulo p) for all primes p, so there is a nontrivial factorization modulo p unless p divides $\ell(u)$. An efficient algorithm for factoring $u(x)$ modulo p can therefore be used in an attempt to reconstruct possible factorizations of $u(x)$ over the integers.

For example, let

$$u(x) = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5. \tag{22}$$

We have seen above in (19) that

$$u(x) \equiv (x^4+2x^3+3x^2+4x+6)(x^3+8x^2+4x+12)(x+3) \text{ (modulo 13)}; \quad (23)$$

and the complete factorization of $u(x)$ modulo 2 shows one factor of degree 6 and another of degree 2 (see exercise 10). From (23) we can see that $u(x)$ has no factor of degree 2, so it must be irreducible over the integers.

This particular example was perhaps too simple; experience shows that most irreducible polynomials can be recognized as such by examining their factors modulo a few primes, but it is *not* always so easy to establish irreducibility. For example, there are polynomials that can be properly factored modulo p for all primes p, with consistent degrees of the factors, yet they are irreducible over the integers (see exercise 12).

Almost all polynomials are irreducible over the integers, as shown in exercise 27. But we usually aren't trying to factor a random polynomial; there is probably some reason to expect a nontrivial factor or else the calculation would not have been attempted in the first place. We need a method that identifies factors when they are there.

In general if we try to find the factors of $u(x)$ by considering its behavior modulo different primes, the results will not be easy to combine; for example, if $u(x)$ actually is the product of four quadratic polynomials, it will be hard to match up their images with respect to different prime moduli. Therefore it is desirable to stick to a single prime and to see how much mileage we can get out of it, once we feel that the factors modulo this prime have the right degrees.

One idea is to work modulo a very *large* prime p, big enough so that the coefficients in any true factorization $u(x) = v(x)w(x)$ over the integers must actually lie between $-p/2$ and $p/2$. Then all possible integer factors can be "read off" from the mod p factors we know how to compute.

Exercise 20 shows how to obtain fairly good bounds on the coefficients of polynomial factors. For example, if (22) were reducible it would have a factor $v(x)$ of degree ≤ 4, and the coefficients of v would be at most 34 in magnitude by the results of that exercise. So all potential factors of $u(x)$ will be fairly evident if we work modulo any prime $p > 68$. Indeed, the complete factorization modulo 71 is

$$(x+12)(x+25)(x^2-13-7)(x^4-24x^3-16x^2+31x-12),$$

and we see immediately that none of these polynomials are factors of (22) over the integers since their constant terms do not divide 5; furthermore there is no way to obtain a divisor of (22) by grouping two of these factors, since none of the conceivable constant terms 12×25, -12×7, $12 \times (-12)$ is congruent to ± 1 or ± 5 (modulo 71).

Incidentally, it is not trivial to obtain good bounds on the coefficients of polynomial factors, since a lot of cancellation can occur when polynomials are multiplied. For example, the innocuous-looking polynomial $x^n - 1$ has irreducible

factors whose coefficients exceed $\exp(n^{1/\lg\lg n})$ for infinitely many n. [See R. C. Vaughan, *Michigan Math. J.* **21** (1974), 289–295.] The factorization of $x^n - 1$ is discussed in exercise 32.

Instead of using a large prime p, which might have to be truly enormous if $u(x)$ has large degree or large coefficients, we can also make use of small p, provided that $u(x)$ is squarefree mod p. For in this case, an important construction introduced by K. Hensel [*Theorie der Algebraischen Zahlen* (Leipzig: Teubner, 1908), Chapter 4] can be used to extend a factorization modulo p in a unique way to a factorization modulo p^e for arbitrarily high e. Hensel's method is described in exercise 22; if we apply it to (23) with $p = 13$ and $e = 2$, we obtain the unique factorization

$$u(x) \equiv (x - 36)(x^3 - 18x^2 + 82x - 66)(x^4 + 54x^3 - 10x^2 + 69x + 84)$$

(modulo 169). Calling these factors $v_1(x)v_3(x)v_4(x)$, we see that $v_1(x)$ and $v_3(x)$ are not factors of $u(x)$ over the integers, nor is their product $v_1(x)v_3(x)$ when the coefficients have been reduced modulo 169 to the range $(-\frac{169}{2}, \frac{169}{2})$. Thus we have exhausted all possibilities, proving once again that $u(x)$ is irreducible over the integers—this time using only its factorization modulo 13.

The example we have been considering is atypical in one important respect: We have been factoring the *monic* polynomial $u(x)$ in (22), so we could assume that all its factors were monic. What should we do if $u_n > 1$? In such a case, the leading coefficients of all but one of the polynomial factors can be varied almost arbitrarily modulo p^e; we certainly don't want to try all possibilities. Perhaps the reader has already noticed this problem. Fortunately there is a simple way out: the factorization $u(x) = v(x)w(x)$ implies a factorization $u_n u(x) = v_1(x)w_1(x)$ where $\ell(v_1) = \ell(w_1) = u_n = \ell(u)$. ("Do you mind if I multiply your polynomial by its leading coefficient before factoring it?") We can proceed essentially as above, but using $p^e > 2B$ where B now bounds the maximum coefficient for factors of $u_n u(x)$ instead of $u(x)$.

Putting these observations all together results in the following procedure:

F1. Find the unique squarefree factorization

$$u(x) \equiv \ell(u)v_1(x)\dots v_r(x) \text{ (modulo } p^e),$$

where p^e is sufficiently large as explained above, and where the $v_j(x)$ are monic. (This will be possible for all but a few primes p, see exercise 23.) Also set $d \leftarrow 1$.

F2. For every combination of factors $v(x) = v_{i_1}(x)\dots v_{i_d}(x)$, with $i_1 = 1$ if $d = \frac{1}{2}r$, form the unique polynomial $\bar{v}(x) \equiv \ell(u)v(x)$ (modulo p^e) whose coefficients all lie in the interval $[-\frac{1}{2}p^e, \frac{1}{2}p^e)$. If $\bar{v}(x)$ divides $\ell(u)u(x)$, output the factor $\mathrm{pp}(\bar{v}(x))$, divide $u(x)$ by this factor, and remove the corresponding $v_i(x)$ from the list of factors modulo p^e; decrease r by the number of factors removed, and terminate the algorithm if $d > \frac{1}{2}r$.

F3. Increase d by 1, and return to F2 if $d > \frac{1}{2}r$. ∎

At the conclusion of this process, the current value of $u(x)$ will be the final irreducible factor of the originally given polynomial. Note that if $|u_0| < |u_n|$, it is preferable to do all of the work with the reverse polynomial $u_0x^n + \cdots + u_n$, whose factors are the reverses of the factors of $u(x)$.

The procedure as stated requires $p^e > 2B$, where B is a bound on the coefficients of *any* divisor of $u_n u(x)$, but we can use a much smaller value of B if we only guarantee it to be valid for divisors of degree $\leq \frac{1}{2}\deg(u)$. In this case the divisibility test in step F2 should be applied to $w(x) = v_1(x)\ldots v_r(x)/v(x)$ instead of $v(x)$, whenever $\deg(v) > \frac{1}{2}\deg(u)$.

The above algorithm contains an obvious bottleneck: We may have to test as many as 2^{r-1} potential factors $v(x)$. The average value of 2^r in a random situation is about n, or perhaps $n^{1.5}$ (see exercise 5), but in nonrandom situations we will want to speed up this part of the routine as much as we can. One way to rule out spurious factors quickly is to compute the trailing coefficient $\bar{v}(0)$ first, continuing only if this divides $\ell(u)u(0)$; the complication explained in the preceding paragraph does not have to be considered unless this divisibility condition is satisfied, since such a test is valid even when $\deg(v) > \frac{1}{2}\deg(u)$.

Another important way to speed up the procedure is to reduce r so that it tends to reflect the true number of factors. The distinct degree factorization algorithm above can be applied for various small primes p_j, thus obtaining for each prime a set D_j of possible degrees of factors modulo p_j; see exercise 26. We can represent D_j as a string of n binary bits. Now we compute the intersection $\bigcap D_j$, namely the logical "and" of these bit strings, and we perform step F2 only for $i_1 + \cdots + i_d \in \bigcap D_j$. Furthermore p is chosen to be that p_j having the smallest value of r. This technique is due to David R. Musser, whose experience suggests trying about five primes p_j (see *JACM* **25** (1978), 271–282). Of course we would stop immediately if the current $\bigcap D_j$ shows that $u(x)$ is irreducible.

Musser has given a complete discussion of a factorization method similar to the steps above, in *JACM* **22** (1975), 291–308. The procedure above incorporates an improvement suggested in 1978 by G. E. Collins, namely to look for trial divisors by taking combinations of d factors at a time rather than combinations of total degree d. This improvement is important because of the statistical behavior of the modulo-p factors of polynomials that are irreducible over the rationals (cf. exercise 37).

Greatest common divisors. Similar techniques can be used to calculate greatest common divisors of polynomials: If $\gcd\bigl(u(x), v(x)\bigr) = d(x)$ over the integers, and if $\gcd\bigl(u(x), v(x)\bigr) = q(x)$ (modulo p) where $q(x)$ is monic, then $d(x)$ is a common divisor of $u(x)$ and $v(x)$ modulo p; hence

$$d(x) \text{ divides } q(x) \quad (\text{modulo } p). \tag{24}$$

If p does not divide the leading coefficients of both u and v, it does not divide the leading coefficient of d; in such a case $\deg(d) \leq \deg(q)$. When $q(x) = 1$ for such a prime p, we must therefore have $\deg(d) = 0$, and $d(x) = \gcd\bigl(\mathrm{cont}(u), \mathrm{cont}(v)\bigr)$. This justifies the remark made in Section 4.6.1 that the simple computation

of $\gcd(u(x), v(x))$ modulo 13 in 4.6.1–6 is enough to prove that $u(x)$ and $v(x)$ are relatively prime over the integers; the comparatively laborious calculations of Algorithm 4.6.1E or Algorithm 4.6.1C are unnecessary. Since two random primitive polynomials are almost always relatively prime over the integers, and since they are relatively prime modulo p with probability $1 - 1/p$, it is usually a good idea to do the computations modulo p.

As remarked before, we need good methods also for the nonrandom polynomials that arise in practice. Therefore we wish to sharpen our techniques and discover how to find $\gcd(u(x), v(x))$ in general, over the integers, based entirely on information that we obtain working modulo primes p. We may assume that $u(x)$ and $v(x)$ are primitive.

Instead of calculating $\gcd(u(x), v(x))$ directly, it will be convenient to search instead for the polynomial

$$\bar{d}(x) = c \cdot \gcd(u(x), v(x)), \tag{25}$$

where the constant c is chosen so that

$$\ell(\bar{d}) = \gcd(\ell(u), \ell(v)). \tag{26}$$

This condition will always hold for suitable c, since the leading coefficient of any common divisor of $u(x)$ and $v(x)$ must be a divisor of $\gcd(\ell(u), \ell(v))$. Once $\bar{d}(x)$ has been found satisfying these conditions, we can readily compute $\operatorname{pp}(\bar{d}(x))$, which is the true greatest common divisor of $u(x)$ and $v(x)$. Condition (26) is convenient since it avoids the uncertainty of unit multiples of the gcd; we have used essentially the same idea to control the leading coefficients in our factorization routine.

If p is a sufficiently large prime, based on the bounds for coefficients in exercise 20 applied either to $\ell(\bar{d})u(x)$ or $\ell(\bar{d})v(x)$, let us compute the unique polynomial $\bar{q}(x) \equiv \ell(\bar{d})q(x)$ (modulo p) having all coefficients in $[-\frac{1}{2}p, \frac{1}{2}p)$. When $\operatorname{pp}(\bar{q}(x))$ divides both $u(x)$ and $v(x)$, it must equal $\gcd(u(x), v(x))$ because of (24). On the other hand if it does not divide both $u(x)$ and $v(x)$ we must have $\deg(q) > \deg(d)$. A study of Algorithm 4.6.1E reveals that this will be the case only if p divides the leading coefficient of one of the nonzero remainders computed by that algorithm with exact integer arithmetic; otherwise Euclid's algorithm modulo p deals with precisely the same sequence of polynomials as Algorithm 4.6.1E except for nonzero constant multiples (modulo p). So only a small number of "unlucky" primes can cause us to miss the gcd, and we will soon find a lucky prime if we keep trying.

If the bound on coefficients is so large that single-precision primes p are insufficient, we can compute $\bar{d}(x)$ modulo several primes p until it has been determined via the Chinese remainder algorithm of Section 4.3.2. This approach, which is due to W. S. Brown and G. E. Collins, has been described in detail by Brown in *JACM* **18** (1971), 478–504. Alternatively, as suggested by J. Moses and D. Y. Y. Yun [*Proc. ACM Conf.* **28** (1973), 159–166], we can use Hensel's

method to determine $\bar{d}(x)$ modulo p^e for sufficiently large e. Hensel's construction appears to be computationally superior to the Chinese remainder approach; but it is valid directly only when

$$\gcd\big(d(x),\, u(x)/d(x)\big) = 1 \qquad \text{or} \qquad \gcd\big(d(x),\, v(x)/d(x)\big) = 1, \tag{27}$$

since the idea is to apply the techniques of exercise 22 to one of the factorizations $\ell(\bar{d})u(x) \equiv \bar{q}(x)u_1(x)$ or $\ell(\bar{d})v(x) \equiv \bar{q}(x)v_1(x)$ (modulo p). Exercises 34 and 35 show that it is possible to arrange things so that (27) holds whenever necessary.

The gcd algorithms sketched here are significantly faster than those of Section 4.6.1 except when the polynomial remainder sequence is very short. Perhaps the best general procedure would be to start with the computation of $\gcd\big(u(x), v(x)\big)$ modulo a fairly small prime p, not a divisor of both $\ell(u)$ and $\ell(v)$. If the result $q(x)$ is 1, we're done; if it has high degree, we use Algorithm 4.6.1C; otherwise we use one of the above methods, first computing a bound for the coefficients of $\bar{d}(x)$ based on the coefficients of $u(x)$ and $v(x)$, and on the (small) degree of $q(x)$. As in the factorization problem, we should apply this procedure to the reverses of $u(x), v(x)$ and reverse the result, if the trailing coefficients are simpler than the leading ones.

Multivariate polynomials. Similar techniques lead to useful algorithms for factorization or gcd calculations on multivariate polynomials with integer coefficients. It is convenient to deal with the polynomial $u(x_1, \ldots, x_t)$ by working modulo the irreducible polynomials $x_2 - a_2, \ldots, x_t - a_t$, which play the rôle of p in the above discussion. Since $v(x) \bmod (x - a) = v(a)$, the value of $u(x_1, \ldots, x_t)$ is the univariate polynomial $u(x_1, a_2, \ldots, a_t)$. When the integers $a_2, \ldots, a_t$ have been chosen so that $u(x_1, a_2, \ldots, a_t)$ has the same degree in x_1 as $u(x_1, x_2, \ldots, x_t)$, an appropriate generalization of Hensel's construction will "lift" squarefree factorizations of this univariate polynomial to factorizations modulo $(x_2 - a_2)^{n_2}$, $\ldots$, $(x_t - a_t)^{n_t}$, where n_j is the degree of x_j in u; at the same time we can also work modulo an appropriate integer prime p. As many as possible of the a_j should be zero, so that sparseness of the intermediate results is retained. For details, see P. S. Wang, *Math. Comp.* **32** (1978), 1215–1231, in addition to the papers by Musser and by Moses and Yun cited earlier.

EXERCISES

▶ **1.** [*M24*] Let p be prime. What is the probability that a random polynomial of degree n has a linear factor (a factor of degree 1), when $n \geq p$? Show that this probability is more than $\frac{1}{2}$. (Assume that each of the p^n monic polynomials modulo p is equally likely.) What is the average number of linear factors?

▶ **2.** [*M25*] (a) Show that any monic polynomial $u(x)$, over a unique factorization domain, may be expressed uniquely in the form

$$u(x) = v(x)^2 w(x),$$

where $w(x)$ is squarefree (has no factor of positive degree of the form $d(x)^2$) and both $v(x)$ and $w(x)$ are monic. (b) (E. R. Berlekamp.) How many monic polynomials of degree n are squarefree modulo p, when p is prime?

3. [*M25*] Let $u_1(x), \ldots, u_r(x)$ be polynomials over a field S, with $u_j(x)$ relatively prime to $u_k(x)$ for all $j \neq k$. For any given polynomials $w_1(x), \ldots, w_r(x)$ over S, prove that there is a unique polynomial $v(x)$ over S such that

$$\deg(v) < \deg(u_1) + \cdots + \deg(u_r)$$

and

$$v(x) \equiv w_j(x) \pmod{u_j(x)}$$

for $1 \leq j \leq r$. (Compare with Theorem 4.3.2C.)

4. [*HM28*] Let a_{np} be the number of monic irreducible polynomials of degree n, modulo a prime p. Find a formula for the generating function $G_p(z) = \sum_n a_{np} z^n$. [*Hint:* Prove the following identity connecting power series: $f(z) = \sum_{j \geq 1} g(z^j)/j^t$ if and only if $g(z) = \sum_{n \geq 1} \mu(n) f(z^n)/n^t$.] What is $\lim_{p \to \infty} a_{np}/p^n$?

5. [*HM30*] Let A_{np} be the average number of factors of a randomly selected polynomial of degree n, modulo a prime p. Show that $\lim_{p \to \infty} A_{np} = H_n$. What is the limiting average value of 2^r, when there are r factors?

6. [*M21*] (J. L. Lagrange, 1771.) Prove the congruence (9). [*Hint:* Factor $x^p - x$ in the field of p elements.]

7. [*M22*] Prove Eq. (14).

8. [*HM20*] How can we be sure that the vectors output by Algorithm N are linearly independent?

9. [*20*] Explain how to construct a table of reciprocals mod 101 in a simple way, given that 2 is a primitive root of 101.

▶ **10.** [*21*] Find the complete factorization of the polynomial $u(x)$ in (22), modulo 2, using Berlekamp's procedure.

11. [*22*] Find the complete factorization of the polynomial $u(x)$ in (22), modulo 5.

▶ **12.** [*M22*] Use Berlekamp's algorithm to determine the number of factors of $u(x) = x^4 + 1$, modulo p, for all primes p. [*Hint:* Consider the cases $p = 2$, $p = 8k + 1$, $p = 8k + 3$, $p = 8k + 5$, $p = 8k + 7$ separately; what is the matrix Q? You need not discover the factors; just determine how many there are.]

13. [*M25*] Give an explicit formula for the factors of $x^4 + 1$, modulo p, for all odd primes p, in terms of the quantities $\sqrt{-1}$, $\sqrt{2}$, $\sqrt{-2}$ (if such square roots exist modulo p).

14. [*M25*] (H. Zassenhaus.) Let $v(x)$ be a solution to (8), and let $w(x) = \prod(x - s)$ where the product is over all $0 \leq s < p$ such that $\gcd(u(x), v(x) - s) \neq 1$. Explain how to compute $w(x)$, given $u(x)$ and $v(x)$. [*Hint:* Eq. (14) implies that $w(x)$ is the polynomial of least degree such that $u(x)$ divides $w(v(x))$.]

▶ **15.** [*M27*] Design an algorithm to calculate the "square root" of a given integer u modulo a given prime p, i.e., to find an integer v such that $v^2 \equiv u \pmod{p}$ whenever such a v exists. Your algorithm should be efficient even for very large primes p. (Note that a solution to this problem leads to a procedure for solving any given quadratic equation modulo p, using the quadratic formula in the usual way.)

16. [*M30*] (a) Given that $f(x)$ is an irreducible polynomial modulo a prime p, of degree n, prove that the p^n polynomials of degree less than n form a field under arithmetic modulo $f(x)$ and p. (*Note:* The existence of irreducible polynomials of each degree is proved in exercise 4; therefore fields with p^n elements exist for all primes p and all $n \geq 1$.) (b) Show that any field with p^n elements has a "primitive root" element ξ such that the elements of the field are $\{0, 1, \xi, \xi^2, \ldots, \xi^{p^n-2}\}$. [*Hint:* Exercise 3.2.1.2–16 provides a proof in the special case $n = 1$.] (c) If $f(x)$ is an irreducible polynomial modulo p, of degree n, prove that $x^{p^m} - x$ is divisible by $f(x)$ if and only if m is a multiple of n. (It follows that we can test irreducibility rather quickly: A given nth degree polynomial $f(x)$ is irreducible modulo p if and only if $x^{p^n} - x$ is divisible by $f(x)$ and $\gcd(x^{p^{n/q}} - x,\ f(x)) = 1$ for all primes q dividing n.)

17. [*M23*] Let F be a field with 13^2 elements. How many elements of F have order f, for each integer f with $1 \leq f < 13^2$? (The "order" of an element a is the least positive integer m such that $a^m = 1$.)

▶ **18.** [*M25*] Let $u(x) = u_n x^n + \cdots + u_0, u_n \neq 0$, be a primitive polynomial with integer coefficients, and let $v(x)$ be the monic polynomial defined by

$$v(x) = u_n^{n-1} \cdot u(x/u_n) = x^n + u_{n-1}x^{n-1} + u_{n-2}u_n x^{n-2} + \cdots u_0 u_n^{n-1}.$$

(a) Given that $v(x)$ has the complete factorization $p_1(x) \ldots p_r(x)$ over the integers, where each $p_j(x)$ is monic, what is the complete factorization of $u(x)$ over the integers? (b) If $w(x) = x^m + w_{m-1}x^{m-1} + \cdots + w_0$ is a factor of $v(x)$, prove that w_k is a multiple of u_n^{m-1-k} for $0 \leq k < m$.

19. [*M20*] (*Eisenstein's criterion.*) Perhaps the best-known class of irreducible polynomials over the integers was introduced by G. Eisenstein in *J. für die reine und angew. Math.* **39** (1850), 166–167: Let p be prime and let $u(x) = u_n x^n + \cdots + u_0$ have the following properties: (i) u_n is not divisible by p; (ii) $u_{n-1}, \ldots, u_0$ are divisible by p; (iii) u_0 is not divisible by p^2. Show that $u(x)$ is irreducible over the integers.

20. [*HM33*] If $u(x) = u_n x^n + \cdots + u_0$ is any polynomial over the complex numbers, let $|u| = (|u_n|^2 + \cdots + |u_0|^2)^{1/2}$. (a) Let $g(x) = (x - \alpha)u(x)$ and $h(x) = (\bar{\alpha}x - 1)u(x)$, where α is any complex number and $\bar{\alpha}$ is its complex conjugate. Prove that $|g| = |h|$. (b) Let the complete factorization of $u(x)$ over the complex numbers be $u_n(x - \alpha_1)\ldots(x - \alpha_n)$, and write $M(u) = \prod_{1 \leq j \leq n} \max(1, |\alpha_j|)$. Prove that $M(u) \leq |u|/|u_n|$. (c) Show that $|u_j| \leq |u_n|\left(\binom{n-1}{j} M(u) + \binom{n-1}{j-1}\right)$ for $0 \leq j \leq n$. (d) Combine these results to prove that if $u(x) = v(x)w(x)$ and $v(x) = v_m x^m + \cdots + v_0$, where u, v, w all have integer coefficients, then the coefficients of v are bounded by

$$|v_j| \leq \binom{m-1}{j}|u| + \binom{m-1}{j-1}|u_n|.$$

21. [*HM30*] The purpose of this exercise is to obtain useful bounds on the coefficients of *multivariate* polynomial factors over the integers. Given a polynomial $u(x_1, \ldots, x_t)$ over the complex numbers, let $|u|$ be $(\sum |u_{j_1 \ldots j_t}|^2)^{1/2}$ summed over all the coefficients. Let $e(x) = e^{2\pi i x}$. (a) Prove that

$$|u|^2 = \int_0^1 \cdots \int_0^1 \left|u(e(\theta_1), \ldots, e(\theta_t))\right|^2 d\theta_t \ldots d\theta_1.$$

(b) Let $u(x) = v(x)w(x)$, where $\deg(v) = m$ and $\deg(w) = k$. Use the results proved in exercise 20 to show that we always have $|v||w| \le f(m,k)^{1/2}|u|$, where $f(m,k) = \binom{2m}{m}\binom{2k}{k}$. (c) Let $u(x_1,\ldots,x_t) = v(x_1,\ldots,x_t)w(x_1,\ldots,x_t)$, where v and w have the respective degrees m_j and k_j in x_j. Prove that

$$|v||w| \le \big(f(m_1,k_1)\ldots f(m_t,k_t)\big)^{1/2}|u|.$$

▶ **22.** [*M24*] (*Hensel's Lemma.*) Let $u(x)$, $v_e(x)$, $w_e(x)$, $a(x)$, $b(x)$ be polynomials with integer coefficients, satisfying the relations

$$u(x) \equiv v_e(x)w_e(x) \pmod{p^e}, \qquad a(x)v_e(x) + b(x)w_e(x) \equiv 1 \pmod{p},$$

where p is prime, $e \ge 1$, $v_e(x)$ is monic, $\deg(a) < \deg(w_e)$, $\deg(b) < \deg(v_e)$, and $\deg(u) = \deg(v_e) + \deg(w_e)$. Show how to compute polynomials $v_{e+1}(x) \equiv v_e(x)$ and $w_{e+1}(x) \equiv w_e(x) \pmod{p^e}$, satisfying the same conditions with e increased by 1. Furthermore, prove that $v_{e+1}(x)$ and $w_{e+1}(x)$ are unique, modulo p^{e+1}.

Use your method for $p = 2$ to prove that (22) is irreducible over the integers, starting with its factorization mod 2 found in exercise 10. (Note that Euclid's extended algorithm, exercise 4.6.1–3, will get the process started for $e = 1$.)

23. [*HM23*] Let $u(x)$ be a squarefree polynomial with integer coefficients. Prove that there are only finitely many primes p such that $u(x)$ is not squarefree modulo p.

24. [*M20*] The text speaks only of factorization over the integers, not over the field of rational numbers. Explain how to find the complete factorization of a polynomial with rational coefficients, over the field of rational numbers.

25. [*M25*] What is the complete factorization of $x^5 + x^4 + x^2 + x + 2$ over the field of rational numbers?

26. [*20*] Let $d_1, \ldots, d_r$ be the degrees of the irreducible factors of $u(x)$ modulo p, with proper multiplicity, so that $d_1 + \cdots + d_r = n = \deg(u)$. Explain how to compute the set $\{\deg(v) \mid u(x) \equiv v(x)w(x) \pmod{p} \text{ for some } v(x), w(x)\}$ by performing $O(r)$ operations on binary bit strings of length n.

27. [*HM30*] Prove that a random primitive polynomial over the integers is "almost always" irreducible, in some appropriate sense.

28. [*M25*] The distinct-degree factorization procedure is "lucky" when there is at most one irreducible polynomial of each degree d; then $g_d(x)$ never needs to be broken into factors. What is the probability of such a lucky circumstance, when factoring a random polynomial of degree n, modulo p, for fixed n as $p \to \infty$?

29. [*M22*] Let $g(x)$ be a product of two or more distinct irreducible polynomials of degree d, modulo an odd prime p. Prove that $\gcd(g(x), t(x)^{(p^d-1)/2} - 1)$ will be a proper factor of $g(x)$ with probability $\ge 1/2 - 1/(2p^d)$, for any fixed $g(x)$, when $t(x)$ is selected at random from among the p^{2d} polynomials of degree $< 2d$ modulo p.

30. [*M25*] Prove that if $q(x)$ is an irreducible polynomial of degree d, modulo p, and if $t(x)$ is any polynomial, then the value of $\big(t(x) + t(x)^p + t(x)^{p^2} + \cdots + t(x)^{p^{d-1}}\big) \bmod q(x)$ is an integer (i.e., a polynomial of degree ≤ 0). Use this fact to design a probabilistic algorithm for factoring a product $g_d(x)$ of degree-d irreducibles, analogous to (21), for the case $p = 2$.

31. [*HM30*] Let p be an odd prime and let $d \geq 1$. Show that there exists a number $n(p,d)$ having the following two properties: (a) For all integers t, exactly $n(p,d)$ irreducible polynomials $q(x)$ of degree d, modulo p, satisfy $(x+t)^{(p^d-1)/2} \bmod q(x) = 1$. (b) For all integers $0 \leq t_1 < t_2 < p$, exactly $n(p,d)$ irreducible polynomials $q(x)$ of degree d, modulo p, satisfy $(x+t_1)^{(p^d-1)/2} \bmod q(x) = (x+t_2)^{(p^d-1)/2} \bmod q(x)$.

▶ **32.** [*M30*] (*Cyclotomic polynomials.*) Let $\Psi_n(x) = \prod_{1\leq k\leq n,\, \gcd(k,n)=1}(x-\omega^k)$, where $\omega = e^{2\pi i/n}$; thus, the roots of $\Psi_n(x)$ are the complex nth roots of unity that aren't mth roots for $m < n$. (a) Prove that $\Psi_n(x)$ is a polynomial with integer coefficients, and that

$$x^n - 1 = \prod_{d\backslash n} \Psi_d(x); \qquad \Psi_n(x) = \prod_{d\backslash n} (x^d-1)^{\mu(n/d)}.$$

(Cf. exercises 4.5.2–10(b) and 4.5.3–28(c).) (b) Prove that $\Psi_n(x)$ is irreducible over the integers, hence the above formula is the complete factorization of $x^n - 1$ over the integers. [*Hint:* If $f(x)$ is an irreducible factor of $\Psi_n(x)$ over the integers, and if ζ is a complex number with $f(\zeta) = 0$, prove that $f(\zeta^p) = 0$ for all primes p not dividing n. It may help to use the fact that $x^n - 1$ is squarefree modulo p for all such primes.] (c) Discuss the calculation of $\Psi_n(x)$, and tabulate the values for $n \leq 15$.

33. [*M18*] True or false: If $u(x) \neq 0$ and the complete factorization of $u(x)$ modulo p is $p_1(x)^{e_1} \ldots p_r(x)^{e_r}$, then $u(x)/\gcd(u(x), u'(x)) = p_1(x) \ldots p_r(x)$.

▶ **34.** [*M25*] (*Squarefree factorization.*) It is clear that any primitive polynomial of a unique factorization domain can be expressed in the form $u(x) = u_1(x)u_2(x)^2u_3(x)^3\ldots$, where the polynomials $u_i(x)$ are squarefree and relatively prime to each other. This representation, in which $u_j(x)$ is the product of all the irreducible polynomials that divide $u(x)$ exactly j times, is unique except for unit multiples; and it is a useful way to represent polynomials that participate in multiplication, division, and gcd operations.

Let $\mathrm{GCD}(u(x), v(x))$ be a procedure that returns three answers:

$$\mathrm{GCD}(u(x), v(x)) = (d(x), u(x)/d(x), v(x)/d(x)), \quad \text{where } d(x) = \gcd(u(x), v(x)).$$

The modular method described in the text following Eq. (25) always ends with a trial division of $u(x)/d(x)$ and $v(x)/d(x)$, to make sure that no "unlucky prime" has been used, so the quantities $u(x)/d(x)$ and $v(x)/d(x)$ are byproducts of the gcd computation; thus we can compute $\mathrm{GCD}(u(x), v(x))$ essentially as fast as $\gcd(u(x), v(x))$ when we are using a modular method.

Devise a procedure that obtains the squarefree representation $(u_1(x), u_2(x), \ldots)$ of a given primitive polynomial $u(x)$ over the integers. Your algorithm should perform exactly e computations of a GCD, where e is the largest subscript with $u_e(x) \neq 1$; furthermore, each GCD calculation should satisfy (27), so that Hensel's construction can be used.

35. [*M22*] (D. Y. Y. Yun.) Design an algorithm that computes the squarefree representation $(w_1(x), w_2(x), \ldots)$ of $w(x) = \gcd(u(x), v(x))$ over the integers, given the squarefree representations $(u_1(x), u_2(x), \ldots)$ and $(v_1(x), v_2(x), \ldots)$ of $u(x)$ and $v(x)$.

36. [*M27*] Extend the procedure of exercise 34 so that it will obtain the squarefree representation $(u_1(x), u_2(x), \ldots)$ of a given polynomial $u(x)$ when the coefficient arithmetic is performed modulo p.

37. [*HM24*] (George E. Collins.) Let $d_1, \ldots, d_r$ be positive integers whose sum is n, and let p be prime. What is the probability that the irreducible factors of a random nth-degree integer polynomial $u(x)$ have degrees $d_1, \ldots, d_r$, when it is completely factored modulo p? Show that this probability is asymptotically the same as the probability that a random permutation on n elements has cycles of lengths $d_1, \ldots, d_r$.

38. [*M48*] (V. R. Pratt.) If possible, find a way to construct proofs of irreducibility for all polynomials that are irreducible over the integers, so that the length of proof is at most a polynomial in $\deg(u)$ and the length of its coefficients. (Only a bound on the *length* of proof is requested here, as in exercise 4.5.4–17, not a bound on the time needed to find such a proof.)

4.6.3. Evaluation of Powers

In this section we shall study the interesting problem of computing x^n efficiently, given x and n, where n is a positive integer. Suppose, for example, that we need to compute x^{16}; we could simply start with x and multiply by x fifteen times. But it is possible to obtain the same answer with only four multiplications, if we repeatedly take the square of each partial result, successively forming x^2, x^4, x^8, x^{16}.

The same idea applies, in general, to any value of n, in the following way: Write n in the binary number system (suppressing zeros at the left). Then replace each "1" by the pair of letters SX, replace each "0" by S, and cross off the "SX" that now appears at the left. The result is a rule for computing x^n, if "S" is interpreted as the operation of *squaring*, and if "X" is interpreted as the operation of *multiplying by* x. For example, if $n = 23$, its binary representation is 10111; so we form the sequence SX S SX SX SX and remove the leading SX to obtain the rule SSXSXSX. This rule states that we should "square, square, multiply by x, square, multiply by x, square, and multiply by x"; in other words, we should successively compute x^2, x^4, x^5, x^{10}, x^{11}, x^{22}, x^{23}.

This "binary method" is easily justified by a consideration of the sequence of exponents in the calculation: If we reinterpret "S" as the operation of multiplying by 2 and "X" as the operation of adding 1, and if we start with 1 instead of x, the rule will lead to a computation of n because of the properties of the binary number system. The method is quite ancient; it appeared before 200 B.C. in Pingala's Hindu classic Chandah-sûtra [see B. Datta and A. N. Singh, *History of Hindu Mathematics* **1** (Bombay: 1935), 76]; however, there seem to be no other references to this method outside of India during the next 1000 years. A clear discussion of how to compute 2^n efficiently for arbitrary n was given by al-Uqlîdisî of Damascus in 952 A.D.; see *The Arithmetic of al-Uqlîdisî* by A. S. Saidan (Dordrecht: D. Reidel, 1975), 341–342, where the general ideas are illustrated for $n = 51$. See also al-Bîrûnî's *Chronology of Ancient Nations*, ed. and translated by E. Sachau (London: 1879), 132–136; this eleventh-century Arabic work had great influence.

The S-and-X binary method for obtaining x^n requires no temporary storage except for x and the current partial result, so it is well suited for incorpora-

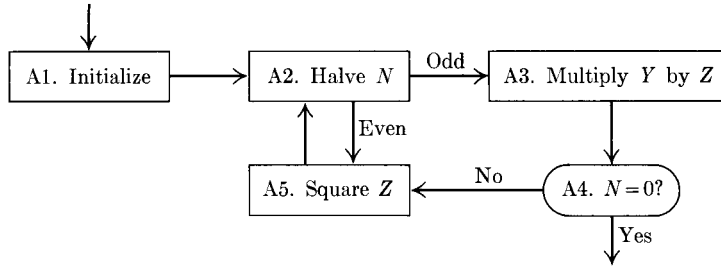


Fig. 12. Evaluation of x^n, based on a right-to-left scan of the binary notation for n.

tion in the hardware of a binary computer. The method can also be readily programmed for either binary or decimal computers; but it requires that the binary representation of n be scanned from left to right, while it is usually more convenient to do this from right to left. For example, with a binary computer we can shift the binary value of n to the right one bit at a time until zero is reached; with a decimal computer we can divide by 2 (or, equivalently, multiply by 5 or $\frac{1}{2}$) to deduce the binary representation from right to left. Therefore the following algorithm, based on a right-to-left scan of the number, is often more convenient:

Algorithm A (*Right-to-left binary method for exponentiation*). This algorithm evaluates x^n, where n is a positive integer. (Here x belongs to any algebraic system in which an associative multiplication, with identity element 1, has been defined.)

A1. [Initialize.] Set $N \leftarrow n$, $Y \leftarrow 1$, $Z \leftarrow x$.

A2. [Halve N.] (At this point, $x^n = Y \cdot Z^N$.) Set $N \leftarrow \lfloor N/2 \rfloor$, and at the same time determine whether N was even or odd. If N was even, skip to step A5.

A3. [Multiply Y by Z.] Set $Y \leftarrow Z$ times Y.

A4. [$N = 0$?] If $N = 0$, the algorithm terminates, with Y as the answer.

A5. [Square Z.] Set $Z \leftarrow Z$ times Z, and return to step A2. ∎

As an example of Algorithm A, consider the steps in the evaluation of x^{23}:

	N	Y	Z
After step A1	23	1	x
After step A4	11	x	x
After step A4	5	x^3	x^2
After step A4	2	x^7	x^4
After step A4	0	x^{23}	x^{16}

A MIX program corresponding to Algorithm A appears in exercise 2.

The great calculator al-Kashî stated Algorithm A about 1414 A.D. [*Istoriko-Mat. Issledovaniîa* **7** (1954), 256–257]. The method is closely related to a procedure for multiplication that was actually used by Egyptian mathematicians as early as 1800 B.C.; for if we change step A3 to "$Y \leftarrow Y + Z$" and step A5 to "$Z \leftarrow Z + Z$", and if we set Y to zero instead of unity in step A1, the algorithm terminates with $Y = nx$. This is a practical method for multiplication by hand, since it involves only the simple operations of doubling, halving, and adding. It is often called the "Russian peasant method" of multiplication, since Western visitors to Russia in the nineteenth century found the method in wide use there.

The number of multiplications required by Algorithm A is $\lfloor \lg n \rfloor + \nu(n)$, where $\nu(n)$ is the number of ones in the binary representation of n. This is one more multiplication than the left-to-right binary method mentioned at the beginning of this section would require, due to the fact that the first execution of step A3 is simply a multiplication by unity.

Because of the bookkeeping time required by this algorithm, the binary method is usually not of importance for small values of n, say $n \leq 10$, unless the time for a multiplication is comparatively large. If the value of n is known in advance, the left-to-right binary method is preferable. In some situations, such as the calculation of $x^n \bmod u(x)$ discussed in Section 4.6.2, it is much easier to multiply by x than to perform a general multiplication or to square a value, so binary methods for exponentiation are primarily suited for quite large n in such cases. If we wish to calculate the exact multiple-precision value of x^n, when x is an integer > 1, binary methods are no help unless n is so huge that the high-speed multiplication routines of Section 4.3.3 are involved; and such applications are rare. Similarly, binary methods are usually inappropriate for raising a polynomial to a power; see R. J. Fateman, *SIAM J. Computing* **3** (1974), 196–213, for a discussion of the extensive literature on polynomial exponentiation. The point of these remarks is that binary methods are nice, but not a panacea. They are most applicable when the time to multiply $x^j \cdot x^k$ is essentially independent of j and k (e.g., for floating point multiplication, or multiplication mod m); in such cases the running time is reduced from order n to order $\log n$.

Fewer multiplications. Several authors have published statements (without proof) that the binary method actually gives the *minimum* possible number of multiplications. But this is not true. The smallest counterexample is $n = 15$, when the binary method needs six multiplications, yet we can calculate $y = x^3$ in two multiplications and $x^{15} = y^5$ in three more, achieving the desired result with only five multiplications. Let us now discuss some other procedures for evaluating x^n, for applications when n is known in advance (e.g., within an optimizing compiler).

The *factor method* is based on a factorization of n. If $n = pq$, where p is the smallest prime factor of n and $q > 1$, we may calculate x^n by first calculating x^p and then raising this quantity to the qth power. If n is prime, we may calculate x^{n-1} and multiply by x. And, of course, if $n = 1$, we have x^n with no calculation at all. Repeated application of these rules gives a procedure for evaluating x^n,

given any value of n. For example, if we want to calculate x^{55}, we first evaluate $y = x^5 = x^4x = (x^2)^2x$; then we form $y^{11} = y^{10}y = (y^2)^5y$. The whole process takes eight multiplications, while the binary method would have required nine. The factor method is better than the binary method on the average, but there are cases ($n = 33$ is the smallest example) where the binary method excels.

The binary method can be generalized to an *m-ary method* as follows: Let $n = d_0m^t + d_1m^{t-1} + \cdots + d_t$, where $0 \le d_j < m$ for $0 \le j \le t$. The computation begins by forming x, x^2, x^3, ..., x^{m-1}. (Actually, only those powers x^{d_j} such that d_j appears in the representation of n are needed, and this observation often saves some of the work.) Then raise x^{d_0} to the mth power and multiply by x^{d_1}; we have computed $y_1 = x^{d_0m+d_1}$. Next, raise y_1 to the mth power and multiply by x^{d_2}, obtaining $y_2 = x^{d_0m^2+d_1m+d_2}$. The process continues in this way until $y_t = x^n$ has been computed. Whenever $d_j = 0$, it is, of course, unnecessary to multiply by x^{d_j}. Note that this method reduces to the left-to-right binary method discussed earlier, when $m = 2$; there is also a less obvious right-to-left m-ary method that takes more memory but only a few more steps (see exercise 9). If m is a small prime, the m-ary method will be particularly efficient for calculating powers of one polynomial modulo another, when the coefficients are treated modulo m (see Eq. 4.6.2–5).

A systematic method that gives the minimum number of multiplications for all of the relatively small values of n (in particular, for most n that occur in practical applications) is indicated in Fig. 13. To calculate x^n, find n in this tree, and the path from the root to n indicates the sequence of exponents that occur in an efficient evaluation of x^n. The rule for generating this "power tree" appears in exercise 5. Computer tests have shown that the power tree gives optimum results for all of the n listed in the figure. But for large enough values of n the power tree method is not always an optimum procedure; the smallest examples are $n = 77$, 154, 233. The first case for which the power tree is superior to both the binary method and the factor method is $n = 23$. The first case for which the factor method beats the power tree method is $n = 19879 = 103 \cdot 193$; such cases are quite rare. (For $n \le 100,000$ the power tree method is better than the factor method 88,803 times; it ties 11,191 times; and it loses only 6 times.)

Addition chains. The most economical way to compute x^n by multiplication is a mathematical problem with an interesting history. We shall now examine it in detail, not only because it is interesting in its own right, but because it is an excellent example of the theoretical questions that arise in the study of "optimum methods of computation."

Although we are concerned with multiplication of powers of x, the problem can easily be reduced to addition, since the exponents are additive. This leads us to the following abstract formulation: An *addition chain for n* is a sequence of integers

$$1 = a_0, \quad a_1, \quad a_2, \quad \ldots, \quad a_r = n \tag{1}$$

with the property that

$$a_i = a_j + a_k, \qquad \text{for some } k \le j < i, \tag{2}$$

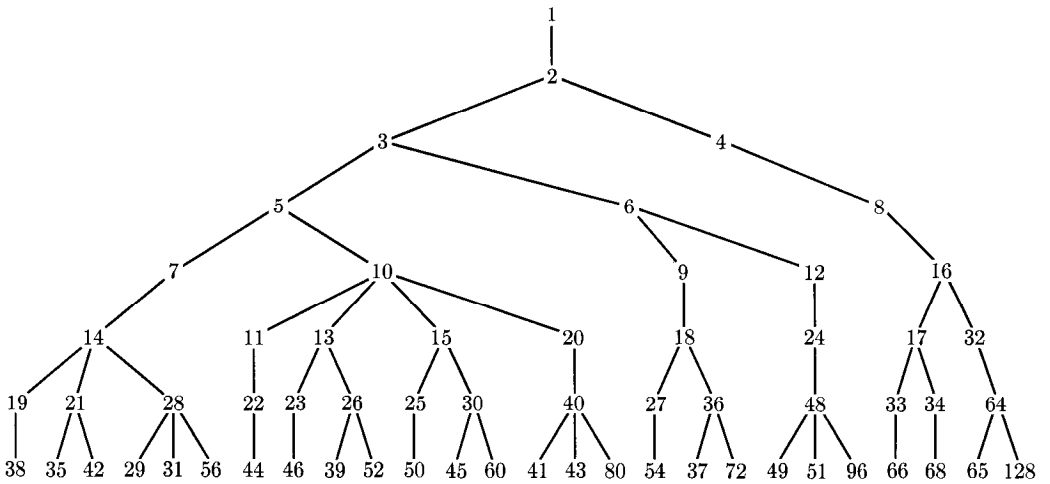


Fig. 13. The "power tree."

for all $i = 1, 2, \ldots, r$. One way of looking at this definition is to consider a simple computer that has an accumulator and is capable of the three operations LDA, STA, and ADD; the machine begins with the number 1 in its accumulator, and it proceeds to compute the number n by adding together previous results. Note that a_1 must equal 2, and a_2 is either 2, 3, or 4.

The shortest length, r, for which there exists an addition chain for n is denoted by $l(n)$. Our goal in the remainder of this section is to discover as much as we can about this function $l(n)$. The values of $l(n)$ for small n are displayed in tree form in Fig. 14, which shows how to calculate x^n with the fewest possible multiplications for all $n \leq 100$.

The problem of determining $l(n)$ was apparently first raised by H. Dellac in 1894, and a partial solution by E. de Jonquières mentioned the factor method [cf. *l'Intermédiaire des Mathématiciens* **1** (1894), 20, 162–164]. In his solution, de Jonquières listed what he felt were the values of $l(p)$ for all prime numbers $p < 200$, but his table entries for $p = 107, 149, 163, 179$ were one too high.

The factor method tells us immediately that

$$l(mn) \leq l(m) + l(n), \tag{3}$$

since we can take the chains $1, a_1, \ldots, a_r = m$ and $1, b_1, \ldots, b_s = n$ and form the chain $1, a_1, \ldots, a_r, a_r b_1, \ldots, a_r b_s = mn$.

We can also recast the m-ary method into addition-chain terminology. Consider the case $m = 2^k$, and write $n = d_0 m^t + d_1 m^{t-1} + \cdots + d_t$ in the m-ary number system; the corresponding addition chain takes the form

$$\begin{aligned} &1, 2, 3, \ldots, m-2, m-1, \\ &\quad 2d_0, 4d_0, \ldots, md_0, md_0 + d_1, \\ &\qquad 2(md_0+d_1), 4(md_0+d_1), \ldots, m(md_0+d_1), m^2 d_0 + md_1 + d_2, \\ &\qquad\quad \ldots, \qquad m^t d_0 + m^{t-1} d_1 + \cdots + d_t. \end{aligned} \tag{4}$$

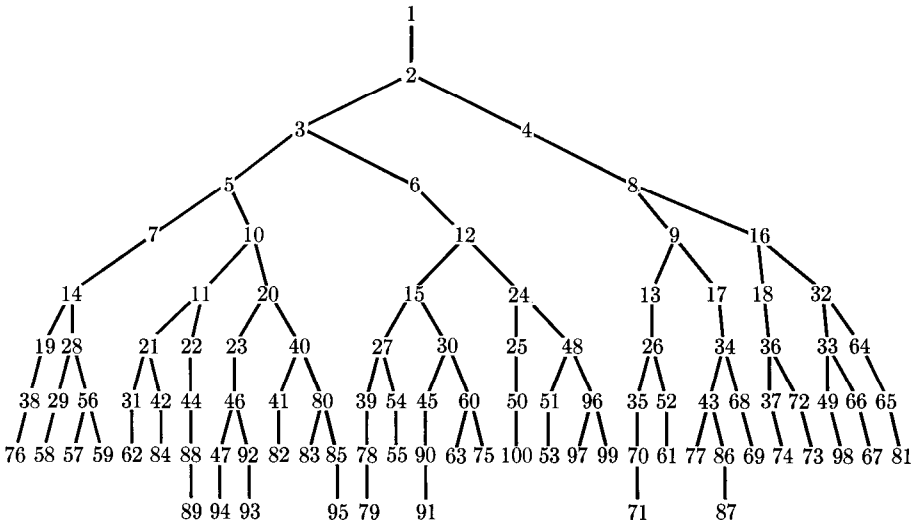


Fig. 14. A tree that minimizes the number of multiplications, for $n \leq 100$.

The length of this chain is $m - 2 + (k+1)t$; and it can often be reduced by deleting certain elements of the first row that do not occur among the coefficients d_j, plus elements among $2d_0$, $4d_0$, ... that already appear in the first row. Whenever digit d_j is zero, the step at the right end of the corresponding line may, of course, be dropped. Furthermore, as E. C. Thurber has observed [*Duke Math. J.* **40** (1973), 907–913], we can omit all the even numbers (except 2) in the first row, if we bring values of the form $d_j/2^e$ into the computation e steps earlier.

The simplest case of the m-ary method is the binary method ($m = 2$), when the general scheme (4) simplifies to the "S" and "X" rule mentioned at the beginning of this section: The binary addition chain for $2n$ is the binary chain for n followed by $2n$; for $2n+1$ it is the binary chain for $2n$ followed by $2n+1$. From the binary method we conclude that

$$l(2^{e_0} + 2^{e_1} + \cdots + 2^{e_t}) \leq e_0 + t, \qquad \text{if } e_0 > e_1 > \cdots > e_t \geq 0. \tag{5}$$

Let us now define two auxiliary functions for convenience in our subsequent discussion:

$$\lambda(n) = \lfloor \lg n \rfloor; \tag{6}$$

$$\nu(n) = \text{number of 1's in the binary representation of } n. \tag{7}$$

Thus $\lambda(17) = 4$, $\nu(17) = 2$; these functions may be defined by the recurrence relations

$$\lambda(1) = 0, \qquad \lambda(2n) = \lambda(2n+1) = \lambda(n) + 1; \tag{8}$$

$$\nu(1) = 1, \qquad \nu(2n) = \nu(n), \qquad \nu(2n+1) = \nu(n) + 1. \tag{9}$$

In terms of these functions, the binary addition chain for n requires exactly $\lambda(n) + \nu(n) - 1$ steps, and (5) becomes

$$l(n) \leq \lambda(n) + \nu(n) - 1. \tag{10}$$

Special classes of chains. We may assume without any loss of generality that an addition chain is "ascending,"

$$1 = a_0 < a_1 < a_2 < \cdots < a_r = n. \tag{11}$$

For if any two a's are equal, one of them may be dropped; and we can also rearrange the sequence (1) into ascending order and remove terms $> n$ without destroying the addition chain property (2). *From now on we shall consider only ascending chains,* without explicitly mentioning this assumption.

It is convenient at this point to define a few special terms relating to addition chains. By definition we have, for $1 \leq i \leq r$,

$$a_i = a_j + a_k \tag{12}$$

for some j and k, $0 \leq k \leq j < i$. Let us say that step i of (11) is a *doubling*, if $j = k = i - 1$; then a_i has the maximum possible value $2a_{i-1}$ that can follow the ascending chain $1, a_1, \ldots, a_{i-1}$. If j (but not necessarily k) equals $i - 1$, let us say that step i is a *star step*. The importance of star steps is explained below. Finally let us say that step i is a *small step* if $\lambda(a_i) = \lambda(a_{i-1})$. Since $a_{i-1} < a_i \leq 2a_{i-1}$, the quantity $\lambda(a_i)$ is always equal to either $\lambda(a_{i-1})$ or $\lambda(a_{i-1}) + 1$; it follows that, in any chain (11), *the length r is equal to $\lambda(n)$ plus the number of small steps.*

Several elementary relations hold between these types of steps: Step 1 is always a doubling. A doubling obviously is a star step, but never a small step. A doubling must be followed by a star step. Furthermore if step i is *not* a small step, then step $i + 1$ is either a small step or a star step, or both; putting this another way, if step $i + 1$ is neither small nor star, step i must be small.

A *star chain* is an addition chain that involves only star steps. This means that each term a_i is the sum of a_{i-1} and a previous a_k; the simple "computer" discussed above after Eq. (2) makes use only of the two operations STA and ADD (not LDA) in a star chain, since each new term of the sequence utilizes the preceding result in the accumulator. Most of the addition chains we have discussed so far are star chains. The minimum length of a star chain for n is denoted by $l^*(n)$; clearly

$$l(n) \leq l^*(n). \tag{13}$$

We are now ready to derive some nontrivial facts about addition chains. First we can show that there must be fairly many doublings if r is not far from $\lambda(n)$.

Theorem A. *If the addition chain* (11) *includes* d *doublings and* $f = r - d$ *nondoublings, then*

$$n \le 2^{d-1}F_{f+3}. \tag{14}$$

Proof. By induction on $r = d+f$, we see that (14) is certainly true when $r = 1$. When $r > 1$, there are three cases: If step r is a doubling, then $\frac{1}{2}n = a_{r-1} \le 2^{d-2}F_{f+3}$; hence (14) follows. If steps r and $r-1$ are both nondoublings, then $a_{r-1} \le 2^{d-1}F_{f+2}$ and $a_{r-2} \le 2^{d-1}F_{f+1}$; hence $n = a_r \le a_{r-1} + a_{r-2} \le 2^{d-1}(F_{f+2} + F_{f+1}) = 2^{d-1}F_{f+3}$ by the definition of the Fibonacci sequence. Finally, if step r is a nondoubling but step $r-1$ is a doubling, then $a_{r-2} \le 2^{d-2}F_{f+2}$ and $n = a_r \le a_{r-1} + a_{r-2} = 3a_{r-2}$. Now $2F_{f+3} - 3F_{f+2} = F_{f+1} - F_f \ge 0$; hence $n \le 2^{d-1}F_{f+3}$ in all cases. ∎

The method of proof we have used shows that inequality (14) is "best possible" under the stated assumptions; the addition chain

$$1, 2, \ldots, 2^{d-1}, 2^{d-1}F_3, 2^{d-1}F_4, \ldots, 2^{d-1}F_{f+3} \tag{15}$$

has d doublings and f nondoublings.

Corollary. *If the addition chain* (11) *includes* f *nondoublings and* s *small steps, then*

$$s \le f \le 3.271s. \tag{16}$$

Proof. Obviously $s \le f$. We have $2^{\lambda(n)} \le n \le 2^{d-1}F_{f+3} \le 2^d\phi^f = 2^{\lambda(n)+s}(\phi/2)^f$, since $d + f = \lambda(n) + s$, and since $F_{f+3} \le 2\phi^f$ when $f \ge 0$. Hence $0 \le s \ln 2 + f \ln(\phi/2)$, and (16) follows from the fact that

$$\ln 2 / \ln(2/\phi) \approx 3.2706. \quad ∎$$

Values of $l(n)$ for special n. It is easy to show by induction that $a_i \le 2^i$, and therefore $\lg n \le r$ in any addition chain (11). Hence

$$l(n) \ge \lceil \lg n \rceil. \tag{17}$$

This lower bound, together with the upper bound (10) given by the binary method, gives us the values

$$l(2^A) = A; \tag{18}$$

$$l(2^A + 2^B) = A + 1, \qquad \text{if } A > B. \tag{19}$$

In other words, the binary method is optimum when $\nu(n) \le 2$. With some further calculation we can extend these formulas to the case $\nu(n) = 3$:

Theorem B.

$$l(2^A + 2^B + 2^C) = A + 2, \qquad \text{if } A > B > C. \tag{20}$$

Proof. We can, in fact, prove a stronger result that will be of use to us later in this section: *All addition chains with exactly one small step have one of the following six types* (where all steps indicated by "..." represent doublings):

Type 1. $1, \ldots, 2^A, 2^A + 2^B, \ldots, 2^{A+C} + 2^{B+C}$; $A > B \geq 0$, $C \geq 0$.

Type 2. $1, \ldots, 2^A, 2^A + 2^B, 2^{A+1} + 2^B, \ldots, 2^{A+C+1} + 2^{B+C}$; $A > B \geq 0$, $C \geq 0$.

Type 3. $1, \ldots, 2^A, 2^A + 2^{A-1}, 2^{A+1} + 2^{A-1}, 2^{A+2}, \ldots, 2^{A+C}$; $A > 0$, $C \geq 2$.

Type 4. $1, \ldots, 2^A, 2^A + 2^{A-1}, 2^{A+1} + 2^A, 2^{A+2}, \ldots, 2^{A+C}$; $A > 0$, $C \geq 2$.

Type 5. $1, \ldots, 2^A, 2^A + 2^{A-1}, \ldots, 2^{A+C} + 2^{A+C-1}, 2^{A+C+1} + 2^{A+C-2}, \ldots, 2^{A+C+D+1} + 2^{A+C+D-2}$; $A > 0$, $C > 0$, $D \geq 0$.

Type 6. $1, \ldots, 2^A, 2^A + 2^B, 2^{A+1}, \ldots, 2^{A+C}$; $A > B \geq 0$, $C \geq 1$.

A straightforward hand calculation shows that these six types exhaust all possibilities. (Note that, by the corollary to Theorem A, there are at most three nondoublings when there is one small step; this maximum of three is attained only in sequences of Type 3. All of the above are star chains, except Type 6 when $B < A - 1$.)

The theorem now follows from the observation that $l(2^A + 2^B + 2^C) \leq A + 2$; and $l(2^A + 2^B + 2^C)$ must be greater than $A + 1$, since none of the six possible types have $\nu(n) > 2$. ∎

(E. de Jonquières stated without proof in 1894 that $l(n) \geq \lambda(n) + 2$ when $\nu(n) > 2$. The first published demonstration of Theorem B was by A. A. Gioia, M. V. Subbarao, and M. Sugunamma in *Duke Math. J.* **29** (1962), 481–487.)

The calculation of $l(2^A + 2^B + 2^C + 2^D)$, when $A > B > C > D$, is more involved; by the binary method it is at most $A + 3$, and by the proof of Theorem B it is at least $A + 2$. The value $A + 2$ is possible, since we know that the binary method is not optimal when $n = 15$ or $n = 23$. The complete behavior when $\nu(n) = 4$ can be determined, as we shall now see.

Theorem C. *If $\nu(n) \geq 4$ then $l(n) \geq \lambda(n) + 3$, except in the following circumstances when $A > B > C > D$ and $l(2^A + 2^B + 2^C + 2^D)$ equals $A + 2$:*

Case 1. $A - B = C - D$. (Example: $n = 15$.)
Case 2. $A - B = C - D + 1$. (Example: $n = 23$.)
Case 3. $A - B = 3$, $C - D = 1$. (Example: $n = 39$.)
Case 4. $A - B = 5$, $B - C = C - D = 1$. (Example: $n = 135$.)

Proof. When $l(n) = \lambda(n) + 2$, there is an addition chain for n having just two small steps; such an addition chain starts out as one of the six types in the proof of Theorem B, followed by a small step, followed by a sequence of nonsmall steps. Let us say that n is "special" if $n = 2^A + 2^B + 2^C + 2^D$ for one of the four

cases listed in the theorem. We can obtain addition chains of the required form for each special n, as shown in exercise 13; therefore it remains for us to prove that no chain with exactly two small steps contains any elements with $\nu(a_i) \geq 4$ except when a_i is special.

Let a "counterexample chain" be an addition chain with two small steps such that $\nu(a_r) \geq 4$, but a_r is not special. If counterexample chains exist, let $1 = a_0 < a_1 < \cdots < a_r = n$ be a counterexample chain of shortest possible length. Then step r is not a small step, since none of the six types in the proof of Theorem B can be followed by a small step with $\nu(n) \geq 4$ except when n is special. Furthermore, step r is not a doubling, otherwise $a_0, \ldots, a_{r-1}$ would be a shorter counterexample chain; and step r is a star step, otherwise $a_0, \ldots, a_{r-2}, a_r$ would be a shorter counterexample chain. Thus

$$a_r = a_{r-1} + a_{r-k}, \qquad k \geq 2; \qquad \text{and } \lambda(a_r) = \lambda(a_{r-1}) + 1. \tag{21}$$

Let c be the number of carries that occur when a_{r-1} is added to a_{r-k} in the binary number system by Algorithm 4.3.1A. Using the fundamental relation

$$\nu(a_r) = \nu(a_{r-1}) + \nu(a_{r-k}) - c, \tag{22}$$

we can prove that *step $r-1$ is not a small step* (see exercise 14).

Let $m = \lambda(a_{r-1})$. Since neither r nor $r-1$ is a small step, $c \geq 2$; and $c = 2$ can hold only when $a_{r-1} \geq 2^m + 2^{m-1}$.

Now let us suppose that $r-1$ is not a star step. Then $r-2$ is a small step, and $a_0, \ldots, a_{r-3}, a_{r-1}$ is a chain with only one small step; hence $\nu(a_{r-1}) \leq 2$ and $\nu(a_{r-2}) \leq 4$. The relation (22) can now hold only if $\nu(a_r) = 4$, $\nu(a_{r-1}) = 2$, $k = 2$, $c = 2$, $\nu(a_{r-2}) = 4$. From $c = 2$ we conclude that $a_{r-1} = 2^m + 2^{m-1}$; hence $a_0, a_1, \ldots, a_{r-3} = 2^{m-1} + 2^{m-2}$ is an addition chain with only one small step, and it must be of Type 1, so a_r belongs to Case 3. Thus $r-1$ *is a star step.*

Now assume that $a_{r-1} = 2^t a_{r-k}$ for some t. If $\nu(a_{r-1}) \leq 3$, then by (22), $c = 2$, $k = 2$, and we see that a_r must belong to Case 3. On the other hand, if $\nu(a_{r-1}) = 4$ then a_{r-1} is special, and it is easy to see by considering each case that a_r also belongs to one of the four cases. (Case 4 arises, for example, when $a_{r-1} = 90$, $a_{r-k} = 45$; or $a_{r-1} = 120$, $a_{r-k} = 15$.) Therefore we may conclude that $a_{r-1} \neq 2^t a_{r-k}$ for any t.

We have proved that $a_{r-1} = a_{r-2} + a_{r-q}$ for some $q \geq 2$. If $k = 2$, then $q > 2$, and $a_0, a_1, \ldots, a_{r-2}, 2a_{r-2}, 2a_{r-2} + a_{r-q} = a_r$ is a counterexample sequence in which $k > 2$; therefore we may assume that $k > 2$.

Let us now suppose that $\lambda(a_{r-k}) = m - 1$; the case $\lambda(a_{r-k}) < m - 1$ may be ruled out by similar arguments, as shown in exercise 14. If $k = 4$, both $r-2$ and $r-3$ are small steps; hence $a_{r-4} = 2^{m-1}$, and (22) is impossible. Therefore $k = 3$; step $r-2$ is small, $\nu(a_{r-3}) = 2$, $c = 2$, $a_{r-1} \geq 2^m + 2^{m-1}$, and $\nu(a_{r-1}) = 4$. There must be at least two carries when a_{r-2} is added to $a_{r-1} - a_{r-2}$; hence $\nu(a_{r-2}) = 4$, and a_{r-2} (being special and $\geq \frac{1}{2}a_{r-1}$) has the form $2^{m-1} + 2^{m-2} + 2^{d+1} + 2^d$ for some d. Now a_{r-1} is either $2^m + 2^{m+1} + 2^{d+1} + 2^d$ or $2^m + 2^{m-1} + 2^{d+2} + 2^{d+1}$, and in both cases a_{r-3} must be $2^{m-1} + 2^{m-2}$, so a_r belongs to Case 3. ∎

E. G. Thurber [*Pacific J. Math.* **49** (1973), 229–242] has extended Theorem C to show that $l(n) \geq \lambda(n) + 4$ when $\nu(n) > 8$. It seems reasonable to conjecture that $l(n) \geq \lambda(n) + \lg \nu(n)$ in general, since A. Schönhage has come very close to proving this (see exercise 29).

Asymptotic values. Theorem C indicates that it is probably quite difficult to get exact values of $l(n)$ for large n, when $\nu(n) > 4$; however, we can determine the approximate behavior in the limit as $n \to \infty$.

Theorem D (A. Brauer, *Bull. Amer. Math. Soc.* **45** (1939), 736–739).

$$\lim_{n\to\infty} l^*(n)/\lambda(n) = \lim_{n\to\infty} l(n)/\lambda(n) = 1. \tag{23}$$

Proof. The addition chain (4) for the 2^k-ary method is a star chain if we delete the second occurrence of any element that appears twice in the chain; for if a_i is the first element among $2d_0$, $4d_0$, ... of the second line that is not present in the first line, we have $a_i \leq 2(m-1)$; hence $a_i = (m-1) + a_j$ for some a_j in the first line. By totaling up the length of the chain, we have

$$\lambda(n) \leq l(n) \leq l^*(n) < \left(1 + \frac{1}{k}\right) \lg n + 2^k \tag{24}$$

for all $k \geq 1$. The theorem follows if we choose, say, $k = \lfloor \frac{1}{2} \lg \lambda(n) \rfloor$. ∎

If we let $k = \lambda\lambda(n) - 2\lambda\lambda\lambda(n)$ in (24) for large n, where $\lambda\lambda(n)$ denotes $\lambda(\lambda(n))$, we obtain the stronger asymptotic bound

$$l(n) \leq l^*(n) \leq \lambda(n) + \lambda(n)/\lambda\lambda(n) + O\big(\lambda(n)\lambda\lambda\lambda(n)/\lambda\lambda(n)^2\big). \tag{25}$$

The second term $\lambda(n)/\lambda\lambda(n)$ is essentially the best that can be obtained from (24). A much deeper analysis of lower bounds can be carried out, to show that this term $\lambda(n)/\lambda\lambda(n)$ is, in fact, essential in (25). In order to see why this is so, let us consider the following fact:

Theorem E (Paul Erdős, *Acta Arithmetica* **6** (1960), 77–81). *Let ϵ be a positive real number. The number of addition chains* (11) *such that*

$$\lambda(n) = m, \qquad r \leq m + (1-\epsilon)m/\lambda(m) \tag{26}$$

is less than α^m, for some $\alpha < 2$, for all suitably large m. (In other words, the number of addition chains so short that (26) is satisfied is substantially less than the number of values of n such that $\lambda(n) = m$, when m is large.)

Proof. We want to estimate the number of possible addition chains, and for this purpose our first goal is to get an improvement of Theorem A that enables us to deal more satisfactorily with nondoublings.

Lemma P. *Let $\delta < \sqrt{2} - 1$ be a fixed positive real number. Call step i of an addition chain a "ministep" if it is not a doubling and if $a_i < a_j(1+\delta)^{i-j}$ for some j, where $0 \le j < i$. If the addition chain contains s small steps and t ministeps, then*

$$t \le s/(1-\theta), \qquad \text{where } (1+\delta)^2 = 2^\theta. \tag{27}$$

Proof. For each ministep i_k, $1 \le k \le t$, we have $a_{i_k} < a_{j_k}(1+\delta)^{i_k - j_k}$ for some $j_k < i_k$. Let $I_1, \ldots, I_t$ be the intervals $(j_1, i_1], \ldots, (j_t, i_t]$, where the notation $(j, i]$ stands for the set of all integers k such that $j < k \le i$. It is possible (see exercise 17) to find nonoverlapping intervals $J_1, \ldots, J_h = (j'_1, i'_1], \ldots, (j'_h, i'_h]$ such that

$$\begin{aligned} I_1 \cup \cdots \cup I_t &= J_1 \cup \cdots \cup J_h, \\ a_{i'_k} < a_{j'_k}(1+\delta)^{2(i'_k - j'_k)}, &\qquad \text{for } 1 \le k \le h. \end{aligned} \tag{28}$$

Now for all steps i outside of the intervals $J_1, \ldots, J_h$ we have $a_i \le 2a_{i-1}$; hence if we let

$$q = (i'_1 - j'_1) + \cdots + (i'_h - j'_h),$$

we have

$$2^{\lambda(n)} \le n \le 2^{r-q}(1+\delta)^{2q} = 2^{\lambda(n)+s-(1-\theta)q} \le 2^{\lambda(n)+s-(1-\theta)t}. \quad \blacksquare$$

Returning to the proof of Theorem E, let us choose $\delta = 2^{\epsilon/4} - 1$, and let us divide the r steps of each addition chain into three classes:

$$t \text{ ministeps}, \qquad u \text{ doublings}, \qquad v \text{ other steps}, \qquad t + u + v = r. \tag{29}$$

Counting another way, we have s small steps, where $s + m = r$. By the hypotheses, Theorem A, and Lemma P, we obtain the relations

$$t \le s/(1-\epsilon/2), \qquad t + v \le 3.271s, \qquad s \le (1-\epsilon)m/\lambda(m). \tag{30}$$

Given s, t, u, v satisfying these conditions, there are at most

$$\binom{r}{t+v}\binom{t+v}{v} \tag{31}$$

ways to assign the steps to the specified classes. Given such a distribution of the steps, let us consider how the non-ministeps can be selected: If step i is one of the "other" steps in (29), $a_i \ge (1+\delta)a_{i-1}$, so $a_i = a_j + a_k$, where $\delta a_{i-1} \le a_k \le a_j \le a_{i-1}$. Also $a_j \le a_i/(1+\delta)^{i-j} \le 2a_{i-1}/(1+\delta)^{i-j}$, so $\delta \le 2/(1+\delta)^{i-j}$. This gives at most β choices for j, where β is a constant that depends only on δ. There are also at most β choices for k, so the number of ways to assign j and k for each of the non-ministeps is at most

$$\beta^{2v}. \tag{32}$$

Finally, once the "j" and "k" have been selected for each of the nonministeps, there are fewer than

$$\binom{r^2}{t} \tag{33}$$

ways to choose the j and the k for the ministeps: We select t distinct pairs $(j_1, k_1), \ldots, (j_t, k_t)$ of indices in the range $0 \le k_h \le j_h < r$, in fewer than (33) ways. Then for each ministep i, in turn, we use a pair of indices (j_h, k_h) such that

a) $j_h < i$;
b) $a_{j_h} + a_{k_h}$ is as small as possible among the pairs not already used for smaller ministeps i;
c) $a_i = a_{j_h} + a_{k_h}$ satisfies the definition of ministep.

If no such pair (j_h, k_h) exists, we get no addition chain; on the other hand, any addition chain with ministeps in the designated places must be selected in one of these ways, so (33) is an upper bound on the possibilities.

Thus the total number of possible addition chains satisfying (26) is bounded by (31) times (32) times (33), summed over all relevant s, t, u, and v. The proof of Theorem E can now be completed by means of a rather standard estimation of these functions (exercise 18). ∎

Corollary. *The value of $l(n)$ is asymptotically $\lambda(n) + \lambda(n)/\lambda\lambda(n)$, for almost all n. More precisely, there is a function $f(n)$ such that $f(n) \to 0$ as $n \to \infty$, and*

$$\Pr\big(\,|l(n) - \lambda(n) - \lambda(n)/\lambda\lambda(n)| \ge f(n)\lambda(n)/\lambda\lambda(n)\,\big) = 0. \tag{34}$$

(See Section 3.5 for the definition of this probability "Pr".)

Proof. The upper bound (25) shows that (34) holds without the absolute value signs. The lower bound comes from Theorem E, if we let $f(n)$ decrease to zero slowly enough so that, when $f(n) \le \epsilon$, the value N is so large that at most ϵN values $n \le N$ have $l(n) \le \lambda(n) + (1 - \epsilon)\lambda(n)/\lambda\lambda(n)$. ∎

Star chains. Optimistic people find it reasonable to suppose that $l(n) = l^*(n)$; given an addition chain of minimal length $l(n)$, it appears hard to believe that we cannot find one of the same length that satisfies the (apparently mild) star condition. But in 1958 Walter Hansen proved the remarkable theorem that, for certain large values of n, the value of $l(n)$ is definitely less than $l^*(n)$, and he also proved several related theorems that we shall now investigate.

Hansen's theorems begin with an investigation of the detailed structure of a star chain. This structure is given in terms of other sequences and sets constructed from the given chain. Let $n = 2^{e_0} + 2^{e_1} + \cdots + 2^{e_t}$, $e_0 > e_1 > \cdots > e_t \ge 0$, and let $1 = a_0 < a_1 < \cdots < a_r = n$ be a star chain for n. If there are d doublings in this chain, we define the auxiliary sequence

$$0 = d_0 \le d_1 \le d_2 \le \cdots \le d_r = d, \tag{35}$$

where d_i is the number of doublings among steps 1, 2, ..., i. We also define a sequence of "multisets" S_0, S_1, ..., S_r, which keep track of the powers of 2 present in the chain. (A *multiset* is a mathematical entity that is like a set, but it is allowed to contain repeated elements; an object may be an element of a multiset several times, and its multiplicity of occurrences is relevant. See exercise 19 for familiar examples of multisets.) The multisets S_i are defined by the rules

a) $S_0 = \{0\}$;

b) If $a_{i+1} = 2a_i$, then $S_{i+1} = S_i + 1 = \{ x + 1 \mid x \in S_i \}$;

c) If $a_{i+1} = a_i + a_k$, $k < i$, then $S_{i+1} = S_i \uplus S_k$.

(The symbol $\uplus$ means that the multisets are combined, adding the multiplicities.) From this definition it follows that

$$a_i = \sum_{x \in S_i} 2^x, \tag{36}$$

where the terms in this sum are not necessarily distinct. In particular,

$$n = 2^{e_0} + 2^{e_1} + \cdots + 2^{e_t} = \sum_{x \in S_r} 2^x. \tag{37}$$

The number of elements in the latter sum is at most 2^f, where $f = r - d$ is the number of nondoublings.

Since n has two different binary representations in (37), we can partition the multiset S_r into multisets M_0, M_1, ..., M_t such that

$$2^{e_j} = \sum_{x \in M_j} 2^x, \qquad 0 \le j \le t. \tag{38}$$

This can be done by arranging the elements of S_r into nondecreasing order $x_1 \le x_2 \le \cdots$ and taking $M_t = \{x_1, x_2, \ldots, x_k\}$, where $2^{x_1} + \cdots + 2^{x_k} = 2^{e_t}$. This must be possible, since e_t is the smallest of the e's. Similarly, $M_{t-1} = \{x_{k+1}, x_{k+2}, \ldots, x_{k'}\}$, and so on; the process is easily visualized in binary notation.

Let M_j contain m_j elements (counting multiplicities); then $m_j \le 2^f - t$, since S_r has at most 2^f elements and it has been partitioned into $t+1$ nonempty multisets. By Eq. (38), we can see that

$$e_j \ge x > e_j - m_j, \qquad \text{for all } x \in M_j. \tag{39}$$

Our examination of the star chain's structure is completed by forming the multisets M_{ij} that record the ancestral history of M_j. The multiset S_i is partitioned into $t + 1$ multisets as follows:

a) $M_{rj} = M_j$;

b) If $a_{i+1} = 2a_i$, then $M_{ij} = M_{(i+1)j} - 1 = \{ x - 1 \mid x \in M_{(i+1)j} \}$;

c) If $a_{i+1} = a_i + a_k$, $k < i$, then (since $S_{i+1} = S_i \uplus S_k$) we let $M_{ij} = M_{(i+1)j}$ minus S_k, that is, we remove the elements of S_k from $M_{(i+1)j}$. If some element of S_k appears in two or more different multisets $M_{(i+1)j}$, we remove it from the set with the largest possible value of j; this rule uniquely defines M_{ij} for each j, when i is fixed.

From this definition it follows that

$$e_j + d_i - d \geq x > e_j + d_i - d - m_j, \qquad \text{for all } x \in M_{ij}. \tag{40}$$

As an example of this detailed construction, let us consider the star chain 1, 2, 3, 5, 10, 20, 23, for which $t = 3$, $r = 6$, $d = 3$, $f = 3$. We obtain the following array of multisets:

	S_0	S_1	S_2	S_3	S_4	S_5	S_6		
$(d_0, d_1, \ldots, d_6)$:	0	1	1	1	2	3	3		
$(a_0, a_1, \ldots, a_6)$:	1	2	3	5	10	20	23		
$(M_{03}, M_{13}, \ldots, M_{63})$:							0	M_3	$e_3 = 0$, $m_3 = 1$
$(M_{02}, M_{12}, \ldots, M_{62})$:							1	M_2	$e_2 = 1$, $m_2 = 1$
$(M_{01}, M_{11}, \ldots, M_{61})$:			0	0	1	2	2	M_1	$e_1 = 2$, $m_1 = 1$
$(M_{00}, M_{10}, \ldots, M_{60})$:	0	1	1	1	2	3	3	M_0	$e_0 = 4$, $m_0 = 2$
				1	2	3	3		

Thus $M_{40} = \{2, 2\}$, etc. From the construction we can see that d_i is the largest element of S_i; hence

$$d_i \in M_{i0}. \tag{41}$$

The most important part of this structure comes from Eq. (40); one of its immediate consequences is

Lemma K. *If M_{ij} and M_{uv} both contain a common integer x, then*

$$-m_v < (e_j - e_v) - (d_u - d_i) < m_j. \quad \blacksquare \tag{42}$$

Although Lemma K may not look extremely powerful, it says (when M_{ij} contains an element in common with M_{uv} and when m_j, m_v are reasonably small) that the number of doublings between steps u and i is approximately equal to the difference between the exponents e_v and e_j. This imposes a certain amount of regularity on the addition chain; and it suggests that we might be able to prove a result analogous to Theorem B above, that $l^*(n) = e_0 + t$, provided that the e_j are far enough apart. The next theorem shows how this can be done.

Theorem H (W. Hansen, *J. für die reine und angew. Math.* **202** (1959), 129–136). *Let $n = 2^{e_0} + 2^{e_1} + \cdots + 2^{e_t}$, $e_0 > e_1 > \cdots > e_t \geq 0$. If*

$$e_0 > 2e_1 + 2.271(t-1) \qquad \text{and} \qquad e_{i-1} \geq e_i + 2m \quad \text{for } 1 \leq i \leq t, \tag{43}$$

where $m = 2^{\lfloor 3.271(t-1) \rfloor} - t$, then $l^*(n) = e_0 + t$.

Proof. We may assume that $t > 2$, since the result of the theorem is true without restriction on the e's when $t \leq 2$. Suppose that we have a star chain $1 = a_0 < a_1 < \cdots < a_r = n$ for n with $r \leq e_0 + t - 1$. Let the integers d, f, $d_0, \ldots, d_r$,

and the multisets M_{ij} and S_i, reflect the structure of this chain, as defined above. By the corollary to Theorem A, we know that $f \le \lfloor 3.271(t-1) \rfloor$; therefore the value of m is a bona fide upper bound for the number m_j of elements in each multiset M_j.

In the summation

$$a_i = \Big(\sum_{x\in M_{i0}} 2^x\Big) + \Big(\sum_{x\in M_{i1}} 2^x\Big) + \cdots + \Big(\sum_{x\in M_{it}} 2^x\Big),$$

no carries propagate from the term corresponding to M_{ij} to the term corresponding to $M_{i(j-1)}$, if we think of this sum as being carried out in the binary number system, since the e's are so far apart. (Cf. (40).) In particular, the sum of all the terms for $j \neq 0$ will not carry up to affect the terms for $j = 0$, so we must have

$$a_i \ge \sum_{x\in M_{i0}} 2^x \ge 2^{\lambda(a_i)}, \qquad 0 \le i \le r. \tag{44}$$

In order to prove Theorem H, we would like to show that in some sense the t extra powers of n must be put in "one at a time," so we want to find a way to tell at which step each of these terms essentially enters the addition chain.

Let j be a number between 1 and t. Since M_{0j} is empty and $M_{rj} = M_j$ is nonempty, we can find the *first* step i for which M_{ij} is not empty.

From the way in which the M_{ij} are defined, we know that step i is a nondoubling: $a_i = a_{i-1} + a_u$ for some $u < i - 1$. We also know that all the elements of M_{ij} are elements of S_u. We will prove that a_u must be relatively small compared to a_i.

Let x_j be an element of M_{ij}. Then since $x_j \in S_u$, there is some v for which $x_j \in M_{uv}$. It follows that

$$d_i - d_u > m, \tag{45}$$

i.e., at least $m + 1$ doublings occur between steps u and i. For if $d_i - d_u \le m$, Lemma K tells us that $|e_j - e_v| < 2m$; hence $v = j$. But this is impossible, because M_{uj} is empty by our choice of step i.

All elements of S_u are less than or equal to $e_1 + d_i - d$. For if $x \in S_u \subseteq S_i$ and $x > e_1 + d_i - d$, then $x \in M_{u0}$ and $x \in M_{i0}$ by (40); so Lemma K implies that $|d_i - d_u| < m$, contradicting (45). In fact, this argument proves that M_{i0} has no elements in common with S_u, so $M_{(i-1)0} = M_{i0}$. From (44) we have $a_{i-1} \ge 2^{\lambda(a_i)}$, and therefore *step i is a small step.*

We can now deduce what is probably the key fact in this entire proof: *All elements of S_u are in M_{u0}.* For if not, let x be an element of S_u with $x \notin M_{u0}$. Since $x \ge 0$, (40) implies that $e_1 \ge d - d_u$, hence

$$e_0 = f + d - s \le 2.271s + d \le 2.271(t-1) + e_1 + d_u.$$

By hypothesis (43), this implies $d_u > e_1$. But $d_u \in S_u$ by (41), and it cannot be in M_{i0}, hence $d_u \le e_1 + d_i - d \le e_1$, a contradiction.

Going back to our element x_j in M_{ij}, we have $x_j \in M_{uv}$; and we have proved that $v = 0$. Therefore, by equation (40) again,

$$e_0 + d_u - d \geq x_j > e_0 + d_u - d. \tag{46}$$

For all $j = 1, 2, \ldots, t$ we have determined a number x_j satisfying (46), and a small step i at which the term 2^{e_j} may be said to have entered into the addition chain. If $j \neq j'$, the step i at which this occurs cannot be the same for both j and j'; for (46) would tell us that $|x_j - x_{j'}| < m$, while elements of M_{ij} and $M_{ij'}$ must differ by more than m, since e_j and $e_{j'}$ are so far apart. We are forced to conclude that the chain contains at least t small steps; but this is a contradiction. ∎

Theorem F (W. Hansen).

$$l(2^A + xy) \leq A + \nu(x) + \nu(y) - 1, \qquad \text{if } \lambda(x) + \lambda(y) \leq A. \tag{47}$$

Proof. An addition chain (which is *not* a star chain in general) may be constructed by combining the binary and factor methods. Let $x = 2^{x_1} + \cdots + 2^{x_u}$ and $y = 2^{y_1} + \cdots + 2^{y_v}$, where $x_1 > \cdots > x_u \geq 0$ and $y_1 > \cdots > y_v \geq 0$.

The first steps of the chain form successive powers of 2, until 2^{A-y_1} is reached; in between these steps, the additional values $2^{x_u-1} + 2^{x_u}$, $2^{x_u-2} + 2^{x_u-1} + 2^{x_u}$, ..., and x are inserted in the appropriate places. After a chain up to $2^{A-y_i} + x(2^{y_1-y_i} + \cdots + 2^{y_{i-1}-y_i})$ has been formed, we continue by adding x and doubling the resulting sum $y_i - y_{i+1}$ times; this yields

$$2^{A-y_{i+1}} + x(2^{y_1-y_{i+1}} + \cdots + 2^{y_i-y_{i+1}}).$$

If this construction is done for $i = 1, 2, \ldots, v$, assuming for convenience that $y_{v+1} = 0$, we have an addition chain for $2^A + xy$ as desired. ∎

Theorem F enables us to find values of n for which $l(n) < l^*(n)$, since Theorem H gives an explicit value of $l^*(n)$ in certain cases. For example, let $x = 2^{1016} + 1$, $y = 2^{2032} + 1$, and let

$$n = 2^{6103} + xy = 2^{6103} + 2^{3048} + 2^{2032} + 2^{1016} + 1.$$

According to Theorem F, we have $l(n) \leq 6106$. But Theorem H also applies, with $m = 508$, and this proves that $l^*(n) = 6107$.

Extensive computer calculations have shown that $n = 12509$ is the smallest value with $l(n) < l^*(n)$. No star chain for this value of n is as short as the sequence 1, 2, 4, 8, 16, 17, 32, 64, 128, 256, 512, 1024, 1041, 2082, 4164, 8328, 8345, 12509. The brute-force methods in the proof of Theorem C could be extended by computer program to determine all n such that $l(n) = \lambda(n) + 3$; this approach would also characterize all n with $\nu(n) = 5$ and $l(n) \neq l^*(n)$. (The smallest such n is $16537 = 2^{14} + 9 \cdot 17$.)

Some conjectures. Although it was reasonable to guess at first glance that $l(n) = l^*(n)$, we have now seen that this is false. Another plausible conjecture [first made by A. Goulard, and supposedly "proved" by E. de Jonquières in *l'Interm. des Math.* **2** (1895), 125–126] is that $l(2n) = l(n) + 1$; a doubling step is so efficient, it seems unlikely that there could be any shorter chain for $2n$ than to add a doubling step to the shortest chain for n. But computer calculations show that this conjecture also fails, since $l(191) = l(382) = 11$. (A star chain of length 11 for 382 is not hard to find; e.g., 1, 2, 4, 5, 9, 14, 23, 46, 92, 184, 198, 382. The number 191 is minimal such that $l(n) = 11$, and it seems to be very difficult to prove by hand that $l(191) > 10$; the computer's proof of this fact, using a backtrack method that will be sketched in Section 7.2.2, involved a detailed examination of 948 cases.) The smallest four values of n such that $l(2n) = l(n)$ are $n = 191, 701, 743, 1111$; E. G. Thurber proved in *Pacific J. Math.* **49** (1973), 229–242, that the third of these is a member of an infinite family of such n, namely $23 \cdot 2^k + 7$ for all $k \geq 5$. It seems reasonable to conjecture that $l(2n) \geq l(n)$, but even this may be false. Kevin R. Hebb has shown that $l(n) - l(mn)$ can get arbitrarily large, for all fixed integers m not a power of 2 [*Notices Amer. Math. Soc.* **21** (1974), A–294]. The smallest case in which $l(mn) < l(n)$ is $l\big((2^{13} + 1)/3\big) = 15$.

Let $c(r)$ be the smallest value of n such that $l(n) = r$. It seems to be most difficult to compute $l(n)$ for these values of n. We have the following table:

r	$c(r)$	r	$c(r)$	r	$c(r)$
1	2	7	29	13	607
2	3	8	47	14	1087
3	5	9	71	15	1903
4	7	10	127	16	3583
5	11	11	191	17	6271
6	19	12	379	18	11231

For $r \leq 11$, the value of $c(r)$ is approximately equal to $c(r-1) + c(r-2)$, and this fact led to speculation by several people that $c(r)$ grows like the function ϕ^r; but the result of Theorem D (with $n = c(r)$) implies that $r/\lg c(r) \to 1$ as $n \to \infty$. [See E. G. Thurber, *Duke Math. J.* **40** (1973), 907–913, for more detailed information about the growth of $c(r)$.] Several people had conjectured at one time that $c(r)$ would always be a prime number; but $c(15) = 11 \cdot 173$ and $c(18) = 11 \cdot 1021$. Perhaps no conjecture about addition chains is safe!

Tabulated values of $l(n)$ show that this function is surprisingly smooth; for example, $l(n) = 13$ for all n in the range $1125 \leq n \leq 1148$. The computer calculations show that a table of $l(n)$ may be prepared for all $n \leq 1000$ by using the formula

$$l(n) = \min(l(n-1) + 1, l) - \delta, \tag{48}$$

where $l = \infty$ if n is prime, otherwise $l = l(p) + l(n/p)$ if p is the smallest prime dividing n; and $\delta = 1$ for n in Table 1, $\delta = 0$ otherwise.

Table 1

VALUES OF n FOR SPECIAL ADDITION CHAINS

23	163	229	319	371	413	453	553	599	645	707	741	813	849	903
43	165	233	323	373	419	455	557	611	659	709	749	825	863	905
59	179	281	347	377	421	457	561	619	667	711	759	835	869	923
77	203	283	349	381	423	479	569	623	669	713	779	837	887	941
83	211	293	355	382	429	503	571	631	677	715	787	839	893	947
107	213	311	359	395	437	509	573	637	683	717	803	841	899	955
149	227	317	367	403	451	551	581	643	691	739	809	845	901	983

Let $d(r)$ be the number of solutions n to the equation $l(n) = r$. The following table lists the first few values of this function:

r	$d(r)$	r	$d(r)$	r	$d(r)$
1	1	6	15	11	246
2	2	7	26	12	432
3	3	8	44	13	772
4	5	9	78	14	1382
5	9	10	136	15	2481

Surely $d(r)$ must be an increasing function of r, but there is no evident way to prove this seemingly simple assertion, much less to determine the asymptotic growth of $d(r)$ for large r.

The most famous problem about addition chains that is still outstanding is the "Scholz-Brauer conjecture," which states that

$$l(2^n - 1) \leq n - 1 + l(n). \tag{49}$$

Computer calculations show, in fact, that equality holds in (49) for $1 \leq n \leq 14$; and hand calculations by E. G. Thurber [*Discrete Math.* **16** (1976), 279–289] have shown that equality holds also for $n = 15, 16, 17, 18, 20, 24, 32$. Much of the research on addition chains has been devoted to attempts to prove (49); addition chains for the number $2^n - 1$, which has so many ones in its binary representation, are of special interest, since this is the worst case for the binary method. Arnold Scholz coined the name "addition chain" (in German) and posed (49) as a problem in 1937 [*Jahresbericht der deutschen Mathematiker-Vereinigung*, class II, **47** (1937), 41–42]; Alfred Brauer proved in 1939 that

$$l^*(2^n - 1) \leq n - 1 + l^*(n). \tag{50}$$

Hansen's theorems show that $l(n)$ can be less than $l^*(n)$, so more work is definitely necessary in order to prove or disprove (49). As a step in this direction, Hansen has defined the concept of an *l^0-chain*, which lies "between" l-chains and l^*-chains. In an l^0-chain, certain of the elements are underlined; the condition is that $a_i = a_j + a_k$, where a_j is the largest underlined element less than a_i.

As an example of an l^0-chain (certainly not a minimum one), consider

$$\underline{1}, \underline{2}, \underline{4}, 5, \underline{8}, 10, 12, \underline{18}; \tag{51}$$

it is easy to verify that the difference between each element and the previous underlined element is in the chain. We let $l^0(n)$ denote the minimum length of an l^0-chain for n. Clearly $l(n) \leq l^0(n) \leq l^*(n)$.

The chain constructed in Theorem F is an l^0-chain (see exercise 22); hence we have $l^0(n) < l^*(n)$ for certain n. It is not known whether or not $l(n) = l^0(n)$ in all cases; if this equation were true, the Scholz-Brauer conjecture would be settled, because of another theorem due to Hansen:

Theorem G. $l^0(2^n - 1) \leq n - 1 + l^0(n)$.

Proof. Let $1 = a_0, a_1, \ldots, a_r = n$ be an l^0-chain of minimum length for n, and let $1 = b_0, b_1, \ldots, b_t = n$ be the subsequence of underlined elements. (We may assume that n is underlined.) Then we can get an l^0-chain for $2^n - 1$ as follows:

a) Include the $l^0(n)+1$ numbers $2^{a_i} - 1$, for $0 \leq i \leq r$, underlined if and only if a_i is underlined.

b) Include the numbers $2^i(2^{b_j} - 1)$, for $0 \leq j < t$ and for $0 < i \leq b_{j+1} - b_j$, all underlined. (This is a total of $b_1 - b_0 + \cdots + b_t - b_{t-1} = n - 1$ numbers.)

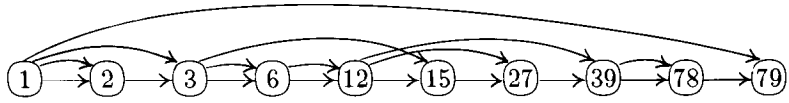
c) Sort the numbers from (a) and (b) into ascending order.

We may easily verify that this gives an l^0-chain: The numbers of (b) are all equal to twice some other element of (a) or (b); furthermore, this element is the preceding underlined element. If $a_i = b_j + a_k$, where b_j is the largest underlined element less than a_i, then $a_k = a_j - b_j \leq b_{j+1} - b_j$, so $2^{a_k}(2^{b_j} - 1) = 2^{a_i} - 2^{a_k}$ appears underlined in the chain, just preceding $2^{a_i} - 1$. Since $2^{a_i} - 1$ is equal to $(2^{a_i} - 2^{a_k}) + (2^{a_k} - 1)$, where both of these values appear in the chain, we have an addition chain with the l^0 property. ∎

The chain corresponding to (51), constructed in the proof of Theorem G, is

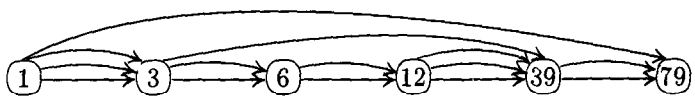
$$\underline{1}, \underline{2}, \underline{3}, \underline{6}, \underline{12}, \underline{15}, \underline{30}, 31, \underline{60}, \underline{120}, \underline{240}, \underline{255}, \underline{510}, \underline{1020}, 1023, \underline{2040},$$
$$\underline{4080}, 4095, \underline{8160}, \underline{16320}, \underline{32640}, \underline{65280}, \underline{130560}, \underline{261120}, \underline{262143}.$$

Graphical representation. An addition chain (1) corresponds in a natural way to a directed graph, where the vertices are labeled a_i for $0 \leq i \leq r$, and where we draw arcs from a_j to a_i and from a_k to a_i as a representation of each step $a_i = a_j + a_k$ in (2). For example, the addition chain 1, 2, 3, 6, 12, 15, 27, 39, 78, 79 that appears in Fig. 14 corresponds to the directed graph



If $a_i = a_j + a_k$ for more than one pair of indices (j, k), we choose a definite j and k for purposes of this construction.

In general, all but the first vertex of such a directed graph will be at the head of exactly two arcs; however, this is not really an important property of the graph, because it conceals the fact that many different addition chains can be essentially equivalent. If a vertex has out-degree 1 (i.e., only one arc leading away), it is used in only one later step, hence the later step is essentially a sum of three inputs $a_j + a_k + a_m$ that might be computed either as $(a_j + a_k) + a_m$ or as $a_j + (a_k + a_m)$ or as $a_k + (a_j + a_m)$. These three choices are immaterial, but the addition-chain conventions force us to distinguish between them. We can avoid such redundancy by deleting any vertex whose out-degree is 1 and attaching the arcs from its predecessors to its successor. For example, the above graph would become



(52)

We can also delete any vertex whose out-degree is 0, except of course the final vertex a_r, since such a vertex corresponds to a useless step in the addition chain.

In this way every addition chain leads to a reduced directed graph that contains one "source" vertex (labeled 1) and one "sink" vertex (labeled n); every vertex but the source has in-degree ≥ 2 and every vertex but the sink has out-degree ≥ 2. Conversely, any such directed graph without oriented cycles corresponds to at least one addition chain, since we can topologically sort the vertices and write down $d - 1$ addition steps for each vertex of in-degree $d > 0$. The length of the addition chain, exclusive of useless steps, can be reconstructed by looking at the reduced graph; it is

$$(\text{number of arcs}) - (\text{number of vertices}) + 1, \tag{53}$$

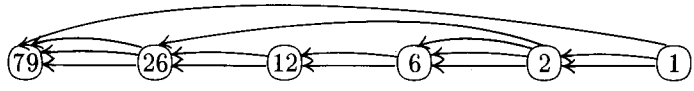
since deletion of a vertex of out-degree 1 also deletes one arc.

We say that two addition chains are *equivalent* if they have the same reduced directed graph. For example, the addition chain 1, 2, 3, 6, 12, 15, 24, 39, 40, 79 is equivalent to the chain we began with, since it also leads to (52). This example shows that a non-star chain can be equivalent to a star chain. An addition chain is equivalent to a star chain if and only if its reduced directed graph can be topologically sorted in only one way.

An important property of this graph representation has been pointed out by N. Pippenger: The label of each vertex is exactly equal to the number of oriented paths from the source to that vertex. Thus, the problem of finding an optimal addition chain for n is equivalent to minimizing the quantity (53) over all directed graphs that have one source vertex and one sink vertex and exactly n oriented paths from the source to the sink.

This characterization has a surprising corollary, because of the symmetry of the directed graph. If we reverse the direction of all the arcs, the source and the

sink exchange rôles, and we obtain another directed graph corresponding to a set of addition chains for the same n; these addition chains have the same length (53) as the chain we started with. For example, if we make the arrows in (52) run from right to left, and if we relabel the vertices according to the number of paths from the right-hand vertex, we get



. (54)

One of the star chains corresponding to this reduced directed graph is

$$1, 2, 4, 6, 12, 24, 26, 52, 78, 79;$$

we may call this a *dual* of the original addition chain.

Exercises 39 and 40 discuss important consequences of this graphical representation and the duality principle.

EXERCISES

1. [*15*] What is the value of Z when Algorithm A terminates?

2. [*24*] Write a MIX program for Algorithm A, to calculate $x^n \bmod w$ given integers n and x, where w is the word size. Assume that MIX has the binary operations SRB, JAE, etc., that are described in Section 4.5.2. Write another program that computes $x^n \bmod w$ in a serial manner (multiplying repeatedly by x), and compare the running times of these programs.

▶ **3.** [*22*] How is x^{975} calculated by (a) the binary method? (b) the ternary method? (c) the quaternary method? (d) the factor method?

4. [*M20*] Find a number n for which the octal (2^3-ary) method gives ten fewer multiplications than the binary method.

▶ **5.** [*24*] Figure 13 shows the first eight levels of the "power tree." The $(k+1)$-st level of this tree is defined as follows, assuming that the first k levels have been constructed: Take each node n of the kth level, from left to right in turn, and attach below it the nodes

$$n+1,\ n+a_1,\ n+a_2,\ \ldots,\ n+a_{k-1} = 2n$$

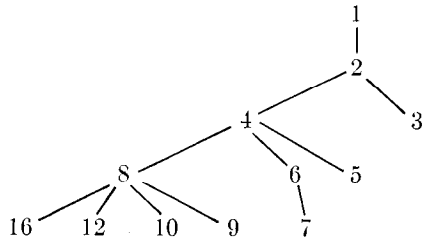
(in this order), where $1, a_1, a_2, \ldots, a_{k-1}$ is the path from the root of the tree to n; but discard any node that duplicates a number that has already appeared in the tree.

Design an efficient algorithm that constructs the first $r+1$ levels of the power tree. [*Hint:* Make use of two sets of variables LINKU[j], LINKR[j] for $0 \le j \le 2^r$; these point upwards and to the right, respectively, if j is a number in the tree.]

6. [*M26*] If a slight change is made to the definition of the power tree that is given in exercise 5, so that the nodes below n are attached in *decreasing* order

$$n + a_{k-1}, \ \ldots, \ n + a_2, \ n + a_1, \ n + 1$$

instead of increasing order, we get a tree whose first five levels are



Show that this tree gives a method of computing x^n that requires exactly as many multiplications as the binary method; therefore it is not as good as the power tree, although it has been constructed in almost the same way.

7. [*M21*] Prove that there are infinitely many values of n

a) for which the factor method is better than the binary method;
b) for which the binary method is better than the factor method;
c) for which the power tree method is better than both the binary and factor methods.

(Here the "better" method shows how to compute x^n using fewer multiplications.)

8. [*M21*] Prove that the power tree (exercise 5) never gives more multiplications for the computation of x^n than the binary method.

9. [*M21*] Design an exponentiation procedure that is analogous to Algorithm A, but based on a general radix $m \geq 2$. Your method should perform approximately $\lg n + \nu + 2m$ multiplications, where ν is the number of nonzero digits in the m-ary representation of n.

10. [*10*] Figure 14 shows a tree that indicates one way to compute x^n with the fewest possible multiplications, for all $n \leq 100$. How can this tree be conveniently represented within a computer, in just 100 memory locations?

▶ **11.** [*M26*] The tree of Fig. 14 depicts addition chains $a_0, a_1, \ldots, a_r$ having $l(a_i) = i$ for all i in the chain. Find all addition chains for n that have this property, when $n = 43$ and when $n = 77$. Show that any tree such as Fig. 14 must include either the path 1, 2, 4, 8, 9, 17, 34, 43, 77 or the path 1, 2, 4, 8, 9, 17, 34, 68, 77.

12. [*M10*] Is it possible to extend the tree shown in Fig. 14 to an infinite tree that yields a minimum-multiplication rule for computing x^n, for all positive integers n?

13. [*M21*] Find a star chain of length $A + 2$ for each of the four cases listed in Theorem C. (Consequently Theorem C holds also with l replaced by l^*.)

14. [*M35*] Complete the proof of Theorem C, by demonstrating that (a) step $r - 1$ is not a small step; and (b) $\lambda(a_{r-k})$ cannot be less than $m - 1$.

15. [*M42*] Write a computer program to extend Theorem C, characterizing all n such that $l(n) = \lambda(n) + 3$ and characterizing all n such that $l^*(n) = \lambda(n) + 3$.

16. [*HM15*] Show that Theorem D is not trivially true just because of the binary method; if $l^B(n)$ denotes the length of the addition chain for n produced by the binary S-and-X method, $l^B(n)/\lambda(n)$ does not approach a limit as $n \to \infty$.

17. [*M25*] Explain how to find the intervals $J_1, \ldots, J_h$ that are required in the proof of Lemma P.

18. [*HM24*] Let β be a positive constant. Show that there is a constant $\alpha < 2$ such that

$$\sum \binom{m+s}{t+v}\binom{t+v}{v}\beta^{2v}\binom{(m+s)^2}{t} < \alpha^m$$

for all large m, where the sum is over all s, t, v satisfying (30).

19. [*M23*] A "multiset" is like a set, but it may contain identical elements repeated a finite number of times. If A and B are multisets, we define new multisets $A \uplus B$, $A \cup B$, and $A \cap B$ in the following way: An element occurring exactly a times in A and b times in B occurs exactly $a+b$ times in $A \uplus B$, exactly $\max(a,b)$ times in $A \cup B$, and exactly $\min(a,b)$ times in $A \cap B$. (A "set" is a multiset that contains no elements more than once; if A and B are sets, so are $A \cup B$ and $A \cap B$, and the definitions given in this exercise agree with the customary definitions of set union and intersection.)

a) The prime factorization of an integer $n > 0$ is a multiset N whose elements are primes, where $\prod_{p \in N} = n$. The fact that every positive integer can be uniquely factored into primes gives us a one-to-one correspondence between the positive integers and the finite multisets of prime numbers; for example, if $n = 2^2 \cdot 3^3 \cdot 17$, the corresponding multiset is $N = \{2, 2, 3, 3, 3, 17\}$. If M and N are the multisets corresponding respectively to m and n, what multisets correspond to $\gcd(m,n)$, $\operatorname{lcm}(m,n)$, and mn?

b) Every monic polynomial $f(z)$ over the complex numbers corresponds in a natural way to the multiset F of its "roots"; we have $f(z) = \prod_{\zeta \in F}(z - \zeta)$. If $f(z)$ and $g(z)$ are the polynomials corresponding to the finite multisets F and G of complex numbers, what polynomials correspond to $F \uplus G$, $F \cup G$, and $F \cap G$?

c) Find as many interesting identities as you can that hold between multisets, with respect to the three operations $\uplus$, $\cup$, $\cap$.

20. [*M20*] What are the sequences S_i and M_{ij} ($0 \le i \le r$, $0 \le j \le t$) arising in Hansen's structural decomposition of star chains (a) of Type 3? (b) of Type 5? (The six "types" are defined in the proof of Theorem B.)

▶ **21.** [*M25*] (W. Hansen.) Let q be any positive integer. Find a value of n such that $l(n) \le l^*(n) - q$.

22. [*M20*] Prove that the addition chain constructed in the proof of Theorem F is an l^0-chain.

23. [*M20*] Prove Brauer's inequality (50).

▶ **24.** [*M22*] Generalize the proof of Theorem G to show that $l^0((B^n - 1)/(B-1)) \le (n-1)\,l^0(B) + l^0(n)$, for any integer $B > 1$; and prove that $l(2^{mn} - 1) \le l(2^m - 1) + mn - m + l^0(n)$.

25. [*20*] Let y be a fraction, $0 < y < 1$, expressed in the binary number system as $y = (.d_1 \ldots d_k)_2$. Design an algorithm to compute x^y using the operations of multiplication and square-root extraction.

▶ **26.** [*M24*] Design an efficient algorithm that computes the nth Fibonacci number F_n, modulo m, given large integers n and m.

▶ **27.** [*24*] (E. G. Straus.) Find a way to compute a general *monomial* $x_1^{n_1}x_2^{n_2}\dots x_m^{n_m}$ in at most $2\lambda(\max(n_1,n_2,\dots,n_m))+2^m-m-1$ multiplications.

28. [*M33*] (A. Schönhage.) The object of this exercise is to give a fairly short proof that $l(n)\geq\lambda(n)+\lg\nu(n)-O(\log\log(\nu(n)+1))$. (a) When $x=(x_k\dots x_0.x_{-1}\dots)_2$ and $y=(y_k\dots y_0.y_{-1}\dots)_2$ are real numbers written in binary notation, let us write $x\subseteq y$ if $x_j\leq y_j$ for all j. Give a simple rule for constructing the smallest number z with the property that $x'\subseteq x$ and $y'\subseteq y$ implies $x'+y'\subseteq z$. Denoting this number by $x\nabla y$, prove that $\nu(x\nabla y)\leq\nu(x)+\nu(y)$. (b) Given any addition chain (11) with $r=l(n)$, let the sequence d_0, d_1, ..., d_r be defined as in (35), and define the sequence A_0, A_1, ..., A_r by the following rules: $A_0=1$; if $a_i=2a_{i-1}$ then $A_i=2A_{i-1}$; otherwise if $a_i=a_j+a_k$ for some $0\leq k\leq j<i$, then $A_i=A_{i-1}\nabla(A_{i-1}/2^{d_j-d_k})$. Prove that this sequence "covers" the given chain, in the sense that $a_i\subseteq A_i$ for $0\leq i\leq r$. (c) Let δ be a positive integer (to be selected later). Call the nondoubling step $a_i=a_j+a_k$ a "baby step" if $d_j-d_k\geq\delta$, otherwise call it a "close step." Let $B_0=1$; $B_i=2B_{i-1}$ if $a_i=2a_{i-1}$; $B_i=B_{i-1}\nabla(B_{i-1}/2^{d_j-d_k})$ if $a_i=a_j+a_k$ is a baby step; and $B_i=\rho(2B_{i-1})$ otherwise, where $\rho(x)$ is the least number y such that $x/2^e\subseteq y$ for $0\leq e\leq\delta$. Show that $A_i\subseteq B_i$ and $\nu(B_i)\leq(1+\delta c_i)2^{b_i}$ for $0\leq i\leq r$, where b_i and c_i respectively denote the number of baby steps and close steps $\leq i$. [*Hint:* Show that the 1's in B_i appear in consecutive blocks of size $\geq 1+\delta c_i$.] (d) We now have $l(n)=r=b_r+c_r+d_r$ and $\nu(n)\leq\nu(B_r)\leq(1+\delta c_r)2^{b_r}$. Explain how to choose δ in order to obtain the inequality stated at the beginning of this exercise. [*Hint:* See (16), and note that $n\leq 2^r\alpha^{b_r}$ for some $\alpha<1$ depending on δ.]

29. [*M49*] Is $\nu(n)\leq 2^{l(n)-\lambda(n)}$ for all positive integers n? (If so, we have the lower bound $l(2^n-1)\geq n-1+\lceil\lg n\rceil$; cf. (17) and (49).)

30. [*20*] An *addition-subtraction chain* has the rule $a_i=a_j\pm a_k$ in place of (2); the imaginary computer described in the text has a new operation code, SUB. (This corresponds in practice to evaluating x^n using both multiplications and divisions.) Find an addition-subtraction chain, for some n, that has fewer than $l(n)$ steps.

31. [*M46*] (D. H. Lehmer.) Explore the problem of minimizing $\epsilon q+(r-q)$ in an addition chain (1), where q is the number of "simple" steps in which $a_i=a_{i-1}+1$, given a small positive "weight" ϵ. (This problem comes closer to reality for many calculations of x^n, if multiplication by x is simpler than a general multiplication; cf. Section 4.6.2.)

32. [*M30*] (A. C. Yao.) Let $l(n_1,\dots,n_m)$ be the length of the shortest addition chain that contains m given numbers $n_1<\cdots<n_m$. Prove that $l(n_1,\dots,n_m)\leq\lambda(n_m)+m\lambda(n_m)/\lambda\lambda(n_m)+O(\lambda(n_m)\lambda\lambda\lambda(n_m)/\lambda\lambda(n_m)^2)$, thereby generalizing (25).

33. [*M47*] What is the asymptotic value of $l(1,4,9,\dots,m^2)-m$, as $m\to\infty$, in the notation of exercise 32?

34. [*M50*] Is $l(2^n-1)\leq n-1+l(n)$ for all positive integers n? Does equality always hold? Does $l(n)=l^0(n)$?

35. [*M30*] (A. C. Yao, F. F. Yao, R. L. Graham.) Associate the "cost" a_ja_k with each step $a_i=a_j+a_k$ of an addition chain (1). Show that the left-to-right binary method yields a chain of minimum total cost, for all positive integers n.

36. [*15*] How many addition chains of length 9 have (52) as their reduced directed graph?

37. [*M23*] The binary addition chain for $n = 2^{e_0} + \cdots + 2^{e_t}$, when $e_0 > \cdots > e_t \geq 0$, is $1, 2, \ldots, 2^{e_0-e_1}, 2^{e_0-e_1}+1, \ldots, 2^{e_0-e_2}+2^{e_1-e_2}, 2^{e_0-e_2}+2^{e_1-e_2}+1, \ldots, n$. This corresponds to the S-and-X method described at the beginning of this section, while Algorithm A corresponds to the addition chain obtained by sorting the two sequences $(1, 2, 4, \ldots, 2^{e_0})$ and $(2^{e_t-1}+2^{e_t}, 2^{e_t-2}+2^{e_t-1}+2^{e_t}, \ldots, n)$ into increasing order. Prove or disprove: Each of these addition chains is a dual of the other.

38. [*M27*] How many addition chains without useless steps are equivalent to each of the addition chains discussed in exercise 37, when $e_0 > e_1 + 1$?

▶ **39.** [*M25*] (J. Olivos, 1979.) Let $l([n_1, n_2, \ldots, n_m])$ be the minimum number of multiplications needed to evaluate the monomial $x_1^{n_1} x_2^{n_2} \ldots x_m^{n_m}$ in the sense of exercise 27, where each n_i is a positive integer. Prove that this problem is equivalent to the problem of exercise 32, by showing that $l([n_1, n_2, \ldots, n_m]) = l(n_1, n_2, \ldots, n_m) + m - 1$. [*Hint:* Generalize the directed graph construction by considering graphs with more than one source vertex.]

▶ **40.** [*M21*] (J. Olivos.) Generalizing the factor method, prove that

$$l(m_1 n_1 + \cdots + m_t n_t) \leq l(m_1, \ldots, m_t) + l(n_1, \ldots, n_t) + t - 1,$$

where $l(n_1, \ldots, n_t)$ is defined in exercise 32.

41. [*M40*] (P. Downey, B. Leong, R. Sethi.) Let G be a connected graph with n vertices $\{1, \ldots, n\}$ and m edges, where the edges join u_j to v_j for $1 \leq j \leq m$. Prove that $l(1, 2, \ldots, 2^{An}, 2^{Au_1} + 2^{Av_1} + 1, \ldots, 2^{Au_m} + 2^{Av_m} + 1) = An + m + k$ for all sufficiently large A, where k is the minimum number of vertices in a vertex cover for G (i.e., a set containing either u_j or v_j for $1 \leq j \leq m$).

4.6.4. Evaluation of Polynomials

Now that we know efficient ways to evaluate the special polynomial x^n, let us consider the general problem of computing an nth degree polynomial

$$u(x) = u_n x^n + u_{n-1} x^{n-1} + \cdots + u_1 x + u_0, \qquad u_n \neq 0, \tag{1}$$

for given values of x. This problem arises frequently in practice.

In the following discussion we shall concentrate on minimizing the number of operations required to evaluate polynomials by computer, blithely assuming that all arithmetic operations are exact. Polynomials are most commonly evaluated using floating point arithmetic, which is not exact, and different schemes for the evaluation will, in general, give different answers. A numerical analysis of the accuracy achieved depends on the coefficients of the particular polynomial being considered, and is beyond the scope of this book; the reader should be careful to investigate the accuracy of any calculations undertaken with floating point arithmetic. In most cases the methods we shall describe turn out to be reasonably satisfactory from a numerical standpoint, but many bad examples can also be given. [See Webb Miller, *SIAM J. Computing* **4** (1975), 105–107, for a survey of the literature on stability of fast polynomial evaluation, and for

a demonstration that certain kinds of numerical stability cannot be guaranteed for some families of high-speed algorithms.]

A beginning programmer will often evaluate the polynomial (1) in a manner corresponding directly to its conventional textbook form: First $u_n x^n$ is calculated, then $u_{n-1}x^{n-1}$, ..., $u_1 x$, and finally all of the terms of (1) are added together. But even if the efficient methods of Section 4.6.3 are used to evaluate powers of x in this approach, the resulting calculation is needlessly slow unless nearly all of the coefficients u_k are zero. If the coefficients are all nonzero, an obvious alternative would be to evaluate (1) from right to left, computing the values of x^k and $u_k x^k + \cdots + u_0$ for $k = 1, \ldots, n$. Such a process involves $2n-1$ multiplications and n additions, and it might also require further instructions to store and retrieve intermediate results from memory.

Horner's rule. One of the first things a novice programmer is usually taught is an elegant way to rearrange this computation, by evaluating $u(x)$ as follows:

$$u(x) = ((\ldots(u_n x + u_{n-1})x + \cdots)x + u_0. \tag{2}$$

Start with u_n, multiply by x, add u_{n-1}, multiply by x, ..., multiply by x, add u_0. This form of the computation is usually called "Horner's rule"; we have already seen it used in connection with radix conversion in Section 4.4. The entire process requires n multiplications and n additions, minus one addition for each coefficient that is zero. Furthermore, there is no need to store partial results, since each quantity arising during the calculation is used immediately after it has been computed.

W. G. Horner gave this rule early in the nineteenth century [*Philosophical Transactions*, Royal Society of London **109** (1819), 308–335] in connection with a procedure for calculating polynomial roots. The fame of the latter method [see J. L. Coolidge, *Mathematics of Great Amateurs* (Oxford, 1949), Chapter 15] accounts for the fact that Horner's name has been attached to (2); but actually Isaac Newton had made use of the same idea 150 years earlier. In a well-known work entitled *De Analysi per Æquationes Infinitas*, originally written in 1669, Newton wrote

$$\overline{\overline{y-4} \times y : +5} \times y : -12 \times y : +17$$

for the polynomial $y^4 - 4y^3 + 5y^2 - 12y + 17$; this clearly uses the idea of (2), since he often denoted grouping by using horizontal lines and colons instead of parentheses. [See D. T. Whiteside, ed., *The Mathematical Papers of Isaac Newton* **2** (Cambridge Univ. Press, 1968), 222.]

Several generalizations of Horner's rule have been suggested. Let us first consider evaluating $u(z)$ when z is a complex number, while the coefficients u_k are real. In particular, when $z = e^{i\theta} = \cos\theta + i\sin\theta$, the polynomial $u(z)$ is essentially two Fourier series,

$$(u_0 + u_1\cos\theta + \cdots + u_n\cos n\theta) \;+\; i(u_1\sin\theta + \cdots + u_n\sin n\theta).$$

Complex addition and multiplication can obviously be reduced to a sequence of ordinary operations on real numbers:

real + complex	requires	1 addition
complex + complex	requires	2 additions
real × complex	requires	2 multiplications
complex × complex	requires	4 multiplications, 2 additions
	or	3 multiplications, 5 additions

(See exercise 41. Subtraction is here considered as if it were equivalent to addition.) Therefore Horner's rule (2) uses either $4n - 2$ multiplications and $3n - 2$ additions or $3n - 1$ multiplications and $6n - 5$ additions to evaluate $u(z)$ when $z = x + iy$ is complex. Actually $2n - 4$ of these additions can be saved, since we are multiplying by the same number z each time. An alternative procedure for evaluating $u(x + iy)$ is to let

$$\begin{aligned} &a_1 = u_n, \qquad b_1 = u_{n-1}, \qquad r = x + x, \qquad s = x^2 + y^2; \\ &a_j = b_{j-1} + ra_{j-1}, \qquad b_j = u_{n-j} - sa_{j-1}, \qquad 1 < j \le n. \end{aligned} \tag{3}$$

Then it is easy to prove by induction that $u(z) = za_n + b_n$. This scheme [*BIT* **5** (1965), 142; cf. also G. Goertzel, *AMM* **65** (1958), 34–35] requires only $2n + 2$ multiplications and $2n + 1$ additions, so it is an improvement over Horner's rule when $n \ge 3$. In the case of Fourier series, when $z = e^{i\theta}$, we have $s = 1$, so the number of multiplications drops to $n + 1$. The moral of this story is that a good programmer does not make indiscriminate use of the built-in "complex arithmetic" features of high-level programming languages.

Consider the process of dividing the polynomial $u(x)$ by $x - x_0$, using Algorithm 4.6.1D to obtain $u(x) = (x - x_0)q(x) + r(x)$; here $\deg(r) < 1$, so $r(x)$ is a constant independent of x, and $u(x_0) = 0 \cdot q(x_0) + r = r$. An examination of this division process reveals that the computation is essentially the same as Horner's rule for evaluating $u(x_0)$. Similarly, if we divide $u(z)$ by the polynomial $(z - z_0)(z - \bar{z}_0) = z^2 - 2x_0z + x_0^2 + y_0^2$, the resulting computation turns out to be equivalent to (3); we obtain $u(z) = (z - z_0)(z - \bar{z}_0)q(z) + a_nz + b_n$, hence $u(z_0) = a_nz_0 + b_n$.

In general, if we divide $u(x)$ by $f(x)$ to obtain $u(x) = f(x)q(x) + r(x)$, and if $f(x_0) = 0$, we have $u(x_0) = r(x_0)$; this observation leads to further generalizations of Horner's rule. For example, we may let $f(x) = x^2 - x_0^2$; this yields the "second-order" Horner's rule

$$\begin{aligned} u(x) = {} & \big(\ldots(u_{2\lfloor n/2\rfloor}x^2 + u_{2\lfloor n/2\rfloor-2})x^2 + \cdots\big)x^2 + u_0 \\ & + \big((\ldots(u_{2\lceil n/2\rceil-1}x^2 + u_{2\lceil n/2\rceil-3})x^2 + \cdots)x^2 + u_1\big)x. \end{aligned} \tag{4}$$

The second-order rule uses $n+1$ multiplications and n additions (see exercise 5); so it is no improvement over Horner's rule from this standpoint. But there are at least two circumstances in which (4) is useful: If we want to evaluate both $u(x)$

and $u(-x)$, this approach yields $u(-x)$ with just one more addition operation; two values can be obtained almost as cheaply as one. Moreover, if we have a computer that allows parallel computations, the two lines of (4) may be evaluated independently, so we save about half the running time.

When our computer allows parallel computation on k arithmetic units at once, a "kth-order" Horner's rule (obtained in a similar manner from $f(x) = x^k - x_0^k$) may be used. Another attractive method for parallel computation has been suggested by G. Estrin [*Proc. Western Joint Computing Conf.* **17** (1960), 33–40]; for $n = 7$, Estrin's method is:

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
$a_1 = u_7x + u_6$	$b_1 = u_5x + u_4$	$c_1 = u_3x + u_2$	$d_1 = u_1x + u_0$	x^2
$a_2 = a_1x^2 + b_1$		$c_2 = c_1x^2 + d_1$		x^4
$a_3 = a_2x^4 + c_2$				

Here $a_3 = u(x)$. However, an interesting analysis by W. S. Dorn [*IBM J. Res. and Devel.* **6** (1962), 239–245] shows that these methods might not actually be an improvement over the second-order rule, if each arithmetic unit must access a memory that communicates with only one processor at a time.

Tabulating polynomial values. If we wish to evaluate an nth degree polynomial at many points in an arithmetic progression (i.e., if we want to calculate $u(x_0)$, $u(x_0 + h)$, $u(x_0 + 2h)$, ...), the process can be reduced to addition only, after the first few steps. For if we start with any sequence of numbers $(\alpha_0, \alpha_1, \ldots, \alpha_n)$ and apply the transformation

$$\alpha_0 \leftarrow \alpha_0 + \alpha_1, \quad \alpha_1 \leftarrow \alpha_1 + \alpha_2, \quad \ldots, \quad \alpha_{n-1} \leftarrow \alpha_{n-1} + \alpha_n, \tag{5}$$

we find that k applications of (5) yields

$$\alpha_j^{(k)} = \binom{k}{0}\beta_j + \binom{k}{1}\beta_{j+1} + \binom{k}{2}\beta_{j+2} + \cdots, \qquad 0 \le j \le n,$$

where β_j denotes the initial value of α_j and $\beta_j = 0$ for $j > n$. In particular,

$$\alpha_0^{(k)} = \binom{k}{0}\beta_0 + \binom{k}{1}\beta_1 + \cdots + \binom{k}{n}\beta_n \tag{6}$$

is a polynomial of degree n in k. By properly choosing the β's, as shown in exercise 7, we can arrange things so that $\alpha_0^{(k)}$ is the desired value $u(x_0 + kh)$, for all k. In other words, each execution of the n additions in (5) will produce the next value of the given polynomial.

Caution: Rounding errors can accumulate after many repetitions of (5), and an error in α_j produces a corresponding error in the coefficients of $x^0, \ldots, x^j$ in the polynomial being computed. Therefore the values of the α's should be "refreshed" after a large number of iterations.

Derivatives and changes of variable. Sometimes we want to find the coefficients of $u(x+x_0)$, given a constant x_0 and the coefficients of $u(x)$. For example, if $u(x) = 3x^2 + 2x - 1$, then $u(x-2) = 3x^2 - 10x + 7$. This is analogous to a radix conversion problem, converting from base x to base $x+2$.

By Taylor's theorem, the desired coefficients are given by the derivatives of $u(x)$ at $x = x_0$, namely

$$u(x+x_0) = u(x_0) + u'(x_0)x + (u''(x_0)/2!)x^2 + \cdots + (u^{(n)}(x_0)/n!)x^n, \qquad (7)$$

so the problem is equivalent to evaluating $u(x)$ and all its derivatives.

If we write $u(x) = q(x)(x - x_0) + r$, then $u(x+x_0) = q(x+x_0)x + r$; so r is the constant coefficient of $u(x+x_0)$, and the problem reduces to finding the coefficients of $q(x+x_0)$, where $q(x)$ is a known polynomial of degree $n-1$. Thus the following algorithm is indicated:

H1. Set $v_j \leftarrow u_j$ for $0 \le j \le n$.

H2. For $k = 0, 1, \ldots, n-1$ (in this order), set $v_j \leftarrow v_j + x_0 v_{j+1}$ for $j = n-1, \ldots, k+1, k$ (in this order). ∎

At the conclusion of step H2 we have $u(x+x_0) = v_n x^n + \cdots + v_1 x + v_0$. This procedure was a principal part of Horner's root-finding method, and when $k = 0$ it is exactly rule (2) for evaluating $u(x_0)$.

Horner's method requires $(n^2+n)/2$ multiplications and $(n^2+n)/2$ additions; but notice that if $x_0 = 1$ we avoid all of the multiplications. Fortunately we can reduce the general problem to the case $x_0 = 1$ by introducing comparatively few multiplications and divisions:

S1. Compute and store the values $x_0^2, \ldots, x_0^n$.

S2. Set $v_j \leftarrow u_j x_0^j$ for $0 \le j \le n$. (Now $v(x) = u(x_0 x)$.)

S3. Perform step H2 but with $x_0 = 1$. (Now $v(x) = u\big(x_0(x+1)\big) = u(x_0 x + x_0)$.)

S4. Set $v_j \leftarrow v_j / x_0^j$ for $0 < j \le n$. (Now $v(x) = u(x+x_0)$ as desired.) ∎

This idea, due to M. Shaw and J. F. Traub [*JACM* **21** (1974), 161–167], has the same number of additions and the same numerical stability as Horner's method, but it needs only $2n-1$ multiplications and $n-1$ divisions. About $\frac{1}{2}n$ of these multiplications can, in turn, be avoided (see exercise 6).

If we want only the first few or the last few derivatives, Shaw and Traub have observed that there are further ways to save time. For example, if we just want to evaluate $u(x)$ and $u'(x)$, we can do the job with $2n-1$ additions and about $n + \sqrt{2n}$ multiplications/divisions as follows:

D1. Compute and store the values $x^2, x^3, \ldots, x^t, x^{2t}$, where $t = \lceil \sqrt{n/2} \rceil$.

D2. Set $v_j \leftarrow u_j x^{f(j)}$ for $0 \le j \le n$, where $f(j) = t - 1 - \big((n-1-j) \bmod 2t\big)$ for $0 \le j < n$, and $f(n) = t$.

D3. Set $v_j \leftarrow v_j + v_{j+1} x^{g(j)}$ for $j = n-1, \ldots, 1, 0$; here $g(j) = 2t$ when $n-j$ is a multiple of $2t$, otherwise $g(j) = 0$ and the multiplication by $x^{g(j)}$ need not be done.

D4. Set $v_j \leftarrow v_j + v_{j+1} x^{g(j)}$ for $j = n-1, \ldots, 2, 1$. Now $v_0 / x^{f(0)} = u(x)$ and $v_1 / x^{f(1)} = u'(x)$. ∎

Adaptation of coefficients. Let us now return to our original problem of evaluating a given polynomial $u(x)$ as rapidly as possible, for "random" values of x. The importance of this problem is due partly to the fact that standard functions such as $\sin x$, $\cos x$, e^x, etc., are usually computed by subroutines that rely on the evaluation of certain polynomials; such polynomials are evaluated so often, it is desirable to find the fastest possible way to do the computation.

Arbitrary polynomials of degree five and higher can be evaluated with fewer operations than Horner's rule requires, if we first "adapt" or "precondition" the coefficients $u_0, u_1, \ldots, u_n$. This adaptation process might involve a lot of work, as explained below; but the preliminary calculation is not wasted, since it must be done only once while the polynomial will be evaluated many times. For examples of "adapted" polynomials for standard functions, see V. Ia. Pan, *USSR Computational Math. and Math. Physics* **2** (1963), 137–146.

The simplest case for which adaptation of coefficients is helpful occurs for a fourth degree polynomial:

$$u(x) = u_4x^4 + u_3x^3 + u_2x^2 + u_1x + u_0, \qquad u_4 \neq 0. \tag{8}$$

This equation can be rewritten in a form originally suggested by T. S. Motzkin,

$$y = (x + \alpha_0)x + \alpha_1, \qquad u(x) = \big((y + x + \alpha_2)y + \alpha_3\big)\alpha_4, \tag{9}$$

for suitably "adapted" coefficients α_0, α_1, α_2, α_3, α_4. The computation in (9) involves three multiplications, five additions, and (on a one-accumulator machine like MIX) one instruction to store the partial result y into temporary storage. By comparison with Horner's rule, we have traded a multiplication for an addition and a possible storage command. Even this comparatively small savings is worthwhile if the polynomial is to be evaluated often. (Of course, if the time for multiplication is comparable to the time for addition, (9) gives no improvement; we will see that a general fourth-degree polynomial always requires at least eight arithmetic operations for its evaluation.)

By equating coefficients in (8) and (9), we obtain formulas for computing the α_j's in terms of the u_k's:

$$\begin{aligned} &\alpha_0 = \tfrac{1}{2}(u_3/u_4 - 1), \qquad \beta = u_2/u_4 - \alpha_0(\alpha_0 + 1), \qquad \alpha_1 = u_1/u_4 - \alpha_0\beta, \\ &\alpha_2 = \beta - 2\alpha_1, \qquad \alpha_3 = u_0/u_4 - \alpha_1(\alpha_1 + \alpha_2), \qquad \alpha_4 = u_4. \end{aligned} \tag{10}$$

A similar scheme, which evaluates a fourth-degree polynomial in the same number of steps as (9), appears in exercise 18; this alternative method will give greater numerical accuracy than (9) in certain cases, although it yields poorer accuracy in others.

Polynomials that arise in practice often have a rather small leading coefficient, so that the division by u_4 in (10) leads to instability. In such a case it is usually preferable to replace x by $|u_4|^{1/4}x$ as the first step, reducing (8) to a polynomial whose leading coefficient is ± 1. A similar transformation applies to

polynomials of higher degrees. This idea is due to C. T. Fike [*CACM* **10** (1967), 175–178], who has presented several interesting examples.

Any polynomial of the fifth degree may be evaluated using four multiplications, six additions, and one storing, by using the rule $u(x) = U(x)x + u_0$, where $U(x) = u_5x^4 + u_4x^3 + u_3x^2 + u_2x + u_1$ is evaluated as in (9). Alternatively, we can do the evaluation with four multiplications, five additions, and three storings, if the calculations take the form

$$y = (x + \alpha_0)^2, \qquad u(x) = \big(((y + \alpha_1)y + \alpha_2)(x + \alpha_3) + \alpha_4\big)\alpha_5. \tag{11}$$

The determination of the α's this time requires the solution of a cubic equation (see exercise 19).

On many computers the number of "storing" operations required by (11) is less than 3; for example, we may be able to compute $(x + \alpha_0)^2$ without storing $x + \alpha_0$. In fact, many computers have more than one arithmetic register for floating point calculations, so we can avoid storing altogether. Because of the wide variety of features available for arithmetic on different computers, we shall henceforth in this section count only the arithmetic operations, not the operations of storing and loading an accumulator. The computation schemes can usually be adapted to any particular computer in a straightforward manner, so that very few of these auxiliary operations are necessary; on the other hand, it must be remembered that this extra overhead might well overshadow the fact that we are saving a multiplication or two, especially if the machine code is being produced by a compiler that does not "optimize."

A polynomial $u(x) = u_6x^6 + \cdots + u_1x + u_0$ of degree six can always be evaluated using four multiplications and seven additions, with the scheme

$$\begin{aligned} z &= (x + \alpha_0)x + \alpha_1, \qquad w = (x + \alpha_2)z + \alpha_3, \\ u(x) &= \big((w + z + \alpha_4)w + \alpha_5\big)\alpha_6. \end{aligned} \tag{12}$$

[See D. E. Knuth, *CACM* **5** (1962), 595–599.] This saves two of the six multiplications required by Horner's rule. Here again we must solve a cubic equation: Since $\alpha_6 = u_6$, we may assume that $u_6 = 1$. Under this assumption, let $\beta_1 = \frac{1}{2}(u_5 - 1)$, $\beta_2 = u_4 - \beta_1(\beta_1 + 1)$, $\beta_3 = u_3 - \beta_1\beta_2$, $\beta_4 = \beta_1 - \beta_2$, $\beta_5 = u_2 - \beta_1\beta_3$. Let β_6 be a real root of the cubic equation

$$2y^3 + (2\beta_4 - \beta_2 + 1)y^2 + (2\beta_5 - \beta_2\beta_4 - \beta_3)y + (u_1 - \beta_2\beta_5) = 0. \tag{13}$$

(This equation always has a real root, since the polynomial on the left approaches $+\infty$ for large positive y, and it approaches $-\infty$ for large negative y; it must assume the value zero somewhere in between.) Now if we define

$$\beta_7 = \beta_6^2 + \beta_4\beta_6 + \beta_5, \qquad \beta_8 = \beta_3 - \beta_6 - \beta_7,$$

we have finally

$$\begin{aligned} &\alpha_0 = \beta_2 - 2\beta_6, \qquad \alpha_2 = \beta_1 - \alpha_0, \qquad \alpha_1 = \beta_6 - \alpha_0\alpha_2, \\ &\alpha_3 = \beta_7 - \alpha_1\alpha_2, \qquad \alpha_4 = \beta_8 - \beta_7 - \alpha_1, \qquad \alpha_5 = u_0 - \beta_7\beta_8. \end{aligned} \tag{14}$$

We can illustrate this procedure with a contrived example: Suppose that we want to evaluate $x^6+13x^5+49x^4+33x^3-61x^2-37x+3$. We obtain $\alpha_6=1$, $\beta_1=6$, $\beta_2=7$, $\beta_3=-9$, $\beta_4=-1$, $\beta_5=-7$, and so we meet with the cubic equation

$$2y^3-8y^2+2y+12=0. \tag{15}$$

This equation has $\beta_6=2$ as a root, and we continue to find $\beta_7=-5$, $\beta_8=-6$, $\alpha_0=3$, $\alpha_2=3$, $\alpha_1=-7$, $\alpha_3=16$, $\alpha_4=6$, $\alpha_5=-27$. The resulting scheme is therefore

$$z=(x+3)x-7, \qquad w=(x+3)z+16, \qquad u(x)=(w+z+6)w-27.$$

By sheer coincidence the quantity $x+3$ appears twice here, so we have found a method that uses three multiplications and six additions.

Another method for handling sixth-degree equations has been suggested by V. Ĭa. Pan [*Problemy Kibernetiki* **5** (1961), 17–29]. His method requires one more addition operation, but it involves only rational operations in the preliminary steps; no cubic equation needs to be solved. We may proceed as follows:

$$\begin{aligned} z&=(x+\alpha_0)x+\alpha_1, \qquad w=z+x+\alpha_2,\\ u(x)&=\big(((z-x+\alpha_3)w+\alpha_4)z+\alpha_5\big)\alpha_6. \end{aligned} \tag{16}$$

To determine the α's, we divide the polynomial once again by $u_6=\alpha_6$ so that $u(x)$ becomes monic. It can then be verified that $\alpha_0=u_5/3$ and that

$$\alpha_1=(u_1-\alpha_0u_2+\alpha_0^2u_3-\alpha_0^3u_4+2\alpha_0^5)/(u_3-2\alpha_0u_4+5\alpha_0^3). \tag{17}$$

Note that Pan's method requires that the denominator in (17) does not vanish. In other words, (16) can be used only when

$$27u_3u_6^2-18u_6u_5u_4+5u_5^3\neq 0; \tag{18}$$

in fact, this quantity should not be so small that α_1 becomes too large. Once α_1 has been determined, the remaining α's may be determined from the equations

$$\begin{gathered} \beta_1=2\alpha_0, \qquad \beta_2=u_4-\alpha_0\beta_1-\alpha_1,\\ \beta_3=u_3-\alpha_0\beta_2-\alpha_1\beta_1, \qquad \beta_4=u_2-\alpha_0\beta_3-\alpha_1\beta_2,\\ \alpha_3=\tfrac12\big(\beta_3-(\alpha_0-1)\beta_2+(\alpha_0-1)(\alpha_0^2-1)\big)-\alpha_1,\\ \alpha_2=\beta_2-(\alpha_0^2-1)-\alpha_3-2\alpha_1,\\ \alpha_4=\beta_4-(\alpha_2+\alpha_1)(\alpha_3+\alpha_1),\\ \alpha_5=u_0-\alpha_1\beta_4. \end{gathered} \tag{19}$$

We have discussed the cases of degree $n=4$, 5, 6 in detail because the smaller values of n arise most frequently in applications. Let us now consider a general evaluation scheme for nth degree polynomials, a method that involves at most $\lfloor n/2\rfloor+2$ multiplications and n additions.

Theorem E. *Every nth degree polynomial* (1) *with real coefficients,* $n \geq 3$, *can be evaluated by the scheme*

$$y = x + c, \qquad w = y^2;$$
$$z = (u_n y + \alpha_0)y + \beta_0 \quad (n \text{ even}), \qquad z = u_n y + \beta_0 \quad (n \text{ odd});$$
$$u(x) = (\ldots((z(w-\alpha_1)+\beta_1)(w-\alpha_2)+\beta_2)\ldots)(w-\alpha_m)+\beta_m; \tag{20}$$

for suitable real parameters c, α_k *and* β_k, *where* $m = \lfloor n/2 \rfloor - 1$. *In fact, it is possible to select these parameters so that* $\beta_m = 0$.

Proof. Let us first examine the circumstances under which the α's and β's can be chosen in (20), if c is fixed. Let

$$p(x) = u(x-c) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0. \tag{21}$$

We want to show that $p(x)$ has the form $p_1(x)(x^2-\alpha_m)+\beta_m$ for some polynomial $p_1(x)$ and some constants α_m, β_m. If we divide $p(x)$ by $x^2-\alpha_m$, we can see that the remainder β_m is a constant only if the auxiliary polynomial

$$q(x) = a_{2m+1}x^m + a_{2m-1}x^{m-1} + \cdots + a_1, \tag{22}$$

formed from every odd-numbered coefficient of $p(x)$, is a multiple of $x - \alpha_m$. Conversely, if $q(x)$ has $x - \alpha_m$ as a factor, then $p(x) = p_1(x)(x^2 - \alpha_m) + \beta_m$, for some constant β_m that may be determined by division.

Similarly, we want $p_1(x)$ to have the form $p_2(x)(x^2 - \alpha_{m-1}) + \beta_{m-1}$, and this is the same as saying that $q(x)/(x - \alpha_m)$ is a multiple of $x - \alpha_{m-1}$; for if $q_1(x)$ is the polynomial corresponding to $p_1(x)$ as $q(x)$ corresponds to $p(x)$, we have $q_1(x) = q(x)/(x - \alpha_m)$. Continuing in the same way, we find that the parameters $\alpha_1, \beta_1, \ldots, \alpha_m, \beta_m$ will exist if and only if

$$q(x) = a_{2m+1}(x-\alpha_1)\ldots(x-\alpha_m). \tag{23}$$

In other words, either $q(x)$ is identically zero (and this can happen only when n is even), or else $q(x)$ is an mth degree polynomial having all real roots.

Now we have a surprising fact discovered by J. Eve [*Numer. Math.* **6** (1964), 17–21]: *If* $p(x)$ *has at least* $n-1$ *complex roots whose real parts are all nonnegative, or all nonpositive, then the corresponding polynomial* $q(x)$ *is identically zero or has all real roots.* (See exercise 23.) Since $u(x) = 0$ if and only if $p(x+c) = 0$, we need merely choose the parameter c large enough that at least $n-1$ of the roots of $u(x) = 0$ have a real part $\geq -c$, and (20) will apply whenever $a_{n-1} = u_{n-1} - ncu_n \neq 0$.

We can also determine c so that these conditions are fulfilled and also that $\beta_m = 0$. First the n roots of $u(x) = 0$ are determined. If $a + bi$ is a root having the largest or the smallest real part, and if $b \neq 0$, let $c = -a$ and $\alpha_m = -b^2$; then $x^2 - \alpha_m$ is a factor of $u(x-c)$. If the root with smallest or

largest real part is real, but the root with *second* smallest (or second largest) real part is nonreal, the same transformation applies. If the two roots with smallest (or largest) real parts are both real, they can be expressed in the form $a - b$ and $a + b$, respectively; let $c = -a$ and $\alpha_m = b^2$. Again $x^2 - \alpha_m$ is a factor of $u(x - c)$. (Still other values of c are often possible; see exercise 24.) The coefficient a_{n-1} will be nonzero for at least one of these alternatives, unless $q(x)$ is identically zero. ∎

Note that this method of proof usually gives at least two values of c, and we also have the chance to permute $\alpha_1, \ldots, \alpha_{m-1}$ in $(m-1)!$ ways. Some of these alternatives may give more desirable numerical accuracy than others.

***Polynomial chains.** Now let us consider questions of optimality. What are the *best possible* schemes for evaluating polynomials of various degrees, in terms of the minimum possible number of arithmetic operations? This question was first analyzed by A. M. Ostrowski in the case that no preliminary adaptation of coefficients is allowed [*Studies in Mathematics and Mechanics presented to R. von Mises* (New York: Academic Press, 1954), 40–48], and by T. S. Motzkin in the case of adapted coefficients [cf. *Bull. Amer. Math. Soc.* **61** (955), 163].

In order to investigate this question, we can extend Section 4.6.3's concept of addition chains to the notion of *polynomial chains.* A polynomial chain is a sequence of the form

$$x = \lambda_0, \quad \lambda_1, \quad \ldots, \quad \lambda_r = u(x), \tag{24}$$

where $u(x)$ is some polynomial in x, and for $1 \le i \le r$

$$\begin{aligned} &\text{either } \lambda_i = (\pm\lambda_j) \circ \lambda_k, && 0 \le j, k < i, \\ &\text{or } \lambda_i = \alpha_j \circ \lambda_k, && 0 \le k < i. \end{aligned} \tag{25}$$

Here "$\circ$" denotes any of the three operations "$+$", "$-$", or "$\times$", and α_j denotes a so-called parameter. Steps of the first kind are called *chain steps*, and steps of the second kind are called *parameter steps.* We shall assume that a different parameter α_j is used in each parameter step; if there are s parameter steps, they should involve $\alpha_1, \alpha_2, \ldots, \alpha_s$ in this order.

It follows that the polynomial $u(x)$ at the end of the chain has the form

$$u(x) = q_n x^n + \cdots + q_1 x + q_0, \tag{26}$$

where $q_n, \ldots, q_1, q_0$ are polynomials in $\alpha_1, \alpha_2, \ldots, \alpha_s$ with integer coefficients. We shall interpret the parameters $\alpha_1, \alpha_2, \ldots, \alpha_s$ as real numbers, and we shall therefore restrict ourselves to considering the evaluation of polynomials with real coefficients. The *result set* R of a polynomial chain is defined to be the set of all vectors $(q_n, \ldots, q_1, q_0)$ of real numbers that occur as $\alpha_1, \alpha_2, \ldots, \alpha_s$ independently assume all possible real values.

If for every choice of $t + 1$ distinct integers $j_0, \ldots, j_t \in \{0, 1, \ldots, n\}$ there is a nonzero multivariate polynomial $f_{j_0 \ldots j_t}$ with integer coefficients such that

$f_{j_0 \ldots j_t}(q_{j_0}, \ldots, q_{j_t}) = 0$ for all $(q_n, \ldots, q_1, q_0)$ in R, let us say that the result set R has at most t *degrees of freedom*, and that the chain (24) has at most t degrees of freedom. We also say that the chain (24) *computes* a given polynomial $u(x) = u_n x^n + \cdots + u_1 x + u_0$ if $(u_n, \ldots, u_1, u_0)$ is in R. It follows that a polynomial chain with at most n degrees of freedom cannot compute all nth degree polynomials (see exercise 27).

As an example of a polynomial chain, consider the following chain corresponding to Theorem E, when n is odd:

$$\begin{aligned} \lambda_0 &= x \\ \lambda_1 &= \alpha_1 + \lambda_0 \\ \lambda_2 &= \lambda_1 \times \lambda_1 \\ \lambda_3 &= \alpha_2 \times \lambda_1 \\ \left.\begin{aligned} \lambda_{1+3i} &= \alpha_{1+2i} + \lambda_{3i} \\ \lambda_{2+3i} &= \alpha_{2+2i} + \lambda_2 \\ \lambda_{3+3i} &= \lambda_{1+3i} \times \lambda_{2+3i} \end{aligned}\right\} & \quad 1 \le i < n/2. \end{aligned} \tag{27}$$

There are $\lfloor n/2 \rfloor + 2$ multiplications and n additions; $\lfloor n/2 \rfloor + 1$ chain steps and $n + 1$ parameter steps. By Theorem E, the result set R includes the set of all $(u_n, \ldots, u_1, u_0)$ with $u_n \neq 0$, so (27) computes all polynomials of degree n. We cannot prove that R has at most n degrees of freedom, since the result set has $n + 1$ independent components.

A polynomial chain with s parameter steps has at most s degrees of freedom. In a sense, this is obvious: we can't compute a function with t degrees of freedom using fewer than t arbitrary parameters. But this intuitive fact is not easy to prove formally; for example, there are continuous functions ("space-filling curves") that map the real line onto a plane, and such functions map a single parameter into two independent parameters. For our purposes, we need to verify that no polynomial functions with integer coefficients can have such a property; a proof appears in exercise 28.

Given this fact, we can proceed to prove the results we seek:

Theorem M (T. S. Motzkin, 1954). *A polynomial chain with $m > 0$ multiplications has at most $2m$ degrees of freedom.*

Proof. Let $\mu_1, \mu_2, \ldots, \mu_m$ be the λ_i's of the chain that are multiplication operations. Then

$$\begin{aligned} \mu_i &= S_{2i-1} \times S_{2i}, \qquad 1 \le i \le m, \\ u(x) &= S_{2m+1}, \end{aligned} \tag{28}$$

where each S_j is a certain sum of μ's, x's, and α's. Write $S_j = T_j + \beta_j$, where T_j is a sum of μ's and x's while β_j is a sum of α's.

Now $u(x)$ is expressible as a polynomial in $x, \beta_1, \ldots, \beta_{2m+1}$ with integer coefficients. Since the β's are expressible as linear functions of $\alpha_1, \ldots, \alpha_s$, the set of values represented by all real values of $\beta_1, \ldots, \beta_{2m+1}$ contains the result set of the chain. Therefore there are at most $2m + 1$ degrees of freedom; this can be improved to $2m$ when $m > 0$, as shown in exercise 30. ∎

An example of the construction in the proof of Theorem M appears in exercise 25. A similar result can be proved for additions:

Theorem A (É. G. Belaga, 1958). *A polynomial chain containing q additions and subtractions has at most $q+1$ degrees of freedom.*

Proof. [*Problemi Kibernetiki* **5** (1961), 7–15.] Let $\kappa_1, \ldots, \kappa_q$ be the λ_i's of the chain that correspond to addition or subtraction operations. Then

$$\begin{aligned} \kappa_i &= \pm T_{2i-1} \pm T_{2i}, \qquad 1 \le i \le q; \\ u(x) &= T_{2q+1}, \end{aligned} \tag{29}$$

where each T_j is a product of κ's, x's, and α's. We may write $T_j = A_j B_j$, where A_j is a product of α's and B_j is a product of κ's and x's. The following transformation may now be made to the chain, successively for $i = 1, 2, \ldots, q$: Let $\beta_i = A_{2i}/A_{2i-1}$, so that $\kappa_i = A_{2i-1}(\pm B_{2i-1} \pm \beta_i B_{2i})$. Then change κ_i to $\pm B_{2i-1} \pm \beta_i B_{2i}$, and replace each occurrence of κ_i in future formulas T_{2i+1}, $T_{2i+2}, \ldots, T_{2q+1}$ by $A_{2i-1}\kappa_i$. (This replacement may change the values of $A_{2i+1}, A_{2i+2}, \ldots, A_{2q+1}$.)

After the above transformation has been done for all i, let $\beta_{q+1} = A_{2q+1}$; then $u(x)$ can be expressed as a polynomial in $\beta_1, \ldots, \beta_{q+1}$, and x, with integer coefficients. We are almost ready to complete the proof, but we must be careful because the polynomials obtained as $\beta_1, \ldots, \beta_{q+1}$ range over all real values may not include all polynomials representable by the original chain (see exercise 26); it is possible to have $A_{2i-1} = 0$, for some values of the α's, and this makes β_i undefined.

To complete the proof, let us observe that the result set R of the original chain can be written $R = R_1 \cup R_2 \cup \cdots \cup R_q \cup R'$, where R_i is the set of result vectors possible when $A_{2i-1} = 0$, and where R' is the set of result vectors possible when all α's are nonzero. The discussion above proves that R' has at most $q+1$ degrees of freedom. If $A_{2i-1} = 0$, then $T_{2i-1} = 0$, so addition step κ_i may be dropped to obtain another chain computing the result set R_i; by induction we see that each R_i has at most q degrees of freedom. Hence by exercise 29, R has at most $q+1$ degrees of freedom. ∎

Theorem C. *If a polynomial chain* (24) *computes all nth degree polynomials $u(x) = u_n x^n + \cdots + u_0$, for some $n \ge 2$, then it includes at least $\lfloor n/2 \rfloor + 1$ multiplications and at least n addition-subtractions.*

Proof. Let there be m multiplication steps. By Theorem M, the chain has at most $2m$ degrees of freedom, so $2m \ge n+1$. Similarly, by Theorem A there are $\ge n$ addition-subtractions. ∎

This theorem states that no *single* method having fewer than $\lfloor n/2 \rfloor + 1$ multiplications or fewer than n additions can evaluate all possible nth degree polynomials. The result of exercise 29 allows us to strengthen this and say that no finite collection of such polynomial chains will suffice for all polynomials of a given degree. Some special polynomials can, of course, be evaluated more

efficiently; all we have really proved is that polynomials whose coefficients are *algebraically independent*, in the sense that they satisfy no nontrivial polynomial equation, require $\lfloor n/2 \rfloor + 1$ multiplications and n additions. Unfortunately the coefficients we deal with in computers are always rational numbers, so the above theorems don't really apply; in fact, exercise 42 shows that we can always get by with $O(\sqrt{n})$ multiplications (and a possibly huge number of additions). From a practical standpoint, the bounds of Theorem C apply to "almost all" coefficients, and they seem to apply to all reasonable schemes for evaluation. Furthermore it is possible to obtain lower bounds corresponding to those of Theorem C even in the rational case: By strengthening the above proofs, V. Strassen has shown, for example, that the polynomial

$$u(x) = \sum_{0 \le k \le n} 2^{2^{kn^3}} x^k \tag{30}$$

cannot be evaluated by any polynomial chain of length $< n^2/\lg n$ unless the chain has at least $\frac{1}{2}n - 2$ multiplications and $n - 4$ additions [*SIAM J. Computing* **3** (1974), 128–149]. The coefficients of (30) are very large; but it is also possible to find polynomials whose coefficients are just 0's and 1's, such that every polynomial chain computing them involves at least $\sqrt{n}/(4 \lg n)$ chain multiplications, for all sufficiently large n, even when the parameters α_j are allowed to be arbitrary complex numbers. [See R. J. Lipton, *SIAM J. Computing* **7** (1978), 61–69; C.-P. Schnorr, *Lecture Notes in Comp. Sci.* **53** (1977), 135–147.] Jean-Paul Van de Wiele has shown that the evaluation of certain 0–1 polynomials requires a total of at least $cn/\log n$ arithmetic operations, for some $c > 0$ [*Proc. IEEE Symp. Foundations of Comp. Sci.* **19** (1978), 159–165].

A gap still remains between the lower bounds of Theorem C and the actual operation counts known to be achievable, except in the trivial case $n = 2$. Theorem E gives $\lfloor n/2 \rfloor + 2$ multiplications, not $\lfloor n/2 \rfloor + 1$, although it does achieve the minimum number of additions. Our special methods for $n = 4$ and $n = 6$ have the minimum number of multiplications, but one extra addition. When n is odd, it is not difficult to prove that the lower bounds of Theorem C cannot be achieved simultaneously for both multiplications and additions; see exercise 33. For $n = 3$, 5, and 7, it is possible to show that at least $\lfloor n/2 \rfloor + 2$ multiplications are necessary. Exercises 35 and 36 show that the lower bounds of Theorem C cannot both be achieved when $n = 4$ or $n = 6$; thus the methods we have discussed are best possible, for $n < 8$. When n is even, Motzkin proved that $\lfloor n/2 \rfloor + 1$ multiplications are sufficient, but his construction involves an indeterminate number of additions (see exercise 39). An optimal scheme for $n = 8$ was found by V. Ia. Pan, who showed that $n + 1$ additions are necessary and sufficient for this case when there are $\lfloor n/2 \rfloor + 1$ multiplications; he also showed that $\lfloor n/2 \rfloor + 1$ multiplications and $n + 2$ additions will suffice for all even $n \ge 10$. Pan's paper [*Proc. ACM Symp. Theory of Computing* **10** (1978), 162–172] also establishes the exact minimum number of multiplications and additions needed when calculations are done entirely with complex numbers instead of

reals, for all degrees n. Exercise 40 discusses the interesting situation that arises for odd values of $n \geq 9$.

It is clear that the results we have obtained about chains for polynomials in a single variable can be extended without difficulty to multivariate polynomials. For example, if we want to find an optimum scheme for polynomial evaluation *without* adaptation of coefficients, we can regard $u(x)$ as a polynomial in the $n+2$ variables $x, u_n, \ldots, u_1, u_0$; exercise 38 shows that n multiplications and n additions are necessary in this case. Indeed, A. Borodin [*Theory of Machines and Computations*, ed. by Z. Kohavi and A. Paz (New York: Academic Press, 1971), 45–58] has proved that Horner's rule (2) is essentially the *only* way to compute $u(x)$ in $2n$ operations without preconditioning.

With minor variations, the above methods can be extended to chains involving division, i.e., to rational functions as well as polynomials. Curiously, the continued-fraction analog of Horner's rule now turns out to be optimal from an operation-count standpoint, if multiplication and division speeds are equal, even when preconditioning is allowed (see exercise 37).

Sometimes division is helpful during the evaluation of polynomials, even though polynomials are defined only in terms of multiplication and addition; we have seen examples of this in the Shaw–Traub algorithms for polynomial derivatives. Another example is the polynomial

$$x^n + \cdots + x + 1;$$

since this polynomial can be written $(x^{n+1}-1)/(x-1)$, we can evaluate it with $l(n+1)$ multiplications (see Section 4.6.3), two subtractions, and one division, while techniques that avoid division seem to require about three times as many operations (see exercise 43).

Special multivariate polynomials. The *determinant* of an $n \times n$ matrix may be considered to be a polynomial in n^2 variables x_{ij}, $1 \leq i, j \leq n$. If $x_{11} \neq 0$, we have

$$\det \begin{pmatrix} x_{11} & x_{12} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ x_{31} & x_{32} & \ldots & x_{3n} \\ \vdots & \vdots & & \vdots \\ x_{n1} & x_{n2} & \ldots & x_{nn} \end{pmatrix} = x_{11} \det \begin{pmatrix} x_{22}-(x_{21}/x_{11})x_{12} & \ldots & x_{2n}-(x_{21}/x_{11})x_{1n} \\ x_{32}-(x_{31}/x_{11})x_{12} & \ldots & x_{3n}-(x_{31}/x_{11})x_{1n} \\ \vdots & & \vdots \\ x_{n2}-(x_{n1}/x_{11})x_{12} & \ldots & x_{nn}-(x_{n1}/x_{11})x_{1n} \end{pmatrix}. \tag{31}$$

The determinant of an $n \times n$ matrix may therefore be evaluated by evaluating the determinant of an $(n-1) \times (n-1)$ matrix and performing an additional $(n-1)^2+1$ multiplications, $(n-1)^2$ additions, and $n-1$ divisions. Since a 2×2 determinant can be evaluated with two multiplications and one addition, we see that the determinant of almost all matrices (namely those for which no division by zero is needed) can be computed with at most $(2n^3-3n^2+7n-6)/6$ multiplications, $(2n^3-3n^2+n)/6$ additions, and $(n^2-n-2)/2$ divisions.

When zero occurs, the determinant is even easier to compute. For example, if $x_{11} = 0$ but $x_{21} \neq 0$, we have

$$\det\begin{pmatrix} 0 & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ x_{31} & x_{32} & \dots & x_{3n} \\ \vdots & \vdots & & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nn} \end{pmatrix} = -x_{21} \det\begin{pmatrix} x_{12} & \dots & x_{1n} \\ x_{32}-(x_{31}/x_{21})x_{22} & \dots & x_{3n}-(x_{31}/x_{21})x_{2n} \\ \vdots & & \vdots \\ x_{n2}-(x_{n1}/x_{21})x_{22} & \dots & x_{nn}-(x_{n1}/x_{21})x_{2n} \end{pmatrix}. \tag{32}$$

Here the reduction to an $(n-1)\times(n-1)$ determinant saves $n-1$ of the multiplications and $n-1$ of the additions used in (31), and this certainly compensates for the additional bookkeeping required to recognize this case. Therefore any determinant can be evaluated with roughly $\frac{2}{3}n^3$ arithmetic operations (including division); this is remarkable, since it is a polynomial with $n!$ terms and n variables in each term.

If we want to evaluate the determinant of a matrix with *integer* elements, the above process appears to be unattractive since it requires rational arithmetic. However, we can use the method to evaluate the determinant mod p, for any prime p, since division mod p is possible (exercise 4.5.2–15). If this is done for sufficiently many primes p, the exact value of the determinant can be found as explained in Section 4.3.2, since Hadamard's inequality (4.6.1–25) gives an upper bound on the magnitude.

The coefficients of the *characteristic polynomial* $\det(xI - X)$ of an $n \times n$ matrix X can also be computed in $O(n^3)$ steps; cf. J. H. Wilkinson, *The Algebraic Eigenvalue Problem* (Oxford: Clarendon Press, 1965), 353–355, 410–411.

The *permanent* of a matrix is a polynomial that is very similar to the determinant; the only difference is that all of its nonzero coefficients are $+1$. Thus we have

$$\operatorname{per}\begin{pmatrix} x_{11} & \dots & x_{1n} \\ \vdots & & \vdots \\ x_{n1} & \dots & x_{nn} \end{pmatrix} = \sum x_{1j_1}x_{2j_2}\dots x_{nj_n}, \tag{33}$$

summed over all permutations $j_1\, j_2 \dots j_n$ of $\{1, 2, \dots, n\}$. It would seem that this function should be even easier to compute than its more complicated-looking cousin, but no way to evaluate the permanent as efficiently as the determinant is known. Exercises 9 and 10 show that substantially fewer than $n!$ operations will suffice, for large n, but the execution time of all known methods still grows exponentially with the size of the matrix. In fact, Leslie G. Valiant has shown that it is as difficult to compute the permanent of a given 0–1 matrix as it is to count the number of accepting computations of a nondeterministic polynomial-time Turing machine, if we ignore polynomial factors in the running time of the calculation. Therefore a polynomial-time evaluation algorithm for permanents would imply that scores of other well known problems that have resisted efficient solution would be solvable in polynomial time. On the other hand, Valiant proved that the permanent of an $n \times n$ integer matrix can be evaluated modulo 2^k in $O(n^{4k-3})$ steps for all $k \geq 2$. [See *Theoretical Comp. Sci.* 8 (1979), 189–201.]

Another fundamental operation involving matrices is, of course, *matrix multiplication:* If $X = (x_{ij})$ is an $m \times n$ matrix, $Y = (y_{jk})$ is an $n \times s$ matrix, and $Z = (z_{ik})$ is an $m \times s$ matrix, then the formula $Z = XY$ means that

$$z_{ik} = \sum_{1 \le j \le n} x_{ij}\, y_{jk}, \qquad 1 \le i \le m, \qquad 1 \le k \le s. \tag{34}$$

This equation may be regarded as the computation of ms simultaneous polynomials in $mn+ns$ variables; each polynomial is the "inner product" of two n-place vectors. A straightforward calculation would involve mns multiplications and $ms(n-1)$ additions; but S. Winograd discovered in 1967 that there is a way to trade about half of the multiplications for additions:

$$z_{ik} = \sum_{1 \le j \le n/2} (x_{i,2j} + y_{2j-1,k})(x_{i,2j-1} + y_{2j,k}) - a_i - b_k + c_{ik};$$

$$a_i = \sum_{1 \le j \le n/2} x_{i,2j}\, x_{i,2j-1}; \qquad b_k = \sum_{1 \le j \le n/2} y_{2j-1,k}\, y_{2j,k};$$

$$c_{ik} = \begin{cases} 0, & n \text{ even;} \\ x_{in} y_{nk}, & n \text{ odd.} \end{cases} \tag{35}$$

This scheme uses $\lceil n/2 \rceil ms + \lfloor n/2 \rfloor (m+s)$ multiplications and $(n+2)ms + (\lfloor n/2 \rfloor - 1)(ms + m + s)$ additions or subtractions; the total number of operations has increased slightly, but the number of multiplications has roughly been halved. [See *IEEE Trans.* **C–17** (1968), 693–694.] Winograd's surprising construction led many people to look more closely at the problem of matrix multiplication, and it touched off widespread speculation that $n^3/2$ multiplications would be necessary to multiply $n \times n$ matrices, because of the somewhat similar lower bound that was known to hold for polynomials in one variable.

An even better scheme for large n was discovered by Volker Strassen in 1968; he found a way to compute the product of 2×2 matrices with only seven multiplications, without relying on the commutativity of multiplication as in (35). Since $2n \times 2n$ matrices can be partitioned into four $n \times n$ matrices, his idea can be used recursively to obtain the product of $2^k \times 2^k$ matrices with only 7^k multiplications instead of $(2^k)^3 = 8^k$. The number of additions also grows as order 7^k. Strassen's original 2×2 identity [*Numer. Math.* **13** (1969), 354–356] used 7 multiplications and 18 additions; S. Winograd later discovered the following more economical formula:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} A & C \\ B & D \end{pmatrix} =$$

$$\begin{pmatrix} aA + bB & w + (c+d)(C-A) + (a+b-c-d)D \\ w + (a-c)(D-C) - d(A-B-C+D) & w + (a-c)(D-C) + (c+d)(C-A) \end{pmatrix},$$

$$\text{where } w = aA - (a-c-d)(A-C+D). \tag{36}$$

If intermediate results are appropriately saved, (36) involves 7 multiplications and only 15 additions; by induction on k, we can multiply $2^k \times 2^k$ matrices with 7^k multiplications and $5(7^k - 4^k)$ additions. The total number of operations needed to multiply $n \times n$ matrices has therefore been reduced from order n^3 to $O(n^{\lg 7}) = O(n^{2.8074})$. A similar reduction applies also to the evaluation of determinants and matrix inverses; cf. J. R. Bunch and J. E. Hopcroft, *Math. Comp.* **28** (1974), 231–236.

Strassen's exponent $\lg 7$ resisted numerous attempts at improvement until 1978, when Viktor Pan discovered that it could be lowered to $\log_{70} 143640 \approx 2.795$ (see exercise 60). This new breakthrough led to further intensive analysis of the problem, and the combined efforts of D. Bini, M. Capovani, G. Lotti, F. Romani, A. Schönhage, V. Pan, S. Winograd, and D. Coppersmith culminated in constructions that have an asymptotic running time of $O(n^{2.5161})$. Exercises 60–64 discuss some of the interesting techniques that were used to derive this bound; a full account of the developments has been prepared by V. Îa. Pan, *Computers and Math.*, to appear.

These theoretical results are quite striking, but from a practical standpoint they are of limited use because n must be very large before we overcome the effect of additional bookkeeping costs. Richard Brent [Stanford Computer Science report CS157 (March, 1970), see also *Numer. Math.* **16** (1970), 145–156] found that a careful implementation of Winograd's scheme (35), with appropriate scaling for numerical stability, became better than the conventional method only when $n \geq 40$, and it saved only about 7 percent of the running time when $n = 100$. For complex arithmetic the situation was somewhat different; (35) became advantageous for $n > 20$, and saved 18 percent when $n = 100$. He estimated that Strassen's scheme would not begin to excel over (35) until $n \approx 250$; and such enormous matrices, containing more than 60,000 entries, rarely occur in practice (unless they are very sparse, when other techniques apply). Furthermore, the known methods of order n^β where $\beta < 2.7$ have such large constants of proportionality that they require more than 10^{23} multiplications before they start to beat Strassen's scheme.

By contrast, the methods we shall discuss next are eminently practical and have found wide use. The *discrete Fourier transform* f of a complex-valued function F of n variables, over respective domains of $m_1, \ldots, m_n$ elements, is defined by the equation

$$f(s_1, \ldots, s_n) = \sum_{\substack{0 \le t_1 < m_1 \\ \cdots \\ 0 \le t_n < m_n}} \exp\left(2\pi i \left(\frac{s_1 t_1}{m_1} + \cdots + \frac{s_n t_n}{m_n}\right)\right) F(t_1, \ldots, t_n) \qquad (37)$$

for $0 \le s_1 < m_1, \ldots, 0 \le s_n < m_n$; the name "transform" is justified because we can recover the values $F(t_1, \ldots, t_n)$ from the values $f(s_1, \ldots, s_n)$, as shown in exercise 13. In the important special case that all $m_j = 2$, we have

$$f(s_1, \ldots, s_n) = \sum_{0 \le t_1, \ldots, t_n \le 1} (-1)^{s_1 t_1 + \cdots + s_n t_n} F(t_1, \ldots, t_n) \qquad (38)$$

for $0 \le s_1, \ldots, s_n \le 1$, and this may be regarded as a simultaneous evaluation of 2^n linear polynomials in 2^n variables $F(t_1, \ldots, t_n)$. A well-known technique due to F. Yates [*The Design and Analysis of Factorial Experiments* (Harpenden: Imperial Bureau of Soil Sciences, 1937)] can be used to reduce the number of additions implied in (38) from $2^n(2^n - 1)$ to $n2^n$. Yates's method can be understood by considering the case $n = 3$: Let $x_{t_1t_2t_3} = F(t_1, t_2, t_3)$.

Given	First step	Second step	Third step
x_{000}	$x_{000}+x_{001}$	$x_{000}+x_{001}+x_{010}+x_{011}$	$x_{000}+x_{001}+x_{010}+x_{011}+x_{100}+x_{101}+x_{110}+x_{111}$
x_{001}	$x_{010}+x_{011}$	$x_{100}+x_{101}+x_{110}+x_{111}$	$x_{000}-x_{001}+x_{010}-x_{011}+x_{100}-x_{101}+x_{110}-x_{111}$
x_{010}	$x_{100}+x_{101}$	$x_{000}-x_{001}+x_{010}-x_{011}$	$x_{000}+x_{001}-x_{010}-x_{011}+x_{100}+x_{101}-x_{110}-x_{111}$
x_{011}	$x_{110}+x_{111}$	$x_{100}-x_{101}+x_{110}-x_{111}$	$x_{000}-x_{001}-x_{010}+x_{011}+x_{100}-x_{101}-x_{110}+x_{111}$
x_{100}	$x_{000}-x_{001}$	$x_{000}+x_{001}-x_{010}-x_{011}$	$x_{000}+x_{001}+x_{010}+x_{011}-x_{100}-x_{101}-x_{110}-x_{111}$
x_{101}	$x_{010}-x_{011}$	$x_{100}+x_{101}-x_{110}-x_{111}$	$x_{000}-x_{001}+x_{010}-x_{011}-x_{100}+x_{101}-x_{110}+x_{111}$
x_{110}	$x_{100}-x_{101}$	$x_{000}-x_{001}-x_{010}+x_{011}$	$x_{000}+x_{001}-x_{010}-x_{011}-x_{100}-x_{101}+x_{110}+x_{111}$
x_{111}	$x_{110}-x_{111}$	$x_{100}-x_{101}-x_{110}+x_{111}$	$x_{000}-x_{001}-x_{010}+x_{011}-x_{100}+x_{101}+x_{110}-x_{111}$

To get from the "Given" to the "First step" requires four additions and four subtractions; and the interesting feature of Yates's method is that exactly the same transformation that takes us from "Given" to "First step" will take us from "First step" to "Second step" and from "Second step" to "Third step." In each case we do four additions, then four subtractions; and after three steps we magically have the desired Fourier transform $f(s_1, s_2, s_3)$ in the place originally occupied by $F(s_1, s_2, s_3)$.

This special case is often called the *Walsh transform* of 2^n data elements, since the corresponding pattern of signs was studied by J. L. Walsh [*Amer. J. Math.* **45** (1923), 5–24]. Note that the number of sign changes from left to right in the "Third step" above assumes the respective values

$$0,\ 7,\ 3,\ 4,\ 1,\ 6,\ 2,\ 5;$$

this is a permutation of the numbers $\{0, 1, 2, 3, 4, 5, 6, 7\}$. Walsh observed that there will be exactly 0, 1, ..., $2^n - 1$ sign changes in the general case, if we permute the transformed elements appropriately, so the coefficients provide discrete approximations to sine waves with various frequencies. (See H. F. Harmuth, *IEEE Spectrum* **6**, 11 (Nov. 1969), 82–91, for applications of this property; and see Section 7.2.1 for further discussion of the Walsh coefficients.)

Yates's method can be generalized to the evaluation of any discrete Fourier transform, and, in fact, to the evaluation of any set of sums that can be written in the general form

$$f(s_1, s_2, \ldots, s_n) = \sum_{\substack{0 \le t_1 < m_1 \\ \cdots \\ 0 \le t_n < m_n}} g_1(s_1, s_2, \ldots, s_n, t_1) g_2(s_2, \ldots, s_n, t_2) \ldots g_n(s_n, t_n) F(t_1, t_2, \ldots, t_n) \quad (39)$$

for $0 \le s_j < m_j$, given the functions $g_j(s_j, \ldots, s_n, t_j)$. We proceed as follows.

$$f^{[0]}(t_1, t_2, t_3, \ldots, t_n) = F(t_1, t_2, t_3, \ldots, t_n);$$

$$f^{[1]}(s_n, t_1, t_2, \ldots, t_{n-1}) = \sum_{0 \le t_n < m_n} g_n(s_n, t_n) f^{[0]}(t_1, t_2, \ldots, t_n);$$

$$f^{[2]}(s_{n-1}, s_n, t_1, \ldots, t_{n-2}) = \sum_{0 \le t_{n-1} < m_{n-1}} g_{n-1}(s_{n-1}, s_n, t_{n-1}) f^{[1]}(s_n, t_1, \ldots, t_{n-1});$$

$$\vdots$$

$$f^{[n]}(s_1, s_2, s_3, \ldots, s_n) = \sum_{0 \le t_1 < m_1} g_1(s_1, \ldots, s_n, t_1) f^{[n-1]}(s_2, s_3, \ldots, s_n, t_1);$$

$$f(s_1, s_2, s_3, \ldots, s_n) = f^{[n]}(s_1, s_2, s_3, \ldots, s_n). \tag{40}$$

For Yates's method as shown above, $g_j(s_j, \ldots, s_n, t_j) = (-1)^{s_j t_j}$; $f^{[0]}(t_1, t_2, t_3)$ represents the "Given"; $f^{[1]}(s_3, t_1, t_2)$ represents the "First step"; etc. Whenever a desired set of sums can be put into the form of (39), for reasonably simple functions $g_j(s_j, \ldots, s_n, t_j)$, the scheme (40) will reduce the amount of computation from order N^2 to order $N \log N$ or thereabouts, where $N = m_1 \ldots m_n$ is the number of data points; furthermore this scheme is ideally suited to parallel computation. The important special case of one-dimensional Fourier transforms is discussed in exercises **14** and **53**; we have considered the one-dimensional case also in Section 4.3.3.

Let us consider one more special case of polynomial evaluation. *Lagrange's interpolation polynomial* of order n, which we shall write as

$$\begin{aligned} u^{[n]}(x) = {} & y_0 \frac{(x - x_1)(x - x_2) \ldots (x - x_n)}{(x_0 - x_1)(x_0 - x_2) \ldots (x_0 - x_n)} \\ & + y_1 \frac{(x - x_0)(x - x_2) \ldots (x - x_n)}{(x_1 - x_0)(x_1 - x_2) \ldots (x_1 - x_n)} \\ & + \cdots + y_n \frac{(x - x_0)(x - x_1) \ldots (x - x_{n-1})}{(x_n - x_0)(x_n - x_1) \ldots (x_n - x_{n-1})}, \end{aligned} \tag{41}$$

is the only polynomial of degree $\le n$ in x that takes on the respective values $y_0, y_1, \ldots, y_n$ at the $n + 1$ distinct points $x = x_0, x_1, \ldots, x_n$. (For it is evident from (41) that $u^{[n]}(x_k) = y_k$ for $0 \le k \le n$. If $f(x)$ is any such polynomial of degree $\le n$, then $g(x) = f(x) - u^{[n]}(x)$ is of degree $\le n$, and $g(x)$ is zero for $x = x_0, x_1, \ldots, x_n$; therefore $g(x)$ is a multiple of the polynomial $(x - x_0)(x - x_1) \ldots (x - x_n)$. The degree of the latter polynomial is greater than n, so $g(x) = 0$.) If we assume that the values of a function in some table are well approximated by a polynomial, Lagrange's formula (41) may therefore be used to "interpolate" for values of the function at points x not appearing in the table. Unfortunately, there seem to be quite a few additions, subtractions, multiplications, and divisions in Lagrange's formula; in fact, there are exactly n additions, $2n^2+2$ subtractions, $2n^2+n-1$ multiplications, and $n+1$ divisions.

But fortunately (as we might be conditioned to suspect by now), improvement is possible.

The basic idea for simplifying (41) is to note that $u^{[n]}(x) - u^{[n-1]}(x)$ is zero for $x = x_0, \ldots, x_{n-1}$; thus $u^{[n]}(x) - u^{[n-1]}(x)$ is a polynomial of degree n or less, and a multiple of $(x - x_0)\ldots(x - x_{n-1})$. We conclude that $u^{[n]}(x) = \alpha_n(x - x_0)\ldots(x - x_{n-1}) + u^{[n-1]}(x)$, where α_n is a constant. This leads us to *Newton's interpolation formula*

$$u^{[n]}(x) = \alpha_n(x - x_0)(x - x_1)\ldots(x - x_{n-1}) + \cdots + \alpha_2(x - x_0)(x - x_1) + \alpha_1(x - x_0) + \alpha_0, \tag{42}$$

where the α's are some constants we should like to determine from x_0, x_1, $\ldots$, x_n, y_0, y_1, $\ldots$, y_n. Note that this formula holds for all n; the coefficient α_k does not depend on x_{k+1}, $\ldots$, x_n, or y_{k+1}, $\ldots$, y_n. Once the α's are known, Newton's interpolation formula is convenient for calculation, since we may generalize Horner's rule once again and write

$$u^{[n]}(x) = ((\ldots(\alpha_n(x - x_{n-1}) + \alpha_{n-1})(x - x_{n-2}) + \cdots)(x - x_0) + \alpha_0). \tag{43}$$

This requires n multiplications and $2n$ additions. Alternatively, we may evaluate each of the individual terms of (42) from right to left; with $2n - 1$ multiplications and $2n$ additions we thereby calculate all of the values $u^{[0]}(x)$, $u^{[1]}(x)$, $\ldots$, $u^{[n]}(x)$, and this indicates whether or not an interpolation process is "converging."

The coefficients α_k in Newton's formula may be found by computing the *divided differences* in the following tableau (shown for $n = 3$):

$$\begin{array}{lllll} y_0 & & & & \\ & (y_1 - y_0)/(x_1 - x_0) = y_1' & & & \\ y_1 & & (y_2' - y_1')/(x_2 - x_0) = y_2'' & & \\ & (y_2 - y_1)/(x_2 - x_1) = y_2' & & (y_3'' - y_2'')/(x_3 - x_0) = y_3''' & \\ y_2 & & (y_3' - y_2')/(x_3 - x_1) = y_3'' & & \\ & (y_3 - y_2)/(x_3 - x_2) = y_3' & & & \\ y_3 & & & & \end{array} \tag{44}$$

It is possible to prove that $\alpha_0 = y_0$, $\alpha_1 = y_1'$, $\alpha_2 = y_2''$, etc., and to show that the divided differences have important relations to the derivatives of the function being interpolated; see exercise 15. Therefore the following calculation (corresponding to (44)) may be used to obtain the α's:

> Start with $(\alpha_0, \alpha_1, \ldots, \alpha_n) \leftarrow (y_0, y_1, \ldots, y_n)$; then, for $k = 1, 2, \ldots, n$ (in this order), set $\alpha_j \leftarrow (\alpha_j - \alpha_{j-1})/(x_j - x_{j-k})$ for $j = n, n - 1, \ldots, k$ (in this order).

This process requires $\frac{1}{2}(n^2 + n)$ divisions and $n^2 + n$ subtractions, so about three-fourths of the work implied in (41) has been saved.

For example, suppose that we want to estimate $\frac{3}{2}!$ from the values of 0!, 1!, 2!, and 3!, using a cubic polynomial. The divided differences are

x	y	y'	y''	y'''
0	1			
		0		
1	1		$\frac{1}{2}$	
		1		$\frac{1}{3}$
2	2		$\frac{3}{2}$	
		4		
3	6			

so $u^{[0]}(x) = u^{[1]}(x) = 1$, $u^{[2]}(x) = \frac{1}{2}x(x-1)+1$, $u^{[3]}(x) = \frac{1}{3}x(x-1)(x-2)+\frac{1}{2}x(x-1)+1$. Setting $x = \frac{3}{2}$ in the latter polynomial gives $-\frac{1}{8}+\frac{3}{8}+1 = 1.25$; presumably the "correct" value is $\Gamma(\frac{3}{2}+1) = \frac{3}{4}\sqrt{\pi} \approx 1.33$.

An important and somewhat surprising application of polynomial interpolation was discovered by Adi Shamir [*CACM* **22** (1979), 612–613], who observed that polynomials mod p can be used to "share a secret." This means that we can design a system of secret keys or passwords such that the knowledge of any $n+1$ of the keys enables efficient calculation of a magic number N that unlocks a door (say), but the knowledge of any n of the keys gives no information whatsoever about N. Shamir's amazingly simple solution to this problem is to choose a random polynomial $u(x) = u_n x^n + \cdots + u_1 x + u_0$, where $0 \le u_i < p$ and p is a large prime number. Each part of the secret is an integer x in the range $0 < x < p$, together with the value of $u(x) \bmod p$; and the supersecret number N is the constant term u_0. Given $n+1$ values $u(x_i)$, we can deduce N by interpolation. But if only n values of $u(x_i)$ are given, there is a unique polynomial $u(x)$ having a given constant term but the same values at $x_1, \ldots, x_n$; thus the n values do not make one particular N more likely than any other.

It is instructive to note that evaluation of the interpolation polynomial is just a special case of the Chinese remainder algorithm of Section 4.3.2 and exercise 4.6.2–3, since we know the values of $u^{[n]}(x)$ modulo the relatively prime polynomials $x - x_0, \ldots, x - x_n$. (As we have seen in Section 4.6.2, $f(x) \bmod (x - x_0) = f(x_0)$.) Under this interpretation, Newton's formula (42) is precisely the "mixed-radix representation" of Eq. 4.3.2–24; and 4.3.2–23 yields another way to compute $\alpha_0, \ldots, \alpha_n$ using the same number of operations as (44).

By applying fast Fourier transforms, it is possible to reduce the running time for interpolation to $O(n(\log n)^2)$, and a similar reduction can also be made for related algorithms such as the solution to the Chinese remainder problem and the evaluation of an nth degree polynomial at n different points. [See E. Horowitz, *Inf. Proc. Letters* **1** (1972), 157–163; R. Moenck and A. Borodin, *J. Comp. Syst. Sci.* **8** (1974), 336–385; and A. Borodin, *Complexity of Sequential and Parallel Numerical Algorithms*, ed. by J. F. Traub (New York: Academic Press, 1973), 149–180.] However, this must be regarded as a purely theoretical possibility at present, since the known algorithms have a rather large overhead factor that makes them unattractive unless n is quite large.

A remarkable modification of the method of divided differences, an extension that applies to rational functions instead of to polynomials, was introduced by T. N. Thiele in 1909. Thiele's method of "reciprocal differences" is discussed in L. M. Milne-Thompson's *Calculus of Finite Differences* (London: MacMillan, 1933), Chapter 5; see also R. W. Floyd, *CACM* **3** (1960), 508.

***Bilinear forms.** Several of the problems we have considered in this section are special cases of the general problem of evaluating a set of *bilinear forms*

$$z_k = \sum_{\substack{1\le i\le m\\ 1\le j\le n}} t_{ijk}x_iy_j, \qquad \text{for } 1 \le k \le s, \tag{45}$$

where the t_{ijk} are specific coefficients belonging to some given field. The three-dimensional array (t_{ijk}) is called an $m \times n \times s$ *tensor*, and we can display it by writing down s matrices of size $m \times n$, one for each value of k. For example, the problem of multiplying complex numbers, namely the problem of evaluating

$$z_1 + iz_2 = (x_1 + ix_2)(y_1 + iy_2) = (x_1y_1 - x_2y_2) + i(x_1y_2 + x_2y_1), \tag{46}$$

is the problem of computing the bilinear form specified by the $2 \times 2 \times 2$ tensor

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Matrix multiplication as defined in (34) is the problem of evaluating a set of bilinear forms corresponding to a particular $mn \times ns \times ms$ tensor. Fourier transforms (37) can also be cast in this mold, although they are linear instead of bilinear, if we let the x's be constant rather than variable.

The evaluation of bilinear forms is most easily studied if we restrict ourselves to what might be called *normal* evaluation schemes, in which all chain multiplications take place between a linear combination of the x's and a linear combination of the y's. Thus, we form r products

$$w_l = (a_{1l}x_1 + \cdots + a_{ml}x_m)(b_{1l}y_1 + \cdots + b_{nl}y_n), \qquad \text{for } 1 \le l \le r, \tag{47}$$

and obtain the z's as linear combinations of these products,

$$z_k = c_{k1}w_1 + \cdots + c_{kr}w_r, \qquad \text{for } 1 \le k \le s. \tag{48}$$

Here all the a's, b's, and c's belong to a given field of coefficients. By comparing (48) to (45), we see that a normal evaluation scheme is correct for the tensor (t_{ijk}) if and only if

$$t_{ijk} = a_{i1}b_{j1}c_{k1} + \cdots + a_{ir}b_{jr}c_{kr} \tag{49}$$

for $1 \le i \le m$, $1 \le j \le n$, and $1 \le k \le s$.

A nonzero tensor (t_{ijk}) is said to be of rank one if there are three vectors $(a_1, \ldots, a_m)$, $(b_1, \ldots, b_n)$, $(c_1, \ldots, c_s)$ such that $t_{ijk} = a_i b_j c_k$ for all i, j, k. We can extend this definition to all tensors by saying that *the rank of* (t_{ijk}) *is the minimum number* r *such that* (t_{ijk}) *is expressible as the sum of* r *rank-one tensors* in the given field. Comparing this definition with Eq. (49) shows that the rank of a tensor is the minimum number of chain multiplications in a normal evaluation of the corresponding bilinear forms. Incidentally, when $s = 1$ the tensor (t_{ijk}) is just an ordinary matrix, and the rank of (t_{ij1}) as a tensor is the same as its rank as a matrix (see exercise 49). The concept of tensor rank was introduced by F. L. Hitchcock in *J. Math. and Physics* **6** (1927), 164–189; its application to the complexity of polynomial evaluation was pointed out in an important paper by V. Strassen, *J. für die reine und angew. Math.* **264** (1973), 184–202.

Winograd's scheme (35) for matrix multiplication is "abnormal" because it mixes x's and y's before multiplying them. The Strassen-Winograd scheme (36), on the other hand, does not rely on the commutativity of multiplication, so it is normal. In fact, (36) corresponds to the following way to represent the $4 \times 4 \times 4$ tensor for 2×2 matrix multiplication as a sum of seven rank-one tensors:

$$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\1&0&0&0\\0&1&0&0\end{pmatrix}\begin{pmatrix}0&0&1&0\\0&0&0&1\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix} = \begin{pmatrix}1&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}1&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}1&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}1&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}$$
$$+\begin{pmatrix}0&0&0&0\\0&1&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}+\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&\bar1&\bar1\\0&0&0&0\\0&0&1&1\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&\bar1&\bar1\\0&0&0&0\\0&0&1&1\\0&0&0&0\end{pmatrix}$$
$$+\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\1&1&\bar1&\bar1\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}+\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\\bar1&0&1&0\\1&0&\bar1&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\\bar1&0&1&0\\1&0&\bar1&0\end{pmatrix}$$
$$+\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}0&0&0&\bar1\\0&0&0&1\\0&0&0&1\\0&0&0&\bar1\end{pmatrix}\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}+\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\begin{pmatrix}\bar1&0&1&1\\0&0&0&0\\1&0&\bar1&\bar1\\\bar1&0&1&1\end{pmatrix}\begin{pmatrix}\bar1&0&1&1\\0&0&0&0\\1&0&\bar1&\bar1\\\bar1&0&1&1\end{pmatrix}\begin{pmatrix}\bar1&0&1&1\\0&0&0&0\\1&0&\bar1&\bar1\\\bar1&0&1&1\end{pmatrix}. \tag{50}$$

(Here $\bar1$ stands for -1.)

The fact that (49) is symmetric in i, j, k and invariant under a variety of transformations makes the study of tensor rank mathematically tractable, and it also leads to some surprising consequences about bilinear forms. We can permute the indices i, j, k to obtain "transposed" bilinear forms, and the transposed tensor clearly has the same rank; but the corresponding bilinear forms are conceptually quite different. For example, a normal scheme for evaluating an $(m \times n)$ times $(n \times s)$ matrix product implies the existence of a normal scheme to evaluate an $(n \times s)$ times $(s \times m)$ matrix product, using the same number of chain multiplications. In matrix terms these two problems hardly seem to be related at all—they involve different numbers of dot products on vectors of different sizes—but in tensor terms they are equivalent. [Cf. V. Ía. Pan, *Uspekhi Mat.*

Nauk **27**,5 (1972), 249–250; J. E. Hopcroft and J. Musinski, *SIAM J. Computing* **2** (1973), 159–173.]

When the tensor (t_{ijk}) can be represented as a sum (49) of r rank-one tensors, let A, B, C be the matrices (a_{il}), (b_{jl}), (c_{kl}) of respective sizes $m \times r$, $n \times r$, $s \times r$; we shall say that A, B, C is a *realization* of the tensor (t_{ijk}). For example, the realization of 2×2 matrix multiplication in (50) can be specified by the matrices

$$A = \begin{pmatrix} 1 & 0 & \bar{1} & 0 & 0 & \bar{1} & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & \bar{1} & 1 & \bar{1} \\ 0 & 0 & 0 & 1 & 1 & \bar{1} & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & \bar{1} \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & \bar{1} & \bar{1} & 0 & 1 \\ 0 & 0 & 1 & \bar{1} & 0 & 1 & 1 \end{pmatrix}, \quad C = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}. \tag{51}$$

An $m \times n \times s$ tensor (t_{ijk}) can also be represented as a matrix by grouping its subscripts together. We shall write $(t_{(ij)k})$ for the $mn \times s$ matrix whose rows are indexed by the pair of subscripts $\langle i, j \rangle$ and whose columns are indexed by k. Similarly, $(t_{k(ij)})$ stands for the $s \times mn$ matrix that contains t_{ijk} in row k and column $\langle i, j \rangle$; $(t_{(ik)j})$ is an $ms \times n$ matrix, and so on. The indices of an array need not be integers, and we are using ordered pairs as indices here. We can use this notation to derive the following simple but useful lower bound on the rank of a tensor.

Lemma T. *Let A, B, C be a realization of an $m \times n \times s$ tensor (t_{ijk}). Then* $\operatorname{rank}(A) \geq \operatorname{rank}(t_{i(jk)})$, $\operatorname{rank}(B) \geq \operatorname{rank}(t_{j(ik)})$, *and* $\operatorname{rank}(C) \geq \operatorname{rank}(t_{k(ij)})$; *consequently*

$$\operatorname{rank}(t_{ijk}) \geq \max\bigl(\operatorname{rank}(t_{i(jk)}), \operatorname{rank}(t_{j(ik)}), \operatorname{rank}(t_{k(ij)})\bigr).$$

Proof. It suffices by symmetry to show that $r \geq \operatorname{rank}(A) \geq \operatorname{rank}(t_{i(jk)})$. Since A is an $m \times r$ matrix, it is obvious that A cannot have rank greater than r. Furthermore, according to (49), the matrix $(t_{i(jk)})$ is equal to AQ, where Q is the $r \times ns$ matrix defined by $Q_{l\langle j,k\rangle} = b_{jl}\, c_{kl}$. If x is any row vector such that $xA = 0$ then $xAQ = 0$, hence all linear dependencies in A occur also in AQ. It follows that $\operatorname{rank}(AQ) \leq \operatorname{rank}(A)$. ∎

As an example of the use of Lemma T, let us consider the problem of polynomial multiplication. Suppose we want to multiply a general polynomial of degree 2 by a general polynomial of degree 3, obtaining the coefficients of the product:

$$\begin{aligned}(x_0 + x_1 u + x_2 u^2)(y_0 + y_1 u + y_2 u^2 + y_3 u^3) \\ = z_0 + z_1 u + z_2 u^2 + z_3 u^3 + z_4 u^4 + z_5 u^5.\end{aligned} \tag{52}$$

This is the problem of evaluating six bilinear forms corresponding to the $3 \times 4 \times 6$ tensor

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{53}$$

For brevity, we may write (52) as $x(u)y(u) = z(u)$, letting $x(u)$ denote the polynomial $x_0 + x_1u + x_2u^2$, etc. Note that we have come full circle from the way we began this section, since Eq. (1) refers to $u(x)$, not $x(u)$; the notation has changed because the *coefficients* of the polynomials are now the variables of interest to us.

If each of the six matrices in (53) is regarded as a vector of length 12 indexed by $\langle i, j\rangle$, it is clear that the vectors are linearly independent, since they are nonzero in different positions; hence the rank of (53) is at least 6 by Lemma T. Conversely, it is possible to obtain the coefficients $z_0, z_1, \ldots, z_5$ by making only six chain multiplications, for example by computing

$$x(0)y(0),\ x(1)y(1),\ \ldots,\ x(5)y(5); \tag{54}$$

this gives the values of $z(0), z(1), \ldots, z(5)$, and the formulas developed above for interpolation will yield the coefficients of $z(u)$. The evaluation of $x(j)$ and $y(j)$ can be carried out entirely in terms of additions and/or parameter multiplications, and the interpolation formula merely takes linear combinations of these values. Thus, all of the chain multiplications are shown in (54), and the rank of (53) is 6. (We used essentially this same technique when multiplying high-precision numbers in Algorithm 4.3.3C.)

The realization A, B, C of (53) sketched in the above paragraph turns out to be

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 4 & 9 & 16 & 25 \end{pmatrix},\quad \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 4 & 9 & 16 & 25 \\ 0 & 1 & 8 & 27 & 64 & 125 \end{pmatrix},\quad \begin{pmatrix} 120 & 0 & 0 & 0 & 0 & 0 \\ -274 & 600 & -600 & 400 & -150 & 24 \\ 225 & -770 & 1070 & -780 & 305 & -50 \\ -85 & 355 & -590 & 490 & -205 & 35 \\ 15 & -70 & 130 & -120 & 55 & -10 \\ -1 & 5 & -10 & 10 & -5 & 1 \end{pmatrix} \times \tfrac{1}{120}. \tag{55}$$

Thus, the scheme does indeed require the minimum number of chain multiplications, but it is completely impractical because it involves so many additions and parameter multiplications. We shall now study a practical approach to the generation of more efficient schemes, suggested by S. Winograd.

In the first place, to evaluate the coefficients of $x(u)y(u)$ when $\deg(x) = m$ and $\deg(y) = n$, one can use the identity

$$x(u)y(u) = \big(x(u)y(u) \bmod p(u)\big) + x_m y_n p(u), \tag{56}$$

when $p(u)$ is any monic polynomial of degree $m+n$. The polynomial $p(u)$ should be chosen so that the coefficients of $x(u)y(u) \bmod p(u)$ are easy to evaluate.

In the second place, to evaluate the coefficients of $x(u)y(u) \bmod p(u)$, when the polynomial $p(u)$ can be factored into $q(u)r(u)$ where $\gcd\big(q(u), r(u)\big) = 1$, one can use the identity

$$\begin{aligned} x(u)y(u) \bmod q(u)r(u) = \Big(&a(u)r(u)\big(x(u)y(u) \bmod q(u)\big) \\ &+ b(u)q(u)\big(x(u)y(u) \bmod r(u)\big)\Big) \bmod q(u)r(u) \end{aligned} \tag{57}$$

where $a(u)r(u)+b(u)q(u)=1$; this is essentially the Chinese remainder theorem applied to polynomials.

In the third place, to evaluate the coefficients of $x(u)y(u) \bmod p(u)$ when $p(u)$ has only one irreducible factor over the field of coefficients, one can use the identity

$$x(u)y(u) \bmod p(u) = \big(x(u) \bmod p(u)\big)\big(y(u) \bmod p(u)\big) \bmod p(u). \tag{58}$$

Repeated application of (56), (57), and (58) tends to produce efficient schemes, as we shall see.

For our example problem (52), let us choose $p(u)=u^5-u$ and apply (56); the reason for this choice of $p(u)$ will appear as we proceed. Writing $p(u)=u(u^4-1)$, rule (57) reduces to

$$\begin{aligned} x(u)y(u) \bmod u(u^4-1) = \big(&-(u^4-1)x_0y_0 \\ &+ u^4\big(x(u)y(u) \bmod (u^4-1)\big)\big) \bmod (u^5-u). \end{aligned} \tag{59}$$

Here we have used the fact that $x(u)y(u) \bmod u = x_0y_0$; in general it is a good idea to choose $p(u)$ in such a way that $p(0)=0$, so that this simplification can be used. If we could now determine the coefficients w_0, w_1, w_2, w_3 of the polynomial $x(u)y(u) \bmod (u^4-1) = w_0+w_1u+w_2u^2+w_3u^3$, our problem would be solved, since

$$u^4\big(x(u)y(u) \bmod (u^4-1)\big) \bmod (u^5-u) = w_0u^4+w_1u+w_2u^2+w_3u^3,$$

and the combination of (56) and (59) would reduce to

$$x(u)y(u) = x_0y_0+(w_1-x_2y_3)u+w_2u^2+w_3u^3+(w_0-x_0y_0)u^4+x_2y_3u^5. \tag{60}$$

(This formula can, of course, be verified directly.)

The problem remaining to be solved is to compute $x(u)y(u) \bmod (u^4-1)$; and this subproblem is interesting in itself. Let us momentarily allow $x(u)$ to be of degree 3 instead of degree 2. Then the coefficients of $x(u)y(u) \bmod (u^4-1)$ are respectively

$$\begin{aligned} &x_0y_0+x_1y_3+x_2y_2+x_3y_1, \quad x_0y_1+x_1y_0+x_2y_3+x_3y_2, \\ &\qquad x_0y_2+x_1y_1+x_2y_0+x_3y_3, \quad x_0y_3+x_1y_2+x_2y_1+x_3y_0, \end{aligned}$$

and the corresponding tensor is

$$\begin{pmatrix}1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0\end{pmatrix}\begin{pmatrix}0&1&0&0\\1&0&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}\begin{pmatrix}0&0&1&0\\0&1&0&0\\1&0&0&0\\0&0&0&1\end{pmatrix}\begin{pmatrix}0&0&0&1\\0&0&1&0\\0&1&0&0\\1&0&0&0\end{pmatrix}. \tag{61}$$

In general when $\deg(x)=\deg(y)=n-1$, the coefficients of $x(u)y(u) \bmod (u^n-1)$ are called the *cyclic convolution* of $(x_0,x_1,\ldots,x_{n-1})$ and $(y_0,y_1,\ldots,y_{n-1})$. The

kth coefficient w_k is the bilinear form $\sum x_i y_j$ summed over all i and j with $i+j \equiv k$ (modulo n).

The cyclic convolution of degree 4 can be obtained by applying rule (57). The first step is to find the factors of u^4-1, namely $(u-1)(u+1)(u^2+1)$. We could write this as $(u^2-1)(u^2+1)$, then apply rule (57), then use (57) again on the part modulo $(u^2-1)=(u-1)(u+1)$; but it is easier to generalize the Chinese remainder rule (57) directly to the case of several relatively prime factors. For example, we have

$$\begin{aligned} &x(u)y(u) \bmod q_1(u)q_2(u)q_3(u) \\ &\quad = \Big(a_1(u)q_2(u)q_3(u)\big(x(u)y(u) \bmod q_1(u)\big) + a_2(u)q_1(u)q_3(u)\big(x(u)y(u) \bmod q_2(u)\big) \\ &\qquad + a_3(u)q_1(u)q_2(u)\big(x(u)y(u) \bmod q_3(u)\big)\Big) \bmod q_1(u)q_2(u)q_3(u), \end{aligned} \tag{62}$$

where $a_1(u)q_2(u)q_3(u) + a_2(u)q_1(u)q_3(u) + a_3(u)q_1(u)q_2(u) = 1$. (The latter equation can be understood in another way, by noting that the partial fraction expansion of $1/q_1(u)q_2(u)q_3(u)$ is $a_1(u)/q_1(u)+a_2(u)/q_2(u)+a_3(u)/q_3(u)$. When each of the q's is a linear polynomial $u-\alpha_i$, the generalized Chinese remainder rule reduces to ordinary interpolation as in Eq. (41), since $f(u) \bmod (u-\alpha_i) = f(\alpha_i)$.) From (62) we obtain

$$\begin{aligned} x(u)y(u) \bmod (u^4-1) = \Big(&\tfrac{u^3+u^2+u+1}{4}x(1)y(1) - \tfrac{u^3-u^2+u-1}{4}x(-1)y(-1) \\ &- \tfrac{u^2-1}{2}\big(x(u)y(u) \bmod (u^2+1)\big)\Big) \bmod (u^4-1). \end{aligned} \tag{63}$$

The remaining problem is to evaluate $x(u)y(u) \bmod (u^2+1)$, and it is time to invoke rule (58). First we reduce $x(u)$ and $y(u)$ mod (u^2+1), obtaining $X(u) = (x_0 - x_2) + (x_1 - x_3)u$, $Y(u) = (y_0 - y_2) + (y_1 - y_3)u$. Then (58) tells us to evaluate $X(u)Y(u) = Z_0 + Z_1u + Z_2u^2$, and to reduce this in turn modulo (u^2+1), obtaining $(Z_0 - Z_2) + Z_1u$. The job of computing $X(u)Y(u)$ is simple; we can use rule (56) with $p(u) = u(u+1)$ and we get

$$Z_0 = X_0Y_0, \quad Z_1 = X_0Y_0 - (X_0-X_1)(Y_0-Y_1) + X_1Y_1, \quad Z_2 = X_1Y_1.$$

(We have thereby rediscovered the trick of Eq. 4.3.3–2 in a more systematic way.) Putting everything together yields the following realization A, B, C of degree-4 cyclic convolution:

$$\begin{pmatrix} 1&1&1&0&1\\ 1&\bar1&0&1&\bar1\\ 1&1&\bar1&0&\bar1\\ 1&\bar1&0&\bar1&1 \end{pmatrix}, \quad \begin{pmatrix} 1&1&1&0&1\\ 1&\bar1&0&1&\bar1\\ 1&1&\bar1&0&\bar1\\ 1&\bar1&0&\bar1&1 \end{pmatrix}, \quad \begin{pmatrix} 1&1&2&\bar2&0\\ 1&\bar1&2&2&\bar2\\ 1&1&\bar2&2&0\\ 1&\bar1&\bar2&\bar2&2 \end{pmatrix} \times \frac{1}{4}. \tag{64}$$

Here $\bar1$ stands for -1 and $\bar2$ for -2.

The tensor for cyclic convolution of degree n satisfies

$$t_{i,j,k} = t_{k,-j,i}, \tag{65}$$

treating the subscripts modulo n, since $t_{ijk} = 1$ if and only if $i + j \equiv k$ (modulo n). Thus if (a_{il}), (b_{jl}), (c_{kl}) is a realization of the cyclic convolution, so is (c_{kl}), $(b_{-j,l})$, (a_{il}); in particular, we can realize (61) by transforming (64) into

$$\begin{pmatrix} 1 & 1 & 2 & \bar{2} & 0 \\ 1 & \bar{1} & 2 & 2 & \bar{2} \\ 1 & 1 & \bar{2} & 2 & 0 \\ 1 & \bar{1} & \bar{2} & \bar{2} & 2 \end{pmatrix} \times \frac{1}{4}, \quad \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & \bar{1} & 0 & \bar{1} & 1 \\ 1 & 1 & \bar{1} & 0 & \bar{1} \\ 1 & \bar{1} & 0 & 1 & \bar{1} \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & \bar{1} & 0 & 1 & \bar{1} \\ 1 & 1 & \bar{1} & 0 & \bar{1} \\ 1 & \bar{1} & 0 & \bar{1} & 1 \end{pmatrix}. \tag{66}$$

Now all of the complicated scalars appear in the A matrix. This is important in practice, since we often want to compute the convolution for many values of y_0, y_1, y_2, y_3 but for a fixed choice of x_0, x_1, x_2, x_3. In such a situation, the arithmetic on x's can be done once and for all, and we need not count it. Thus (66) leads to the following scheme for evaluating the cyclic convolution w_0, w_1, w_2, w_3 when x_0, x_1, x_2, x_3 are known in advance:

$$\begin{gathered} s_1 = y_0 + y_2, \quad s_2 = y_1 + y_3, \quad s_3 = s_1 + s_2, \quad s_4 = s_1 - s_2, \\ s_5 = y_0 - y_2, \quad s_6 = y_3 - y_1, \quad s_7 = s_5 - s_6; \\ m_1 = \tfrac{x_0+x_1+x_2+x_3}{4} \cdot s_3, \quad m_2 = \tfrac{x_0-x_1+x_2-x_3}{4} \cdot s_4, \quad m_3 = \tfrac{x_0+x_1-x_2-x_3}{2} \cdot s_5, \\ m_4 = \tfrac{-x_0+x_1+x_2-x_3}{2} \cdot s_6, \quad m_5 = \tfrac{x_3-x_1}{2} \cdot s_7; \\ t_1 = m_1 + m_2, \quad t_2 = m_3 + m_5, \quad t_3 = m_1 - m_2, \quad t_4 = m_4 - m_5; \\ w_0 = t_1 + t_2, \quad w_1 = t_3 + t_4, \quad w_2 = t_1 - t_2, \quad w_3 = t_3 - t_4. \end{gathered} \tag{67}$$

There are 5 multiplications and 15 additions, while the definition of cyclic convolution involves 16 multiplications and 12 additions. We will prove later that 5 multiplications are necessary.

Going back to our original multiplication problem (52), using (60), we have derived the realization

$$\begin{pmatrix} 4 & 0 & 1 & 1 & 2 & \bar{2} & 0 \\ 0 & 0 & 1 & \bar{1} & 2 & 2 & \bar{2} \\ 0 & 4 & 1 & 1 & \bar{2} & 2 & 0 \end{pmatrix} \times \frac{1}{4}, \quad \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & \bar{1} & 0 & \bar{1} & 1 \\ 0 & 0 & 1 & 1 & \bar{1} & 0 & \bar{1} \\ 0 & 1 & 1 & \bar{1} & 0 & 1 & \bar{1} \end{pmatrix}, \quad \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \bar{1} & 1 & \bar{1} & 0 & 1 & \bar{1} \\ 0 & 0 & 1 & 1 & \bar{1} & 0 & \bar{1} \\ 0 & 0 & 1 & \bar{1} & 0 & \bar{1} & 1 \\ \bar{1} & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}. \tag{68}$$

This scheme uses one more than the minimum number of chain multiplications, but it requires far fewer parameter multiplications than (55). Of course, it must be admitted that the scheme is still rather complicated: If our goal is simply to compute the coefficients z_0, z_1, $\ldots$, z_5 of the product of two given polynomials $(x_0 + x_1u + x_2u^2)(y_0 + y_1u + y_2u^2 + y_3u^3)$, as a one-shot problem, our best bet is still to use the obvious method that does 12 multiplications and 6 additions—unless (say) the x's and y's are matrices. Note that if the x's are fixed as the y's vary, the new scheme does the evaluation with 7 multiplications and 17 additions. Even though (68) isn't especially useful as it stands, our derivation has

illustrated important techniques that are useful in a variety of other situations. For example, Winograd has used this approach to compute Fourier transforms using significantly fewer multiplications than the "fast Fourier transform" algorithm needs (see exercise 53).

Let us conclude this section by determining the exact rank of the $n \times n \times n$ tensor that corresponds to the multiplication of two polynomials modulo a third,

$$z_0 + z_1 u + \cdots + z_{n-1}u^{n-1}$$
$$= (x_0 + x_1 u + \cdots + x_{n-1}u^{n-1})(y_0 + y_1 u + \cdots + y_{n-1}u^{n-1}) \bmod p(u). \quad (69)$$

Here $p(u)$ stands for any given monic polynomial of degree n; in particular, $p(u)$ might be $u^n - 1$, so one of the results of our investigation will be to deduce the rank of the tensor corresponding to cyclic convolution of degree n. It will be convenient to write $p(u)$ in the form

$$p(u) = u^n - p_{n-1}u^{n-1} - \cdots - p_1 u - p_0, \quad (70)$$

so that $u^n \equiv p_0 + p_1 u + \cdots + p_{n-1}u^{n-1}$ (modulo $p(u)$).

The tensor element t_{ijk} is the coefficient of u^k in $u^{i+j} \bmod p(u)$; and this is the element in row i, column k of the matrix P^j, where

$$P = \begin{pmatrix} 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \ldots & 1 \\ p_0 & p_1 & p_2 & \cdots & p_{n-1} \end{pmatrix} \quad (71)$$

is called the "companion matrix" of $p(u)$. (The indices i, j, k in our discussion will run from 0 to $n-1$ instead of from 1 to n.) It is convenient to transpose the tensor, for if $T_{ijk} = t_{ikj}$ the individual layers of (T_{ijk}) for $k = 0, 1, 2, \ldots, n-1$ are simply given by the matrices

$$I \quad P \quad P^2 \quad \ldots \quad P^{n-1}. \quad (72)$$

The first rows of the matrices in (72) are respectively the unit vectors $(1,0,0,\ldots,0)$, $(0,1,0,\ldots,0)$, $(0,0,1,\ldots,0)$, $\ldots$, $(0,0,0,\ldots,1)$, hence a linear combination such as $\sum_{0\le k<n} v_k P^k$ will be the zero matrix if and only if the v_k are all zero. Furthermore, most of these linear combinations are actually nonsingular matrices, for we have

$$(w_0, w_1, \ldots, w_{n-1}) \sum_{0\le k<n} v_k P^k = (0,0,\ldots,0)$$
$$\text{if and only if} \quad v(u)w(u) \equiv 0 \pmod{p(u)},$$

where $v(u) = v_0 + v_1 u + \cdots + v_{n-1}u^{n-1}$ and $w(u) = w_0 + w_1 u + \cdots + w_{n-1}u^{n-1}$. Thus, $\sum_{0\le k<n} v_k P^k$ is a singular matrix if and only if the polynomial $v(u)$ is a multiple of some factor of $p(u)$. We are now ready to prove the desired result.

Theorem W (S. Winograd, 1975). *Let $p(u)$ be a monic polynomial of degree n whose complete factorization over a given infinite field is*

$$p(u) = p_1(u)^{e_1} \dots p_q(u)^{e_q}. \tag{73}$$

Then the rank of the tensor (72) *corresponding to the bilinear forms* (69) *is* $2n-q$ *over this field.*

Proof. The bilinear forms can be evaluated with only $2n-q$ chain multiplications by using rules (56), (57), (58) in an appropriate fashion, so we must prove only that the rank r is $\geq 2n-q$. The above discussion establishes the fact that $\operatorname{rank}(T_{(ij)k}) = n$; hence by Lemma T, any $n \times r$ realization A, B, C of (T_{ijk}) has $\operatorname{rank}(C) = n$. Our strategy will be to use Lemma T again, by finding a vector $(v_0, v_1, \dots, v_{n-1})$ that has the following two properties:

a) The vector $(v_0, v_1, \dots, v_{n-1})C$ has at most $q + r - n$ nonzero coefficients.
b) The matrix $v(P) = \sum_{0 \leq k < n} v_k P^k$ is nonsingular.

This and Lemma T will prove that $q + r - n \geq n$, since the identity

$$\sum_{1 \leq l \leq r} a_{il}\, b_{jl} \Bigg(\sum_{0 \leq k < n} v_k\, c_{kl} \Bigg) = v(P)_{ij}$$

shows how to realize the $n \times n \times 1$ tensor $v(P)$ of rank n with $q + r - n$ chain multiplications.

We may assume for convenience that the first n columns of C are linearly independent. Let D be the $n \times n$ matrix such that the first n columns of DC are equal to the identity matrix. Our goal will be achieved if there is a linear combination $(v_0, v_1, \dots, v_{n-1})$ of at most q rows of D, such that $v(P)$ is nonsingular; such a vector will satisfy conditions (a) and (b).

Since the rows of D are linearly independent, no irreducible factor $p_\lambda(u)$ divides the polynomials corresponding to every row. Given a vector $w = (w_0, w_1, \dots, w_{n-1})$, let "covered($w$)" be the set of all λ such that $w(u)$ is not a multiple of $p_\lambda(u)$. From two vectors v and w we can find a linear combination $v + \alpha w$ such that

$$\text{covered}(v + \alpha w) = \text{covered}(v) \cup \text{covered}(w), \tag{74}$$

for some α in the field. The reason is that if λ is covered by v or w but not both, then λ is covered by $v + \alpha w$ for all nonzero α; if λ is covered by both v and w but λ is not covered by $v + \alpha w$, then λ is covered by $v + \beta w$ for all $\beta \neq \alpha$. By trying $q + 1$ different values of α, at least one must yield (74). In this way we can systematically construct a linear combination of at most q rows of D, covering all λ for $1 \leq \lambda \leq q$. ∎

One of the most important corollaries of Theorem W is that the rank of a tensor can depend on the field from which we draw the elements of the realization A, B, C. For example, consider the tensor corresponding to cyclic convolution of degree 5; this is equivalent to multiplication of polynomials mod $p(u) = u^5 - 1$. Over the field of rational numbers, the complete factorization of $p(u)$ is $(u-1)\times(u^4+u^3+u^2+u+1)$ by exercise 4.6.2–32, so the tensor rank is $10-2=8$. On the other hand, the complete factorization over the real numbers, in terms of the number $\phi = \frac{1}{2}(1+\sqrt{5})$, is $(u-1)(u^2+\phi u+1)(u^2-\phi^{-1}u+1)$; thus, the rank is only 7, if we allow arbitrary real numbers to appear in A, B, C. Over the complex numbers the rank is 5. This phenomenon does not occur in two-dimensional tensors (i.e., matrices), where the rank can be determined by evaluating determinants of submatrices and testing for 0. The rank of a matrix does not change when the field containing its elements is embedded in a larger field, but the rank of a tensor *can* decrease when the field gets larger.

In the paper that introduced Theorem W [*Math. Systems Theory* **10** (1977), 169–180], Winograd went on to show that *all* realizations of (69) in $2n-q$ chain multiplications correspond to the use of (57), when q is greater than 1. Furthermore he has shown that the only way to evaluate the coefficients of $x(u)y(u)$ in $\deg(x)+\deg(y)+1$ chain multiplications is to use interpolation or to use (56) with a polynomial that splits into distinct linear factors in the field. Finally he has proved that the only way to evaluate $x(u)y(u) \bmod p(u)$ in $2n-1$ chain multiplications when $q=1$ is essentially to use (58). These results hold for *all* polynomial chains, not only "normal" ones. He has extended the results to multivariate polynomials in *SIAM J. Computing* **9** (1980), 225–229.

The tensor rank of an arbitrary $m \times n \times 2$ tensor in a suitably large field has been determined by Joseph Ja'Ja', *SIAM J. Computing* **8** (1979), 443–462.

For further reading. In this section we have barely scratched the surface of a very large subject in which many beautiful theories are emerging; a considerably more comprehensive treatment appears in the book *Computational Complexity of Algebraic and Numeric Problems* by A. Borodin and I. Munro (New York: American Elsevier, 1975).

EXERCISES

1. [*15*] What is a good way to evaluate an "odd" polynomial

$$u(x) = u_{2n+1}x^{2n+1} + u_{2n-1}x^{2n-1} + \cdots + u_1x?$$

▶ **2.** [*M20*] Instead of computing $u(x+x_0)$ by steps H1 and H2 as in the text, discuss the application of Horner's rule (2) when *polynomial* multiplication and addition are used instead of arithmetic in the domain of coefficients.

3. [*20*] Give a method analogous to Horner's rule, for evaluating a polynomial in two variables $\sum_{i+j\le n} u_{ij}x^iy^j$. (This polynomial has $(n+1)(n+2)/2$ coefficients, and "total degree" n.) Count the number of additions and multiplications you use.

4. [*M20*] The text shows that scheme (3) is superior to Horner's rule when we are evaluating a polynomial with real coefficients at a complex point z. Compare (3) to Horner's rule when *both* the coefficients and the variable z are complex numbers; how many (real) multiplications and addition-subtractions are required by each method?

5. [*M15*] Count the number of multiplications and additions required by the second-order rule (4).

6. [*22*] (L. de Jong and J. van Leeuwen.) Show how to improve on steps S1, ..., S4 of the Shaw–Traub algorithm by computing only about $\frac{1}{2}n$ powers of x_0.

7. [*M24*] How can $\beta_0, \ldots, \beta_n$ be calculated so that (6) has the value $u(x_0 + kh)$ for all integers k?

8. [*M20*] The factorial power $x^{\underline{k}}$ is defined to be $k!\binom{x}{k} = x(x-1)\ldots(x-k+1)$. Explain how to evaluate $u_n x^{\underline{n}} + \cdots + u_1 x^{\underline{1}} + u_0$ with at most n multiplications and $2n-1$ additions, starting with x and the $n+3$ constants $u_n, \ldots, u_0, 1, n-1$.

9. [*M24*] (H. J. Ryser.) Show that if $X = (x_{ij})$ is an $n \times n$ matrix, then

$$\operatorname{per}(X) = \sum (-1)^{n-\epsilon_1-\cdots-\epsilon_n} \prod_{1\le i\le n} \sum_{1\le j\le n} \epsilon_j x_{ij}$$

summed over all 2^n choices of $\epsilon_1, \ldots, \epsilon_n$ equal to 0 or 1 independently. Count the number of addition and multiplication operations required to evaluate $\operatorname{per}(X)$ by this formula.

10. [*M21*] The permanent of an $n \times n$ matrix $X = (x_{ij})$ may be calculated as follows: Start with the n quantities $x_{11}, x_{12}, \ldots, x_{1n}$. For $1 \le k < n$, assume that the $\binom{n}{k}$ quantities A_{kS} have been computed, for all k-element subsets S of $\{1, 2, \ldots, n\}$, where $A_{kS} = \sum x_{1j_1} \ldots x_{kj_k}$ summed over all $k!$ permutations $j_1 \ldots j_k$ of the elements of S; then form all of the sums

$$A_{(k+1)S} = \sum_{j\in S} A_{k(S\setminus\{j\})}\, x_{(k+1)j}.$$

We have $\operatorname{per}(X) = A_{n\{1,\ldots,n\}}$.

How many additions and multiplications does this method require? How much temporary storage is needed?

11. [*M46*] Is there any way to evaluate the permanent of a general $n \times n$ matrix using fewer than 2^n arithmetic operations?

12. [*M50*] What is the minimum number of multiplications required to form the product of two $n \times n$ matrices? What is the smallest exponent β such that $O(n^{\beta+\epsilon})$ multiplications are sufficient for all $\epsilon > 0$?

13. [*M23*] Find the inverse of the general discrete Fourier transform (37), by expressing $F(t_1, \ldots, t_n)$ in terms of the values of $f(s_1, \ldots, s_n)$. [*Hint:* See Eq. 1.2.9–13.]

▶ **14.** [*HM28*] (*"Fast Fourier transforms."*) Show that the scheme (40) can be used to evaluate the one-dimensional discrete Fourier transform

$$f(s) = \sum_{0\le t<2^n} F(t)\,\omega^{st}, \qquad \omega = e^{2\pi i/2^n}, \qquad 0 \le s < 2^n,$$

using arithmetic on complex numbers. Estimate the number of arithmetic operations performed.

▸ **15.** [*HM28*] The nth *divided difference* $f(x_0, x_1, \ldots, x_n)$ of a function $f(x)$ at $n+1$ distinct points x_0, x_1, $\ldots$, x_n is defined by the formula

$$f(x_0, x_1, \ldots, x_n) = (f(x_0, x_1, \ldots, x_{n-1}) - f(x_1, \ldots, x_{n-1}, x_n))/(x_0 - x_n),$$

for $n > 0$. Thus $f(x_0, x_1, \ldots, x_n) = \sum_{0\le k\le n} f(x_k)/\prod_{0\le j\le n,\, j\ne k}(x_k - x_j)$ is a symmetric function of its $n+1$ arguments. (a) Prove that $f(x_0, \ldots, x_n) = f^{(n)}(\theta)/n!$, for some θ between $\min(x_0, \ldots, x_n)$ and $\max(x_0, \ldots, x_n)$, if the nth derivative $f^{(n)}(x)$ exists and is continuous. [*Hint:* Prove the identity

$$f(x_0, x_1, \ldots, x_n) = \int_0^1 dt_1 \int_0^{t_1} dt_2 \ldots \int_0^{t_{n-1}} dt_n f^{(n)}(x_0(1-t_1) + x_1(t_1-t_2) + \cdots + x_{n-1}(t_{n-1}-t_n) + x_n(t_n-0)).$$

This formula also defines $f(x_0, x_1, \ldots, x_n)$ in a useful manner when the x_j are not distinct.] (b) If $y_j = f(x_j)$, show that $\alpha_j = f(x_0, \ldots, x_j)$ in Newton's interpolation polynomial (42).

16. [*M22*] How can we readily compute the coefficients of $u^{[n]}(x) = u_n x^n + \cdots + u_0$, if we are given the values of x_0, x_1, $\ldots$, x_{n-1}, α_0, α_1, $\ldots$, α_n in Newton's interpolation polynomial (42)?

17. [*M45*] Is there a way to evaluate the polynomial

$$\sum_{1\le i<j\le n} x_i x_j = x_1 x_2 + \cdots + x_{n-1} x_n$$

with fewer than $n-1$ multiplications and $2n-4$ additions? (There are $\binom{n}{2}$ terms.)

18. [*M20*] If the fourth-degree scheme (9) were changed to

$$y = (x + \alpha_0)x + \alpha_1, \qquad u(x) = ((y - x + \alpha_2)y + \alpha_3)\alpha_4,$$

what formulas for computing the α_j's in terms of the u_k's would take the place of (10)?

▸ **19.** [*M24*] Explain how to determine the adapted coefficients α_0, α_1, $\ldots$, α_5 in (11) from the coefficients u_5, $\ldots$, u_1, u_0 of $u(x)$, and find the α's for the particular polynomial $u(x) = x^5 + 5x^4 - 10x^3 - 50x^2 + 13x + 60$.

▸ **20.** [*21*] Write a MIX program that evaluates a fifth-degree polynomial according to scheme (11); try to make the program as efficient as possible, by making slight modifications to (11). Use MIX's floating point arithmetic operators FADD and FMUL, which are described in Section 4.2.1.

21. [*20*] Find two additional ways to evaluate the polynomial $x^6 + 13x^5 + 49x^4 + 33x^3 - 61x^2 - 37x + 3$ by scheme (12), using the two roots of (15) that were not considered in the text.

22. [*18*] What is the scheme for evaluating $x^6 - 3x^5 + x^4 - 2x^3 + x^2 - 3x - 1$, using Pan's method (16)?

23. [*HM30*] (J. Eve.) Let $f(z) = a_n z^n + a_{n-1}z^{n-1} + \cdots + a_0$ be a polynomial of degree n with real coefficients, having at least $n-1$ roots with a nonnegative real part. Let

$$g(z) = a_n z^n + a_{n-2}z^{n-2} + \cdots + a_{n \bmod 2}\, z^{n \bmod 2},$$
$$h(z) = a_{n-1}z^{n-1} + a_{n-3}z^{n-3} + \cdots + a_{(n-1) \bmod 2}\, z^{(n-1) \bmod 2}.$$

Assume that $h(z)$ is not identically zero.

a) Show that $g(z)$ has at least $n-2$ imaginary roots (i.e., roots whose real part is zero), and $h(z)$ has at least $n-3$ imaginary roots. [*Hint:* Consider the number of times the path $f(z)$ circles the origin as z goes around the path shown in Fig. 15, for a sufficiently large radius R.]

b) Prove that the squares of the roots of $g(z) = 0$ and $h(z) = 0$ are all real.

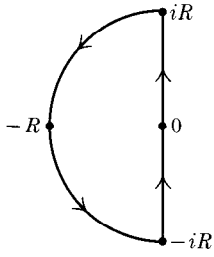


Fig. 15. Proof of Eve's theorem.

▸ **24.** [*M24*] Find values of c and α_k, β_k satisfying the conditions of Theorem E, for the polynomial $u(x) = (x+7)(x^2+6x+10)(x^2+4x+5)(x+1)$. Choose these values so that $\beta_2 = 0$. Give two different solutions to this problem!

25. [*M20*] When the construction in the proof of Theorem M is applied to the (inefficient) polynomial chain

$$\lambda_1 = \alpha_1 + \lambda_0, \quad \lambda_2 = -\lambda_0 - \lambda_0, \quad \lambda_3 = \lambda_1 + \lambda_1, \quad \lambda_4 = \alpha_2 \times \lambda_3,$$
$$\lambda_5 = \lambda_0 - \lambda_0, \quad \lambda_6 = \alpha_6 - \lambda_5, \quad \lambda_7 = \alpha_7 + \lambda_6, \quad \lambda_8 = \lambda_7 \times \lambda_7,$$
$$\lambda_9 = \lambda_1 \times \lambda_4, \quad \lambda_{10} = \alpha_8 - \lambda_9, \quad \lambda_{11} = \lambda_3 - \lambda_{10},$$

how can $\beta_1, \beta_2, \ldots, \beta_9$ be expressed in terms of $\alpha_1, \ldots, \alpha_8$?

▸ **26.** [*M21*] (a) Give the polynomial chain corresponding to Horner's rule for evaluating polynomials of degree $n = 3$. (b) Using the construction that appears in the text's proof of Theorem A, express κ_1, κ_2, κ_3, and the result polynomial $u(x)$ in terms of β_1, β_2, β_3, β_4, and x. (c) Show that the result set obtained in (b), as β_1, β_2, β_3, and β_4 independently assume all real values, omits certain vectors in the result set of (a).

27. [*M22*] Let R be a set that includes all $(n+1)$-tuples $(q_n, \ldots, q_1, q_0)$ of real numbers such that $q_n \neq 0$; prove that R does not have at most n degrees of freedom.

28. [*HM20*] Show that if $f_0(\alpha_1, \ldots, \alpha_s)$, $\ldots$, $f_s(\alpha_1, \ldots, \alpha_s)$ are multivariate polynomials with integer coefficients, then there is a nonzero polynomial $g(x_0, \ldots, x_s)$ with integer coefficients such that $g(f_0(\alpha_1, \ldots, \alpha_s), \ldots, f_s(\alpha_1, \ldots, \alpha_s)) = 0$ for all real α_1, $\ldots$, α_s. (Hence any polynomial chain with s parameters has at most s degrees of freedom.) [*Hint:* Use the theorems about "algebraic dependence" that are found, for example, in B. L. van der Waerden's *Modern Algebra*, tr. by Fred Blum (New York: Ungar, 1949), Section 64.]

▶ **29.** [*M20*] Let R_1, R_2, ..., R_m all be sets of $(n+1)$-tuples of real numbers having at most t degrees of freedom. Show that the union $R_1 \cup R_2 \cup \cdots \cup R_m$ also has at most t degrees of freedom.

▶ **30.** [*M28*] Prove that a polynomial chain with m_c chain multiplications and m_p parameter multiplications has at most $2m_c + m_p + \delta_{0m_c}$ degrees of freedom. [*Hint:* Generalize Theorem M, showing that the first chain multiplication and each parameter multiplication can essentially introduce only one new parameter into the result set.]

31. [*M23*] Prove that a polynomial chain capable of computing all *monic* polynomials of degree n has at least $\lfloor n/2 \rfloor$ multiplications and at least n addition-subtractions.

32. [*M24*] Find a polynomial chain of minimum possible length that can compute all polynomials of the form $u_4x^4 + u_2x^2 + u_0$; and prove that its length is minimal.

▶ **33.** [*M25*] Let $n \geq 3$ be odd. Prove that a polynomial chain with $\lfloor n/2 \rfloor + 1$ multiplication steps cannot compute all polynomials of degree n unless it has at least $n+2$ addition-subtraction steps. [*Hint:* See exercise 30.]

34. [*M26*] Let λ_0, λ_1, ..., λ_r be a polynomial chain in which all of the addition and subtraction steps are parameter steps, and in which there is at least one parameter multiplication. Assume that this scheme has m multiplications and $k = r - m$ addition-subtractions, and that the polynomial computed by the chain has maximum degree n. Prove that all polynomials computable by this chain, for which the coefficient of x^n is not zero, can be computed by another chain that has at most m multiplications and at most k additions, and no subtractions; and whose last step is the only parameter multiplication.

▶ **35.** [*M25*] Show that any polynomial chain that computes a general fourth-degree polynomial using three multiplications must have at least five addition-subtractions. [*Hint:* Assume that there are only four addition-subtractions, and show that exercise 34 applies; this means the scheme must have a particular form that is incapable of representing all fourth-degree polynomials.]

36. [*M27*] Show that any polynomial chain that computes a general sixth-degree polynomial using only four multiplications must have at least seven addition-subtractions. (Cf. exercise 35.)

37. [*M21*] (T. S. Motzkin.) Show that "almost all" rational functions of the form

$$(u_nx^n + u_{n-1}x^{n-1} + \cdots + u_1x + u_0)/(x^n + v_{n-1}x^{n-1} + \cdots + v_1x + v_0),$$

with coefficients in a field S, can be evaluated using the scheme

$$\alpha_1 + \beta_1/(x + \alpha_2 + \beta_2/(x + \cdots + \beta_n/(x + \alpha_{n+1})\ldots)),$$

for suitable α_j, β_j in S. (This continued fraction scheme has n divisions and $2n$ additions; by "almost all" rational functions we mean all except those whose coefficients satisfy some nontrivial polynomial equation.) Determine the α's and β's for the rational function $(x^2 + 10x + 29)/(x^2 + 8x + 19)$.

▸ **38.** [*HM32*] (V. Îa. Pan, 1962.) The purpose of this exercise is to prove that Horner's rule is really optimal if no preliminary adaptation of coefficients is made; we need n multiplications and n additions to compute $u_n x^n + \cdots + u_1 x + u_0$, if the variables u_n, $\ldots$, u_1, u_0, x, and arbitrary constants are given. Consider chains that are as before except that u_n, $\ldots$, u_1, u_0, x are each considered to be variables; we may say, for example, that $\lambda_{-j-1} = u_j$, $\lambda_0 = x$. In order to show that Horner's rule is best, it is convenient to prove a somewhat more general theorem: Let $A = (a_{ij})$, $0 \le i \le m$, $0 \le j \le n$, be an $(m+1) \times (n+1)$ matrix of real numbers, of rank $n+1$; and let $B = (b_0, \ldots, b_m)$ be a vector of real numbers. Prove that *any polynomial chain that computes*

$$P(x; u_0, \ldots, u_n) = \sum_{0 \le i \le m} (a_{i0}u_0 + \cdots + a_{in}u_n + b_i)x^i$$

involves at least n chain multiplications. (Note that this does not mean only that we are considering some fixed chain in which the parameters α_j are assigned values depending on A and B; it means that both the chain *and* the values of the α's may depend on the given matrix A and vector B. No matter how A, B, and the values of α_j are chosen, it is impossible to compute $P(x; u_0, \ldots, u_n)$ without doing n "chain-step" multiplications.) The assumption that A has rank $n+1$ implies that $m \ge n$. [*Hint:* Show that from any such scheme we can derive another that has fewer chain multiplications and that has n decreased by one.]

39. [*M29*] (T. S. Motzkin, 1954.) Show that schemes of the form $w_1 = x(x+\alpha_1)+\beta_1$, $w_k = w_{k-1}(w_1 + \gamma_k x + \alpha_k) + \delta_k x + \beta_k$ for $1 < k \le m$, where the α_k, β_k are real and the γ_k, δ_k are integers, can be used to evaluate all monic polynomials of degree $2m$ over the real numbers. (We may have to choose α_k, β_k, γ_k, and δ_k differently for different polynomials.) Try to let $\delta_k = 0$ whenever possible.

40. [*M41*] Can the lower bound in the number of multiplications in Theorem C be raised from $\lfloor n/2 \rfloor + 1$ to $\lceil n/2 \rceil + 1$? (Cf. exercise 33.)

41. [*22*] Show that the real and imaginary parts of $(a + bi)(c + di)$ can be obtained by doing 3 multiplications and 5 additions of real numbers, where two of the additions involve a and b only.

42. [*36*] (M. Paterson and L. Stockmeyer.) (a) Prove that a polynomial chain with $m \ge 2$ chain multiplications has at most $m^2 + 1$ degrees of freedom. (b) Show that for all $n \ge 2$ there exist polynomials of degree n, all of whose coefficients are 0 or 1, that cannot be evaluated by any polynomial chain with fewer than $\lfloor \sqrt{n} \rfloor$ multiplications, if we require all parameters α_j to be integers. (c) Show that any polynomial of degree n with integer coefficients can be evaluated by an all-integer algorithm that performs at most $2\lfloor \sqrt{n} \rfloor$ multiplications, if we don't care how many additions we do.

43. [*22*] Explain how to evaluate $x^n + \cdots + x + 1$ with $2l(n+1) - 2$ multiplications and $l(n+1)$ additions (no divisions or subtractions), where $l(n)$ is the function studied in Section 4.6.3.

▸ **44.** [*HM22*] Let (t_{ijk}) be an $m \times n \times s$ tensor, and let F, G, H be nonsingular matrices of respective sizes $m \times m$, $n \times n$, $s \times s$. If

$$T_{ijk} = \sum_{1 \le p \le m} \sum_{1 \le q \le n} \sum_{1 \le r \le s} F_{ip} G_{jq} H_{kr} t_{pqr}$$

for all i, j, k, prove that the tensor (T_{ijk}) has the same rank as (t_{ijk}). [*Hint:* Consider what happens when F^{-1}, G^{-1}, H^{-1} are applied in the same way to (T_{ijk}).]

45. [*M28*] Prove that all pairs (z_1, z_2) of bilinear forms in (x_1, x_2) and (y_1, y_2) can be evaluated with at most three chain multiplications. In other words, show that every $2 \times 2 \times 2$ tensor has rank ≤ 3.

46. [*M25*] Prove that for all m, n, and s there exists an $m \times n \times s$ tensor whose rank is at least $\lceil mns/(m+n+s) \rceil$. Conversely, show that every $m \times n \times s$ tensor has rank at most $mns/\max(m, n, s)$.

47. [*M48*] Is it possible to determine the rank of any given tensor (t_{ijk}) over, say, the field of rational numbers, in a finite number of steps? (There is a finite way to compute the tensor rank over algebraically closed fields like the complex numbers, since this is a special case of the results of Alfred Tarski, *A Decision Method for Elementary Algebra and Geometry*, 2nd ed. (Berkeley, California: Univ. of California Press, 1951); but the known algorithms do not make this computation really feasible except for very small tensors. Over the field of rational numbers, the problem isn't even known to be solvable in finite time.)

48. [*M49*] If (t_{ijk}) and (t'_{ijk}) are tensors of sizes $m \times n \times s$ and $m' \times n' \times s'$, respectively, their *direct sum* $(t_{ijk}) \oplus (t'_{ijk}) = (t''_{ijk})$ is the $(m+m') \times (n+n') \times (s+s')$ tensor defined by $t''_{ijk} = t_{ijk}$ if $i \leq m$, $j \leq n$, $k \leq s$; $t''_{ijk} = t'_{i-m,j-n,k-s}$ if $i > m$, $j > n$, $k > s$; and $t''_{ijk} = 0$ otherwise. Their direct product $(t_{ijk}) \otimes (t'_{ijk}) = (t'''_{ijk})$ is the $mm' \times nn' \times ss'$ tensor defined by $t_{\langle ii' \rangle \langle jj' \rangle \langle kk' \rangle} = t_{ijk} t'_{i'j'k'}$. Derive the upper bounds $\operatorname{rank}(t''_{ijk}) \leq \operatorname{rank}(t_{ijk}) + \operatorname{rank}(t'_{ijk})$ and $\operatorname{rank}(t'''_{ijk}) \leq \operatorname{rank}(t_{ijk}) \cdot \operatorname{rank}(t'_{ijk})$.

▸ **49.** [*HM25*] Show that the rank of an $m \times n \times 1$ tensor (t_{ijk}) is the same as its rank as an $m \times n$ matrix (t_{ij1}), according to the traditional definition of matrix rank as the maximum number of linearly independent rows.

50. [*HM20*] (S. Winograd.) Let (t_{ijk}) be the $mn \times n \times m$ tensor corresponding to multiplication of an $m \times n$ matrix by an $n \times 1$ column vector. Prove that the rank of (t_{ijk}) is mn.

▸ **51.** [*M24*] (S. Winograd.) Devise an algorithm for cyclic convolution of degree 2 that uses 2 multiplications and 4 additions, not counting operations on the x_i. Similarly, devise an algorithm for degree 3, using 4 multiplications and 11 additions. (Cf. (67), which solves the analogous problem for degree 4.)

52. [*M25*] (S. Winograd.) Let $n = n'n''$ where $\gcd(n', n'') = 1$. Given normal schemes for cyclic convolutions of degrees n' and n'', using respectively (m', m'') chain multiplications, (p', p'') parameter multiplications, and (a', a'') additions, show how to construct a normal scheme for cyclic convolution of degree n using $m'm''$ chain multiplications, $p'n'' + m'p''$ parameter multiplications, and $a'n'' + m'a''$ additions.

53. [*HM40*] (S. Winograd.) Let ω be a complex mth root of unity, and consider the one-dimensional discrete Fourier transform

$$f(s) = \sum_{1 \leq t \leq m} F(t)\omega^{st}, \qquad \text{for } 1 \leq s \leq m.$$

(a) When $m = p^e$ is a power of an odd prime, show that efficient normal schemes for computing cyclic convolutions of degrees $(p-1)p^k$, for $0 \leq k < e$, will lead to efficient algorithms for computing the Fourier transform on m complex numbers. Give a similar construction for the case $p = 2$. (b) When $m = m'm''$ and $\gcd(m', m'') = 1$, show that Fourier transformation algorithms for m' and m'' can be combined to yield a Fourier transformation algorithm for m elements.

54. [*M23*] Theorem W refers to an infinite field. How many elements must a finite field have in order for the proof of Theorem W to be valid?

55. [*HM22*] Determine the rank of tensor (72) when P is an *arbitrary* $n \times n$ matrix.

56. [*M32*] (V. Strassen.) Show that any polynomial chain that evaluates a set of *quadratic forms* $\sum_{1\le i,j\le n} \tau_{ijk}x_ix_k$ for $1 \le k \le s$ must use at least $\frac{1}{2}\text{rank}(\tau_{ijk}+\tau_{jik})$ chain multiplications. [*Hint:* Show that the minimum number of chain multiplications is the minimum rank of (t_{ijk}) taken over all tensors (t_{ijk}) such that $t_{ijk}+t_{jik} = \tau_{ijk}+\tau_{jik}$ for all i, j, k.] Use this to prove that any polynomial chain that evaluates a set of bilinear forms (45) corresponding to a tensor (t_{ijk}), whether normal or abnormal, must use at least $\frac{1}{2}\text{rank}(t_{ijk})$ chain multiplications.

57. [*M20*] Show that fast Fourier transforms can be used to compute the coefficients of the product $x(u)y(u)$ of two given polynomials of degree n, using $O(n \log n)$ operations of (exact) addition and multiplication of complex numbers. [*Hint:* Consider the product of Fourier transforms of the coefficients.]

58. [*HM28*] (a) Show that any realization A, B, C of the polynomial multiplication tensor (53) must have the following property: Any nonzero linear combination of the three rows of A must be a vector with at least four nonzero elements; and any nonzero linear combination of the four rows of B must have at least three nonzero elements. (b) Find a realization A, B, C of (53) using only 0, $+1$, and -1 as elements, where $t = 8$. Try to use as many 0's as possible.

▶ **59.** [*M40*] (**H. J. Nussbaumer, 1980.**) The text defines the cyclic convolution of two sequences $(x_0, x_1, \ldots, x_{n-1})$ and $(y_0, y_1, \ldots, y_{n-1})$ to be the sequence $(z_0, z_1, \ldots, z_{n-1})$ where $z_k = x_0y_k + \cdots + x_ky_0 + x_{k+1}y_{n-1} + \cdots + x_{n-1}y_{k+1}$. Let us define the *negacyclic convolution* similarly, but with

$$z_k = x_0y_k + \cdots + x_ky_0 - (x_{k+1}y_{n-1} + \cdots + x_{n-1}y_{k+1}).$$

Construct efficient algorithms for cyclic and negacyclic convolution over the integers when n is a power of 2. Your algorithms should deal entirely with integers, and they should perform at most $O(n \log n)$ multiplications and at most $O(n \log n \log\log n)$ additions or subtractions or divisions of even numbers by 2.

60. [*M27*] (V. Îa. Pan.) The problem of $(m \times n)$ times $(n \times s)$ matrix multiplication corresponds to an $mn \times ns \times sm$ tensor $(t_{\langle i,j'\rangle\langle j,k'\rangle\langle k,i'\rangle})$ where $t_{\langle i,j'\rangle\langle j,k'\rangle\langle k,i'\rangle} = 1$ if and only $i' = i$ and $j' = j$ and $k' = k$. The rank of this tensor $T(m,n,s)$ is the smallest r such that numbers $a_{ij'l}$, $b_{jk'l}$, $c_{ki'l}$ exist satisfying

$$\sum_{\substack{1\le i\le m\\1\le j\le n\\1\le k\le s}} x_{ij}\,y_{jk}\,z_{ki} = \sum_{1\le l\le r}\Bigg(\sum_{\substack{1\le i\le m\\1\le j'\le n}} a_{ij'l}\,x_{ij'}\Bigg)\Bigg(\sum_{\substack{1\le j\le n\\1\le k'\le s}} b_{jk'l}\,y_{jk'}\Bigg)\Bigg(\sum_{\substack{1\le k\le s\\1\le i'\le m}} c_{ki'l}\,z_{ki'}\Bigg).$$

Let $M(n)$ be the rank of $T(n,n,n)$. The purpose of this exercise is to exploit the symmetry of such a trilinear representation, obtaining efficient realizations of matrix multiplication over the integers when $m = n = s = 2\nu$. For convenience we divide the indices $\{1,\ldots,n\}$ into two subsets $O = \{1,3,\ldots,n-1\}$ and $E = \{2,4,\ldots,n\}$ of ν elements each, and we set up a one-to-one correspondence between O and E by the rule $\tilde{\imath} = i+1$ if $i \in O$; $\tilde{\imath} = i-1$ if $i \in E$. Thus we have $\tilde{\tilde{\imath}} = i$ for all indices i.

a) The first construction is based on the identity

$$abc + ABC = (a+A)(b+B)(c+C) - (a+A)bC - A(b+B)c - aB(c+C).$$

It follows that

$$\sum_{1\le i,j,k\le n} x_{ij}\,y_{jk}\,z_{ki} = \sum_{(i,j,k)\in S} (x_{ij}+x_{\bar{k}\tilde{\imath}})(y_{jk}+y_{\tilde{\imath}\bar{\jmath}})(z_{ki}+z_{\bar{\jmath}\bar{k}}) - \Sigma_1 - \Sigma_2 - \Sigma_3,$$

where $S = E\times E\times E \cup E\times E\times O \cup E\times O\times E \cup O\times E\times E$ is the set of all triples of indices containing at most one odd index; Σ_1 is the sum of all terms of the form $(x_{ij}+x_{\bar{k}\tilde{\imath}})y_{jk}\,z_{\bar{\jmath}\bar{k}}$ for $(i,j,k)\in S$; and Σ_2, Σ_3 similarly are sums of the terms $x_{\bar{k}\tilde{\imath}}(y_{jk}+y_{\tilde{\imath}\bar{\jmath}})z_{ki}$, $x_{ij}\,y_{\tilde{\imath}\bar{\jmath}}(z_{ki}+z_{\bar{\jmath}\bar{k}})$. Clearly S has $4\nu^3 = \frac{1}{2}n^3$ terms. Show that each of Σ_1, Σ_2, Σ_3 can be realized as the sum of $3\nu^2$ trilinear terms; furthermore, if the 3ν triples of the forms $(i,i,\tilde{\imath})$ and $(i,\tilde{\imath},i)$ and $(\tilde{\imath},i,i)$ are removed from S, we can modify Σ_1, Σ_2, and Σ_3 in such a way that the identity is still valid, without adding any new trilinear terms. Thus $M(n) \le \frac{1}{2}n^3 - \frac{3}{2}n + \frac{9}{4}n^2$ when n is even.

b) Apply the method of (a) to show that two *independent* matrix multiplication problems of size $m\times n\times s$ can be performed with $mns + mn + ns + sm$ noncommutative multiplications.

c) The second construction is based on the identity

$$\begin{aligned} abc + ABC + \mathcal{A}\mathcal{B}\mathcal{C} = {} & (a+A+\mathcal{A})(b+B+\mathcal{B})(c+C+\mathcal{C}) \\ & - (a\mathcal{B}(c+C+\mathcal{C}) + Ab(c+C+\mathcal{C}) + \mathcal{A}B(c+C+\mathcal{C})) \\ & - (a(b+B)C + A(B+\mathcal{B})\mathcal{C} + \mathcal{A}(\mathcal{B}+b)c) \\ & - ((a+\mathcal{A})b\mathcal{C} + (A+a)Bc + (\mathcal{A}+A)\mathcal{B}C) \\ & - (aB\mathcal{C} + A\mathcal{B}c + \mathcal{A}bC). \end{aligned}$$

Show that

$$\sum_{1\le i,j,k\le n} x_{ij}\,y_{jk}\,z_{ki} = \sum_{\substack{(i,j,k)\in S \\ 0\le\epsilon,\varsigma,\eta\le 1}} t(i,j,k;\epsilon,\varsigma,\eta) - \Sigma_1 - \Sigma_2 - \Sigma_3;$$

here $t(i,j,k;\epsilon,\varsigma,\eta) = ((-1)^{\varsigma+\eta}x_{i+\epsilon,j+\varsigma} + (-1)^{\epsilon+\varsigma}x_{j+\eta,k+\epsilon} + (-1)^{\eta+\epsilon}x_{k+\varsigma,i+\eta})\cdot((-1)^{\eta+\epsilon}y_{j+\varsigma,k+\eta} + (-1)^{\varsigma+\eta}y_{k+\epsilon,i+\varsigma} + (-1)^{\epsilon+\varsigma}y_{i+\eta,j+\epsilon})\cdot((-1)^{\epsilon+\varsigma}z_{k+\eta,i+\epsilon} + (-1)^{\eta+\epsilon}z_{i+\varsigma,j+\eta} + (-1)^{\varsigma+\eta}z_{j+\epsilon,k+\varsigma})$ corresponds to the first term on the right-hand side of the above identity and Σ_1, Σ_2, Σ_3 correspond respectively to the next three groups of terms; the remaining terms (namely those corresponding to $aB\mathcal{C} + A\mathcal{B}c + \mathcal{A}bC$) cancel out of the sum. The set S in this case is different from the S in part (a); it consists of all $(i,j,k)\in O\times O\times O$ such that $i\le j$ and $i<k$. It follows from this construction that $M(n) \le \frac{8}{3}((\frac{n}{2})^3 - (\frac{n}{2})) + 6n^2$ when n is even.

61. [*M23*] Let (t_{ijk}) be a tensor over an arbitrary field. We define $\operatorname{rank}_d(t_{ijk})$ as the minimum value of r such that there is a realization of the form

$$\sum_{1\le l\le r} a_{il}(u)b_{jl}(u)c_{kl}(u) = t_{ijk}u^d + O(u^{d+1}),$$

where $a_{il}(u)$, $b_{jl}(u)$, $c_{kl}(u)$ are polynomials in u over the field. Thus rank_0 is the ordinary rank of a tensor. Prove that (a) $\operatorname{rank}_{d+1}(t_{ijk}) \le \operatorname{rank}_d(t_{ijk})$; (b) $\operatorname{rank}(t_{ijk}) \le \binom{d+2}{2}\operatorname{rank}_d(t_{ijk})$; (c) $\operatorname{rank}_d((t_{ijk})\oplus(t'_{ijk})) \le \operatorname{rank}_d(t_{ijk}) + \operatorname{rank}_d(t'_{ijk})$, in the sense of exercise 48; (d) $\operatorname{rank}_{d+d'}((t_{ijk})\otimes(t'_{ijk})) \le \operatorname{rank}_d(t_{ijk})\cdot\operatorname{rank}_{d'}(t'_{ijk})$.

62. [*M24*] The *border rank* of (t_{ijk}), denoted by $\underline{\text{rank}}(t_{ijk})$, is $\min_{d\ge 0} \text{rank}_d(t_{ijk})$, where rank_d is defined in exercise 61. Prove that the tensor $\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)\left(\begin{smallmatrix}0&1\\0&0\end{smallmatrix}\right)$ has rank 3 but border rank 2, over every field.

63. [*M30*] Let $T(m,n,s)$ be the tensor for matrix multiplication as in exercise 59. (a) Show that $T(m,n,s)\otimes T(M,N,S) = T(mM,nN,sS)$. (b) If $T(m,n,s)$ has rank $\le r$, show that the number of noncommutative multiplications $M(N)$ for the product of $N\times N$ matrices is $O(N^{\beta(m,n,s,r)})$ as $N\to\infty$, where $\beta(m,n,s,r) = 3\log r/\log mns$. (c) If $T(m,n,s)$ has border rank $\le r$, show that we have $M(N) = O(N^{\beta(m,n,s,r)}(\log N)^2)$. (d) If the tensor $pT(m,n,s)$ obtained by taking the direct sum of p copies of $T(m,n,s)$ has border rank $\le pr$, where $p < r$, show that a similar formula holds. (This tensor corresponds to p independent matrix multiplications.)

64. [*M40*] (Pan and Winograd.) The purpose of this exercise is to combine the ideas of exercises 60–63, in order to obtain asymptotic bounds on $M(N)$ as $N\to\infty$.

a) Modify the construction of exercise 60(b) to show that $T(m,n,s)\oplus T(s,m,n)$ has border rank $\le mns + mn + ns$.

b) Show that the construction of exercise 60(c) can be altered to prove that the border rank of $T(m,n,2s)\oplus T(2n,s,m)\oplus T(s,2m,n)$ is at most $2(m+1)n(s+2)$, by finding constants α_1, α_2, …, α_6 such that

$$\begin{aligned}
u^{2d}\sum_{i,j,k,\sigma}&(x_{ij}y_{j\tilde{k}}z_{\tilde{k}i} + X_{\tilde{\jmath}k}Y_{ki}Z_{i\tilde{\jmath}} + \mathcal{X}_{k\tilde{\imath}}\mathcal{Y}_{\tilde{\imath}j}Z_{jk})\\
&\equiv \sum_{i,j,k,\sigma}(\sigma x_{ij} + u^d X_{\tilde{\jmath}k} + \sigma u^{d+2}\mathcal{X}_{k\tilde{\imath}})(\sigma u^{d-1}y_{j\tilde{k}} + \sigma u^d Y_{ki} + \mathcal{Y}_{\tilde{\imath}j})\\
&\qquad\qquad\times(u^{d+1}z_{\tilde{k}i} + \sigma Z_{i\tilde{\jmath}} + \sigma u^{d-2}Z_{jk})\\
&+\alpha_2\sum_{i,j,\sigma}(\alpha_1\sigma x_{ij} + \sigma u^{d+2}\textstyle\sum_k \mathcal{X}_{k\tilde{\imath}})(\alpha_1\mathcal{Y}_{\tilde{\imath}j} + \sigma u^d\sum_k Y_{ki})\\
&\qquad\qquad\times(\alpha_1\sigma Z_{i\tilde{\jmath}} + u^{d+1}\textstyle\sum_k z_{\tilde{k}i})\\
&+\alpha_3\sum_{i,j,\sigma}(\sigma x_{ij} + u^d\textstyle\sum_k X_{\tilde{\jmath}k})(\mathcal{Y}_{\tilde{\imath}j} + \sigma u^{d-1}\sum_k y_{j\tilde{k}})\\
&\qquad\qquad\times(\sigma Z_{i\tilde{\jmath}} + \sigma u^{d-2}\textstyle\sum_k Z_{jk})\\
&+\alpha_4\sum_{j,k,\sigma}(\sigma\textstyle\sum_i x_{ij} + u^d X_{\tilde{\jmath}k})(\sum_i \mathcal{Y}_{\tilde{\imath}j} + \sigma u^{d-1}y_{j\tilde{k}})\\
&\qquad\qquad\times(\sigma\textstyle\sum_i Z_{i\tilde{\jmath}} + \sigma u^{d-2}Z_{jk})\\
&+\alpha_5\sum_{j,\sigma}(\sigma\textstyle\sum_i x_{ij} + u^d\sum_k X_{\tilde{\jmath}k})(\sum_i \mathcal{Y}_{\tilde{\imath}j} + \sigma u^{d-1}\sum_k y_{j\tilde{k}})\\
&\qquad\qquad\times(\sigma\textstyle\sum_i Z_{i\tilde{\jmath}} + \sigma u^{d-2}\sum_k Z_{jk})\\
&+\alpha_6\sum_{j,\sigma}(\sigma\textstyle\sum_i x_{ij})(\sum_i \mathcal{Y}_{\tilde{\imath}j})(\sigma\sum_i Z_{i\tilde{\jmath}})
\end{aligned}$$

(modulo u^{2d+1}), for sufficiently large d. Here the summations are to be carried out for $1\le i\le m$, $1\le j\le n$, $1\le k\le s$, and $\sigma = \pm 1$; the subscript $\tilde{\imath}$ means σi, so that $\tilde{\imath}$ takes the $2m$ values ± 1, …, $\pm m$.

c) Use the result of (b) in connection with exercise 63 to show that $M(N) = O(N^{\beta+\epsilon})$ for all $\epsilon > 0$, where $\beta = \beta(m,n,2s,\frac{2}{3}(m+1)n(s+2))$.

*4.7. MANIPULATION OF POWER SERIES

IF WE ARE GIVEN two power series

$$U(z) = U_0 + U_1 z + U_2 z^2 + \cdots, \qquad V(z) = V_0 + V_1 z + V_2 z^2 + \cdots, \tag{1}$$

whose coefficients belong to a field, we can form their sum, their product, their quotient, etc., to obtain new power series. A polynomial is obviously a special case of a power series, in which there are only finitely many terms.

Of course, only a finite number of terms can be represented and stored within a computer, so it makes sense to ask whether power series arithmetic is even possible on computers; and if it is possible, what makes it different from polynomial arithmetic? The answer is that we work with only the first N coefficients of the power series, where N is a parameter that may in principle be arbitrarily large; instead of ordinary polynomial arithmetic, we are essentially doing polynomial arithmetic modulo z^N, and this often leads to a somewhat different point of view. Furthermore, special operations like "reversion" can be performed on power series but not on polynomials, since polynomials are not closed under these operations.

Manipulation of power series has several applications to numerical analysis, but perhaps its greatest use is the determination of asymptotic expansions (as we have seen in Section 1.2.11.3), or the calculation of quantities defined by certain generating functions. The latter applications make it desirable to calculate the coefficients exactly, instead of with floating point arithmetic. All of the algorithms in this section, with obvious exceptions, can be done using rational operations only, so the techniques of Section 4.5.1 can be used to obtain exact results when desired.

The calculation of $W(z) = U(z) \pm V(z)$ is, of course, trivial, since we have $W_n = U_n \pm V_n$ for $n = 0, 1, 2, \ldots$. It is also easy to calculate $W(z) = U(z)V(z)$, using the familiar "Cauchy product rule":

$$W_n = \sum_{0 \le k \le n} U_k V_{n-k} = U_0 V_n + U_1 V_{n-1} + \cdots + U_n V_0. \tag{2}$$

The quotient $W(z) = U(z)/V(z)$, when $V_0 \neq 0$, can be obtained by interchanging U and W in (2); we obtain the rule

$$\begin{aligned} W_n &= \Bigg(U_n - \sum_{0 \le k < n} W_k V_{n-k} \Bigg) \Big/ V_0 \\ &= (U_n - W_0 V_n - W_1 V_{n-1} - \cdots - W_{n-1} V_1)/V_0. \end{aligned} \tag{3}$$

This recurrence relation for the W's makes it easy to determine $W_0, W_1, W_2, \ldots$ successively, without inputting U_n and V_n until after W_{n-1} has been computed. Let us say that a power series manipulation algorithm with the latter property is "on-line"; an on-line algorithm can be used to determine N coefficients $W_0, W_1, \ldots, W_{N-1}$ of the result without knowing N in advance, so it is possible in

theory to run the algorithm indefinitely and compute the entire power series; or to run it until a certain condition is met. (The opposite of "on-line" is "off-line.")

If the coefficients U_k and V_k are integers but the W_k are not, the recurrence relation (3) involves computation with fractions. This can be avoided by the all-integer approach described in exercise 2.

Let us now consider the operation of computing $W(z) = V(z)^\alpha$, where α is an "arbitrary" power. For example, we could calculate the square root of $V(z)$ by taking $\alpha = \frac{1}{2}$, or we could find $V(z)^{-10}$ or even $V(z)^\pi$. If V_m is the first nonzero coefficient of $V(z)$, we have

$$\begin{aligned} V(z) &= V_m z^m(1 + (V_{m+1}/V_m)z + (V_{m+2}/V_m)z^2 + \cdots), \\ V(z)^\alpha &= V_m^\alpha z^{\alpha m}(1 + (V_{m+1}/V_m)z + (V_{m+2}/V_m)z^2 + \cdots)^\alpha. \end{aligned} \tag{4}$$

This will be a power series if and only if αm is a nonnegative integer. From (4) we can see that the problem of computing general powers can be reduced to the case that $V_0 = 1$; then the problem is to find coefficients of

$$W(z) = (1 + V_1 z + V_2 z^2 + V_3 z^3 + \cdots)^\alpha. \tag{5}$$

Clearly $W_0 = 1^\alpha = 1$.

The obvious way to find the coefficients of (5) is to use the binomial theorem (Eq. 1.2.9–19), or (if α is a positive integer) to try repeated squaring as in Section 4.6.3; but a much simpler and more efficient device for calculating powers has been suggested by J. C. P. Miller. [See P. Henrici, *JACM* **3** (1956), 10–15.] If $W(z) = V(z)^\alpha$, we have by differentiation

$$W_1 + 2W_2 z + 3W_3 z^2 + \cdots = W'(z) = \alpha V(z)^{\alpha-1} V'(z); \tag{6}$$

therefore

$$W'(z)V(z) = \alpha W(z)V'(z). \tag{7}$$

If we now equate the coefficients of z^{n-1} in (7), we find that

$$\sum_{0 \le k \le n} k W_k V_{n-k} = \alpha \sum_{0 \le k \le n} (n-k) W_k V_{n-k}, \tag{8}$$

and this gives us a useful computational rule valid for all $n \ge 1$:

$$\begin{aligned} W_n &= \sum_{1 \le k \le n} \left(\left(\frac{\alpha+1}{n} \right) k - 1 \right) V_k W_{n-k} \\ &= \big((\alpha{+}1{-}n)V_1 W_{n-1} + (2\alpha{+}2{-}n)V_2 W_{n-2} + \cdots + n\alpha V_n\big)/n. \end{aligned} \tag{9}$$

This equation leads to a simple on-line algorithm by which we can successively determine W_1, W_2, ..., using approximately $2n$ multiplications to compute the nth coefficient. Note the special case $\alpha = -1$, in which (9) becomes the special case $U(z) = V_0 = 1$ of (3).

A similar technique can be used to form $f\big(V(z)\big)$ when f is any function that satisfies a simple differential equation. (For example, see exercise 4.) A comparatively straightforward "power series method" is often used to obtain the solution of differential equations; this technique is explained in nearly all textbooks about differential equations.

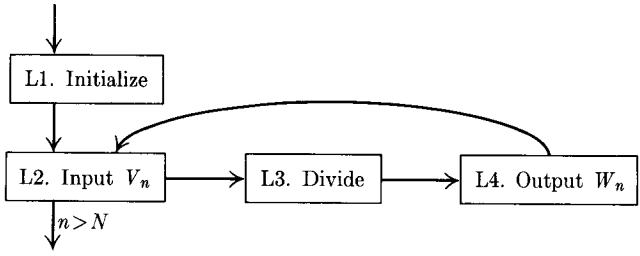


Fig. 16. Power series reversion by Algorithm L.

Reversion of series. The transformation of power series that is perhaps of greatest interest is called "reversion of series." This problem is to solve the equation

$$z = t + V_2t^2 + V_3t^3 + V_4t^4 + \cdots \tag{10}$$

for t, obtaining the coefficients of the power series

$$t = z + W_2z^2 + W_3z^3 + W_4z^4 + \cdots. \tag{11}$$

Several interesting ways to achieve such a reversion are known. We might say that the "classical" method is one based on Lagrange's remarkable inversion formula [*Mémoires Acad. Royale des Sciences et Belles-Lettres de Berlin* **24** (1768), 251–326], which states that

$$W_n = U_{n-1}/n,$$

if

$$U_0 + U_1t + U_2t^2 + \cdots = (1 + V_2t + V_3t^2 + \cdots)^{-n}. \tag{12}$$

For example, we have $(1-t)^{-5} = \binom{4}{4} + \binom{5}{4}t + \binom{6}{4}t^2 + \cdots$; hence W_5 in the reversion of $z = t - t^2$ is equal to $\binom{8}{4}/5 = 14$. This checks with the formulas for enumerating binary trees in Section 2.3.4.4.

Relation (12) shows that we can revert the series (10) if we compute the negative powers $(1+V_2t+V_3t^2+\cdots)^{-n}$ for $n = 1, 2, 3, \ldots$. A straightforward application of this idea would lead to an on-line reversion algorithm that uses approximately $N^3/2$ multiplications to find N coefficients, but Eq. (9) makes it possible to work with only the first n coefficients of $(1 + V_2t + V_3t^2 + \cdots)^{-n}$, obtaining an on-line algorithm that requires only about $N^3/6$ multiplications.

Algorithm L (*Lagrangian power series reversion*). This on-line algorithm inputs the value of V_n in (10) and outputs the value of W_n in (11), for $n = 2, 3, 4, \ldots, N$. (The number N need not be specified in advance; some other termination criterion may be substituted.)

L1. [Initialize.] Set $n \leftarrow 1$, $U_0 \leftarrow 1$. (The relation

$$(1 + V_2t + V_3t^2 + \cdots)^{-n} = U_0 + U_1t + \cdots + U_{n-1}t^{n-1} + O(t^n) \tag{13}$$

will be maintained throughout this algorithm.)

L2. [Input V_n.] Increase n by 1. If $n > N$, the algorithm terminates; otherwise input the next coefficient, V_n.

L3. [Divide.] Set $U_k \leftarrow U_k - U_{k-1}V_2 - \cdots - U_1V_k - V_{k+1}$, for $k = 1, 2, \ldots, n-2$ (in this order); then set

$$U_{n-1} \leftarrow -2U_{n-2}V_2 - 3U_{n-3}V_3 - \cdots - (n-1)U_1V_{n-1} - nV_n.$$

(We have thereby divided $U(z)$ by $V(z)/z$; cf. (3) and (9).)

L4. [Output W_n.] Output U_{n-1}/n (which is W_n) and return to L2. ∎

When applied to the example $z = t - t^2$, Algorithm L computes

n	V_n	U_0	U_1	U_2	U_3	U_4	W_n
1	1	1					1
2	−1	1	2				1
3	0	1	3	6			2
4	0	1	4	10	20		5
5	0	1	5	15	35	70	14

Exercise 8 shows that a slight modification of Algorithm L will solve a considerably more general problem with only a little more effort.

Let us now consider solving the equation

$$U_1z + U_2z^2 + U_3z^3 + \cdots = t + V_2t^2 + V_3t^3 + \cdots \tag{14}$$

for t, obtaining the coefficients of the power series

$$t = W_1z + W_2z^2 + W_3z^3 + W_4z^4 + \cdots. \tag{15}$$

Eq. (10) is the special case $U_1 = 1$, $U_2 = U_3 = \cdots = 0$. If $U_1 \neq 0$, we may assume that $U_1 = 1$, if we replace z by (U_1z); but we shall consider the general equation (14), since U_1 might equal zero.

Algorithm T (*General power series reversion*). This on-line algorithm inputs the values of U_n and V_n in (14) and outputs the value of W_n in (15), for $n = 1, 2, 3, \ldots, N$. An auxiliary matrix T_{mn}, $1 \leq m \leq n \leq N$, is used in the calculations.

T1. [Initialize.] Set $n \leftarrow 1$. Let the first two inputs (namely, U_1 and V_1) be stored in T_{11} and V_1, respectively. (We must have $V_1 = 1$.)

T2. [Output W_n.] Output the value of T_{1n} (which is W_n).

T3. [Input U_n, V_n.] Increase n by 1. If $n > N$, the algorithm terminates; otherwise store the next two inputs (namely, U_n and V_n) in T_{1n} and V_n.

T4. [Multiply.] Set

$$T_{mn} \leftarrow T_{11}T_{m-1,n-1} + T_{12}T_{m-1,n-2} + \cdots + T_{1,n-m+1}T_{m-1,m-1}$$

and $T_{1n} \leftarrow T_{1n} - V_mT_{mn}$, for $2 \leq m \leq n$. (After this step we have

$$t^m = T_{mm}z^m + T_{m,m+1}z^{m+1} + \cdots + T_{mn}z^n + O(z^{n+1}), \tag{16}$$

for $1 \leq m \leq n$. It is easy to verify (16) by induction for $m \geq 2$, and when $m = 1$, we have $U_n = T_{1n} + V_2T_{2n} + \cdots + V_nT_{nn}$ by (14) and (16).) Return to step T2. ∎

Equation (16) explains the mechanism of this algorithm, which is due to Henry C. Thacher, Jr. [*CACM* **9** (1966), 10–11]. The running time is essentially the same as Algorithm L, but considerably more storage space is required. An example of this algorithm is worked out in exercise 9.

Still another approach to power series reversion has been proposed by R. P. Brent and H. T. Kung [*JACM* **25** (1978), 581–595], based on the fact that standard iterative procedures used to find roots of equations over the real numbers can also be applied to equations over power series. In particular, we can consider Newton's method for computing approximations to a real number t such that $f(t) = 0$, given a function f that is well-behaved near t: If x is a good approximation to t, then $\phi(x) = x - f(x)/f'(x)$ will be even better, for if we write $x = t + \epsilon$ we have $f(x) = f(t) + \epsilon f'(t) + O(\epsilon^2)$, $f'(x) = f'(t) + O(\epsilon)$; consequently $\phi(x) = t + \epsilon - \big(0 + \epsilon f'(t) + O(\epsilon^2)\big)/\big(f'(t) + O(\epsilon)\big) = t + O(\epsilon^2)$. Applying this idea to power series, let $f(x) = V(x) - U(z)$, where U and V are the power series in Eq. (14). We wish to find the power series t in z such that $f(t) = 0$. Let $x = W_1 z + \cdots + W_{n-1} z^{n-1} = t + O(z^n)$ be an "approximation" to t of order n; then $\phi(x) = x - f(x)/f'(x)$ will be an approximation of order $2n$, since the assumptions of Newton's method hold for this f and t.

In other words, we can use the following procedure:

Algorithm N (*General power series reversion by Newton's method*). This "semi-on-line" algorithm inputs the values of U_n and V_n in (14) for $2^k \le n < 2^{k+1}$ and then outputs the values of W_n in (15) for $2^k \le n < 2^{k+1}$, thereby producing its answers in batches of 2^k at a time, for $k = 0, 1, 2, \ldots, K$.

N1. [Initialize.] Set $N \leftarrow 1$. (We will have $N = 2^k$.) Input the first coefficients U_1 and V_1 (where $V_1 = 1$), and set $W_1 \leftarrow U_1$.

N2. [Output.] Output W_n for $N \le n < 2N$.

N3. [Input.] Set $N \leftarrow 2N$. If $N > 2^K$, the algorithm terminates; otherwise input the values U_n and V_n for $N \le n < 2N$.

N4. [Newtonian step.] Use an algorithm for power series composition (see exercise 11) to evaluate the coefficients Q_j and R_j $(0 \le j < N)$ in the power series

$$\begin{aligned} U_1 z + \cdots + U_{2N-1} z^{2N-1} - V(W_1 z + \cdots + W_{N-1} z^{N-1}) \\ = R_0 z^N + R_1 z^{N+1} + \cdots + R_{N-1} z^{2N-1} + O(z^{2N}), \\ V'(W_1 z + \cdots + W_{N-1} z^{N-1}) = Q_0 + Q_1 z + \cdots + Q_{N-1} z^{N-1} + O(z^N), \end{aligned}$$

where $V(x) = x + V_2 x^2 + \cdots$ and $V'(x) = 1 + 2V_2 x + \cdots$. Then set $W_N, \ldots, W_{2N-1}$ to the coefficients in the power series

$$\frac{R_0 + R_1 z + \cdots + R_{N-1} z^{N-1}}{Q_0 + Q_1 z + \cdots + Q_{N-1} z^{N-1}} = W_N + \cdots + W_{2N-1} z^{N-1} + O(z^N)$$

and return to step N2. ∎

The running time for this algorithm to obtain the coefficients up to $N = 2^K$ is $T(N)$, where

$$T(2N) = T(N) + (\text{time to do step N4}) + O(N). \tag{17}$$

Straightforward algorithms for composition and division in step N4 will take order N^3 steps, so Algorithm N will run slower than Algorithm T. However, Brent and Kung have found a way to do the required composition of power series with $O(N \log N)^{3/2}$ arithmetic operations, and exercise 6 gives an even faster algorithm for division; hence (17) shows that power series reversion can be achieved by doing only $O(N \log N)^{3/2}$ operations as $N \to \infty$. (On the other hand the constant of proportionality is such that N must be really large before Algorithms L and T lose out to this "high-speed" method.)

Historical note: J. N. Bramhall and M. A. Chapple published the first $O(n^3)$ method for power series reversion in *CACM* **4** (1961), 317–318, 503. It was an off-line algorithm whose running time was approximately the same as that of Algorithms L and T.

Iteration of series. If we want to study the behavior of an iterative process $x_n \leftarrow f(x_{n-1})$, we are interested in studying the n-fold composition of a given function f with itself, namely $x_n = f(f(\ldots f(x_0)\ldots))$. Let us define $f^0(x) = x$ and $f^n(x) = f\big(f^{n-1}(x)\big)$, so that

$$f^{m+n}(x) = f^m\big(f^n(x)\big) \tag{18}$$

for all integers m, $n \geq 0$. It also makes sense to define $f^n(x)$ when n is a negative integer, namely to let f^n and f^{-n} be inverse functions such that $x = f^n\big(f^{-n}(x)\big)$; then (18) holds for *all* integers m and n. Reversion of series is essentially the operation of finding the inverse function $f^{-1}(x)$; for example, Eqs. (10) and (11) essentially state that $z = V\big(W(z)\big)$ and that $t = W\big(V(t)\big)$, so $W = V^{-1}$.

Suppose we are given two power series $V(z) = z + V_2 z^2 + \cdots$ and $W(z) = z + W_2 z^2 + \cdots$ such that $W = V^{-1}$. Let u be any nonzero constant, and consider the function

$$U(z) = W\big(uV(z)\big). \tag{19}$$

It is easy to see that $U\big(U(z)\big) = W\big(u^2 V(z)\big)$, and in general that

$$U^n(z) = W\big(u^n V(z)\big) \tag{20}$$

for all integers n. Therefore we have a simple expression for the nth iterate U^n, which can be calculated with roughly the same amount of work for all n. Furthermore, we can even use (20) to define U^n for non-integer values of n; the "half iterate" $U^{1/2}$, for example, is a function such that $U^{1/2}\big(U^{1/2}(z)\big) = U(z)$. (There are two such functions $U^{1/2}$, obtained by using $\sqrt{u}$ and $-\sqrt{u}$ as the value of $u^{1/2}$.)

We obtained the simple state of affairs in (20) by starting with V and u, then defining U. But in practice we generally want to go the other way: Starting with some given function U, we want to find V and u such that (19) holds, i.e., such that

$$V\big(U(z)\big) = u\,V(z). \tag{21}$$

Such a function V is called the *Schröder function* of U, because it was introduced by Ernst Schröder in *Math. Annalen* **3** (1871), 296–322. Let us now look at the problem of finding the Schröder function $V(z) = z + V_2 z^2 + \cdots$ of a given power series $U = U_1 z + U_2 z^2 + \cdots$. Clearly $u = U_1$ if (21) is to hold.

Expanding (21) with $u = U_1$ and equating coefficients of z leads to a sequence of equations that begins

$$\begin{aligned} U_1^2 V_2 + U_2 &= U_1 V_2 \\ U_1^3 V_3 + 2U_1 U_2 V_2 + U_3 &= U_1 V_3 \\ U_1^4 V_4 + 3U_1^2 U_2 V_3 + 2U_1 U_3 V_2 + U_2^2 V_2 + U_3 &= U_1 V_4 \end{aligned}$$

and so on. Clearly there is no solution when $U_1 = 0$ (unless trivially $U_2 = U_3 = \cdots = 0$); otherwise there is a unique solution unless U_1 is a root of unity. We might have expected that something funny would happen when $U_1^n = 1$, since Eq. (20) tells us that $U^n(z) = z$ if the Schröder function exists. For the moment let us assume that U_1 is nonzero and not a root of unity; then the Schröder function does exist, and the next question is how to compute it without doing too much work.

The following procedure has been suggested by R. P. Brent and J. F. Traub. Equation (21) leads to subproblems of a similar but more complicated form, so we set ourselves a more general task whose subtasks have the same form: Let us try to find $V(z) = V_0 + V_1 z + \cdots + V_{n-1} z^{n-1}$ such that

$$V\big(U(z)\big) = W(z)V(z) + S(z) + O(z^n), \tag{22}$$

given $U(z)$, $W(z)$, $S(z)$, and n, where n is a power of 2 and $U(0) = 0$. If $n = 1$ we simply let $V_0 = S(0)/\big(1 - W(0)\big)$, with $V_0 = 1$ if $S(0) = 0$ and $W(0) = 1$. Furthermore it is possible to go from n to $2n$: First we find $R(z)$ such that

$$V\big(U(z)\big) = W(z)V(z) + S(z) - z^n R(z) + O(z^{2n}). \tag{23}$$

Then we compute

$$\hat{W}(z) = W(z)\big(z/U(z)\big)^n, \qquad \hat{S}(z) = R(z)\big(z/U(z)\big)^n \tag{24}$$

and find $\hat{V}(z) = V_n + V_{n+1} z + \cdots + V_{2n-1} z^{n-1}$ such that

$$\hat{V}\big(U(z)\big) = \hat{W}(z)\hat{V}(z) + \hat{S}(z) + O(z^n). \tag{25}$$

It follows that $V^*\big(U(z)\big) = W(z)V^*(z) + S(z) + O(z^{2n})$ as desired, where $V^*(z)$ is equal to $V(z) + z^n \hat{V}(z)$.

The running time $T(n)$ of this procedure satisfies

$$T(2n) = 2T(n) + C(n), \tag{26}$$

where $C(n)$ is the time to compute $R(z)$, $\hat{W}(z)$, and $\hat{S}(z)$. The function $C(n)$ is dominated by the time to compute $V\big(U(z)\big)$ modulo z^{2n}, and $C(n)$ presumably grows faster than order $n^{1+\epsilon}$; therefore the solution $T(n)$ to (26) will be of order $C(n)$. For example, if $C(n) = cn^3$ we have $T(n) \approx \frac{4}{3}cn^3$; or if $C(n)$ is $O(n \log n)^{3/2}$ using "fast" composition, we have $T(n) = O(n \log n)^{3/2}$.

The procedure breaks down when $W(0) = 1$ and $S(0) \neq 0$, so we need to investigate when this can happen. It is easy to prove by induction on n that the solution of (22) by the Brent–Traub method entails consideration of exactly n subproblems, in which the coefficient of $V(z)$ on the right-hand side takes the respective values $W(z)\big(z/U(z)\big)^j$ for $0 \leq j < n$ in some order. If $W(0) = U_1$ and if U_1 is not a root of unity, we therefore have $W(0) = 1$ only when $j = 1$; the procedure will fail in this case only if (22) has no solution for $n = 2$.

The Schröder function for U can therefore be found by solving (22) for $n = 2, 4, 8, 16, \ldots$, with $W(z) = U_1$ and $S(z) = 0$, whenever U_1 is nonzero and not a root of unity.

If $U_1 = 1$, there is no Schröder function unless $U(z) = z$. But Brent and Traub have found a fast way to compute $U^n(z)$ even when $U_1 = 1$, by making use of a function $V(z)$ such that

$$V\big(U(z)\big) = U'(z)V(z). \tag{27}$$

If two functions $U(z)$ and $\hat{U}(z)$ both satisfy (27), for the same V, it is easy to check that their composition $U\big(\hat{U}(z)\big)$ does too; therefore all iterates of $U(z)$ are solutions of (27). Suppose that $U(z) = z + U_k z^k + U_{k+1}z^{k+1} + \cdots$ where $k \geq 2$ and $U_k \neq 0$. Then it can be shown that there is a unique power series of the form $V(z) = z^k + V_{k+1}z^{k+1} + V_{k+2}z^{k+2} + \cdots$ satisfying (27). Conversely if such a function $V(z)$ is given, and if $k \geq 2$ and U_k are given, then there is a unique power series of the form $U(z) = z + U_k z^k + U_{k+1}z^{k+1} + \cdots$ satisfying (27); the desired iterate $U^n(z)$ is the unique power series $P(z)$ satisfying

$$V\big(P(z)\big) = P'(z)V(z) \tag{28}$$

such that $P(z) = z + nU_k z^k + \cdots$. Both $V(z)$ and $P(z)$ can be found by appropriate algorithms (see exercise 14).

If U_1 is a kth root of unity, but not equal to 1, the same method can be applied to the function $U^k(z) = z + \cdots$, and $U^k(z)$ can be found from $U(z)$ by doing $l(k)$ composition operations (cf. Section 4.6.3). We can also handle the case $U_1 = 0$: If $U(z) = U_k z^k + U_{k+1}z^{k+1} + \cdots$ where $k \geq 2$ and $U_k \neq 0$, the idea is to find a solution to the equation $V\big(U(z)\big) = U_k V(z)^k$; then $U^n(z) = V^{-1}\big(U_k^{(k^n-1)/(k-1)}V(z)^{k^n}\big)$. Finally, if $U(z) = U_0 + U_1 z + \cdots$ where $U_0 \neq 0$, let α be a "fixed point" such that $U(\alpha) = \alpha$, and let $\hat{U}(z) = U(\alpha + z) - \alpha = zU'(\alpha) + z^2U''(\alpha)/2! + \cdots$; then $U^n(z) = \hat{U}^n(z - \alpha) + \alpha$. Further details can be found in Brent and Traub's paper [*SIAM J. Computing* **9** (1980), 54–66].

Algebraic functions. The coefficients of each power series $W(z)$ that satisfies a general equation of the form

$$A_n(z)W(z)^n + \cdots + A_1(z)W(z) + A_0(z) = 0, \tag{29}$$

where each $A_i(z)$ is a polynomial, can be computed efficiently by using methods due to H. T. Kung and J. F. Traub; see *JACM* **25** (1978), 245–260.

EXERCISES

1. [*M10*] The text explains how to divide $U(z)$ by $V(z)$ when $V_0 \neq 0$; how should the division be done when $V_0 = 0$?

2. [*20*] If the coefficients of $U(z)$ and $V(z)$ are integers and $V_0 \neq 0$, find a recurrence relation for the integers $V_0^{n+1}W_n$, where W_n is defined by (3). How would you use this for power series division?

3. [*M15*] Does formula (9) give the right results when $\alpha = 0$? When $\alpha = 1$?

▶ **4.** [*HM23*] Show that simple modifications of (9) can be used to calculate $e^{V(z)}$ and $\ln(1 + V(z))$, when $V(z) = V_1 z + V_2 z^2 + \cdots$.

5. [*M00*] What happens when a power series is reverted twice; i.e., if the output of Algorithm L or T is reverted again?

▶ **6.** [*M21*] (H. T. Kung.) Apply Newton's method to the computation of $W(z) = 1/V(z)$, when $V(0) \neq 0$, by finding the power series root of the equation $f(x) = 0$, where $f(x) = x^{-1} - V(z)$.

7. [*M23*] Use Lagrange's inversion formula (12) to find a simple expression for the coefficient W_n in the reversion of $z = t - t^m$.

▶ **8.** [*M25*] Lagrange's inversion formula can be generalized as follows: If $W(z) = W_1 z + W_2 z^2 + \cdots = G_1 t + G_2 t^2 + G_3 t^3 + \cdots = G(t)$, where z and t are related by Eq. (10), then $W_n = U_{n-1}/n$ where

$$U_0 + U_1 z + U_2 z^2 + \cdots = (G_1 + 2G_2 z + 3G_3 z^2 + \cdots)(1 + V_2 z + V_3 z^2 + \cdots)^{-n}.$$

(Equation (12) is the special case $G_1 = 1$, $G_2 = G_3 = \cdots = 0$. This equation can be proved, for example, by using tree-enumeration formulas as in exercise 2.3.4.4–33.) Extend Algorithm L so that it obtains the coefficients W_1, W_2, ... in this more general situation, without substantially increasing its running time.

9. [*11*] Find the values of T_{mn} computed by Algorithm T as it determines the first five coefficients in the reversion of $z = t - t^2$.

10. [*M20*] Given that $y = x^\alpha + a_1 x^{\alpha+1} + a_2 x^{\alpha+2} + \cdots$, $\alpha \neq 0$, show how to compute the coefficients in the expansion $x = y^{1/\alpha} + b_2 y^{2/\alpha} + b_3 y^{3/\alpha} + \cdots$.

▶ **11.** [*M25*] (*Composition of power series.*) Let

$$U(z) = U_0 + U_1 z + U_2 z^2 + \cdots \quad \text{and} \quad V(z) = V_1 z + V_2 z^2 + V_3 z^3 + \cdots;$$

design an algorithm that computes the first N coefficients of $U(V(z))$.

12. [*M20*] Find a connection between polynomial division and power series division: Given polynomials $u(x)$ and $v(x)$ of respective degrees m and n over a field, show how to find the polynomials $q(x)$, $r(x)$ such that $u(x) = q(x)v(x) + r(x)$ and $\deg(r) < n$, using only operations on power series.

13. [*M27*] (*Rational function approximation.*) It is occasionally desirable to find polynomials whose quotient has the same initial terms as a given power series. For example, if $W(z) = 1 + z + 3z^2 + 7z^3 + \cdots$, there are essentially four different ways to express $W(z)$ as $w_1(z)/w_2(z) + O(z^4)$ where $w_1(z)$ and $w_2(z)$ are polynomials with $\deg(w_1) + \deg(w_2) < 4$:

$$\begin{aligned}
(1 + z + 3z^2 + 7z^3)/1 &= 1 + z + 3z^2 + 7z^3 + 0z^4 + \cdots,\\
(3 - 4z + 2z^2)/(3 - 7z) &= 1 + z + 3z^2 + 7z^3 + \tfrac{49}{3}z^4 + \cdots,\\
(1 - z)/(1 - 2z - z^2) &= 1 + z + 3z^2 + 7z^3 + 17z^4 + \cdots,\\
1/(1 - z - 2z^2 - 2z^3) &= 1 + z + 3z^2 + 7z^3 + 15z^4 + \cdots.
\end{aligned}$$

Rational functions of this kind are commonly called *Padé approximations*, since they were studied extensively by H. E. Padé [*Annales Scient. de l'École Normale Supérieure* (3) **9** (1892), S1–S93].

Show that all Padé approximations $W(z) = w_1(z)/w_2(z) + O(z^N)$ with $\deg(w_1) + \deg(w_2) < N$ can be obtained by applying an extended Euclidean algorithm to the polynomials z^N and $W_0 + W_1 z + \cdots + W_{N-1} z^{N-1}$; and design an all-integer algorithm for the case that each W_i is an integer. [*Hint:* See exercise 4.6.1–26.]

▸ **14.** [*HM30*] Fill in the details of Brent and Traub's method for calculating $U^n(z)$ when $U(z) = z + U_k z^k + \cdots$, using (27) and (28).

And it shall be,
when thou hast made an end of reading this book,
that thou shalt bind a stone to it,
and cast it into the midst of Euphrates.

—Jeremiah 51:63

ANSWERS TO EXERCISES

This branch of mathematics is the only one, I believe,
in which good writers frequently get results entirely erroneous.
. . . It may be doubted if there is a single
extensive treatise on probabilities in existence
which does not contain solutions absolutely indefensible.

—C. S. PEIRCE, in *Popular Science Monthly* (1878)

NOTES ON THE EXERCISES

1. An average problem for a mathematically inclined reader.

3. See W. J. LeVeque, *Topics in Number Theory* **2** (Reading, Mass.: Addison–Wesley, 1956), Chapter 3. [*Note:* One of the men who read a preliminary draft of the manuscript of this book reported that he had discovered a truly remarkable proof, which the margin of his copy was too small to contain.]

SECTION 3.1

1. (a) This will usually fail, since "round" telephone numbers are often selected by the telephone user when possible. In some communities, telephone numbers are perhaps assigned randomly. But it would be a mistake in any case to try to get several successive random numbers from the same page, since the same telephone number is often listed several times in a row.

(b) But do you use the left-hand page or the right-hand page? Say, use the left-hand page number, divide by 2, and take the units digit. The total number of pages should be a multiple of 20; but even so, this method will have some bias.

(c) The markings on the faces will slightly bias the die, but for practical purposes this method is quite satisfactory (and it has been used by the author in the preparation of several examples in this set of books). See *Math. Comp.* **15** (1961), 94–95, for further discussion of these dice.

(d) (This is a hard question thrown in purposely as a surprise.) The number is not quite uniformly random. If the average number of emissions per minute is m, the probability that the counter registers k is $e^{-m}m^k/k!$ (the Poisson distribution); so the digit 0 is selected with probability $e^{-m}\sum_{k\geq 0} m^{10k}/(10k)!$, etc. The units digit will be even with probability $e^{-m}\cosh m = \frac{1}{2}+\frac{1}{2}e^{-2m}$, and this is never equal to $\frac{1}{2}$ (although the error is negligibly small when m is large).

(e) Okay, provided that the time since the previous digit selected in this way is random. However, there is possible bias in borderline cases.

(f,g) No, people usually think of certain digits (like 7) with higher probability.

(h) Okay; your assignment of numbers to the horses had probability $\frac{1}{10}$ of assigning a given digit to the winning horse.

2. The number of such sequences is the multinomial coefficient $1000000!/(100000!)^{10}$; the probability is this number divided by $10^{1000000}$, the total number of sequences of a million digits. By Stirling's approximation we find that the probability is close to $1/(16\pi^4 10^{22}\sqrt{2\pi}) \approx 2.55 \times 10^{-26}$, about one chance in 4×10^{25}.

3. 3040504030.

4. Step K11 can be entered only from step K10 or step K2, and in either case we find it impossible for X to be zero by a simple argument. If X could be zero at that point, the algorithm would not terminate.

5. Since only 10^{10} ten-digit numbers are possible, some value of X must be repeated during the first $10^{10}+1$ steps; and as soon as a value is repeated, the sequence continues to repeat its past behavior.

6. (a) Arguing as in the previous exercise, the sequence must eventually repeat a value; let this repetition occur for the first time at step $\mu+\lambda$, where $X_{\mu+\lambda} = X_\mu$. (This condition defines μ and λ.) We have $0 \leq \mu < m$, $0 < \lambda \leq m$, $\mu+\lambda \leq m$. The values $\mu = 0$, $\lambda = m$ are attained iff f is a cyclic permutation; and $\mu = m-1$, $\lambda = 1$ occurs, e.g., if $X_0 = 0$, $f(x) = x+1$ for $x < m-1$, and $f(m-1) = m-1$.

(b) We have, for $r \geq n$, $X_r = X_n$ iff $r-n$ is a multiple of λ and $n \geq \mu$. Hence $X_{2n} = X_n$ iff n is a multiple of λ and $n \geq \mu$. The desired results now follow immediately. [*Note:* This is essentially a proof of the familiar mathematical result that the powers of an element in a finite semigroup include a unique idempotent element: take $X_0 = a$, $f(x) = ax$.]

(c) Once n has been found, generate X_i and X_{n+i} for $i \geq 0$ until first finding $X_i = X_{n+i}$; then $\mu = i$. If none of the values of X_{n+i} for $0 \leq i \leq \mu$ is equal to X_n, it follows that $\lambda = n$, otherwise λ is the smallest such i.

7. (a) The least $n > 0$ such that $n - (\ell(n)-1)$ is a multiple of λ and $\ell(n)-1 \geq \mu$ is $n = 2^{\lceil \lg \max(\mu+1,\lambda)\rceil} - 1 + \lambda$. [This may be compared with the least $n > 0$ such that $X_{2n} = X_n$, namely $\lambda\delta_{\mu 0} + \mu + \lambda - 1 - ((\mu+\lambda-1) \bmod \lambda)$.]

(b) Start with $X = Y = X_0$, $k = m = 1$. (At key places in this algorithm we will have $X = X_{2m-k-1}$, $Y = X_{m-1}$, and $m = \ell(2m-k)$.) To generate the next random number, do the following steps: Set $X \leftarrow f(X)$ and $k \leftarrow k-1$. If $X = Y$, stop (the period length λ is equal to $m-k$). Otherwise if $k = 0$, set $Y \leftarrow X$, $m \leftarrow 2m$, $k \leftarrow m$. Output X.

Notes: Brent has also considered a more general method in which the successive values of $Y = X_{n_i}$ satisfy $n_1 = 0$, $n_{i+1} = 1 + \lfloor pn_i \rfloor$ where p is any number greater than 1. He showed that the best choice of p, approximately 2.4771, saves about 3 percent of the iterations by comparison with $p = 2$. (See exercise 4.5.4–4.)

The method in part (b) has a serious deficiency, however, since it might generate a lot of nonrandom numbers before shutting off. For example, we might have a particularly bad case such as $\lambda = 1$, $\mu = 2^k$. A method based on Floyd's idea in exercise 6(b), namely one that maintains $Y = X_{2n}$ and $X = X_n$ for $n = 0, 1, 2, \ldots$, will require a few more function evaluations than Brent's method, but it will stop before any number has been output twice.

On the other hand if f is unknown (e.g., if we are receiving the values $X_0, X_1, \ldots$ on-line from an outside source) or if f is difficult to apply, the following cycle detection algorithm due to R. W. Gosper will be preferable: Maintain an auxiliary table $T_0, T_1, \ldots, T_m$, where $m = \lfloor \lg n \rfloor$ when receiving X_n. Initially $T_0 \leftarrow X_0$. For $n = 1, 2, \ldots$, compare X_n with each of $T_0, \ldots, T_{\lfloor \lg n \rfloor}$; if no match is found, set $T_{e(n)} \leftarrow X_n$, where $e(n) = \max\{\, e \mid 2^e \text{ divides } n + 1 \,\}$. But if a match $X_n = T_k$ is found, then $\lambda = n - \max\{\, m \mid m < n \text{ and } e(m) = k \,\}$. This procedure does not stop before generating a number twice, but at most $\lceil \frac{2}{3}\lambda \rceil$ of the X's will have occurred more than once. [MIT AI Laboratory Memo 239 (Feb. 29, 1972), Hack 132.]

See also R. Sedgewick and T. G. Szymanski, *Proc. ACM Symp. Th. Comp.* **11** (1979), 74–80.

8. (a,b) $00, 00, \ldots$ [62 starting values]; $10, 10, \ldots$ [19]; $60, 60, \ldots$ [15]; $50, 50, \ldots$ [1]; $24, 57, 24, 57, \ldots$ [3]. (c) 42 or 69; these both lead to a set of fifteen distinct values, namely (42 or 69), 76, 77, 92, 46, 11, 12, 14, 19, 36, 29, 84, 05, 02, 00.

9. Since $X < b^n$, we have $X^2 < b^{2n}$, and the middle square is $\lfloor X^2/b^n \rfloor \le X^2/b^n$. If $X > 0$, then $X^2/b^n < Xb^n/b^n = X$.

10. If $X = ab^n$, the next number of the sequence has the same form; it is $(a^2 \bmod b^n)b^n$. If a is a multiple of b or of all the prime factors of b, the sequence will soon degenerate to zero; if not, the sequence will degenerate into a cycle of numbers having the same general form as X.

Further facts about the middle-square method have been found by B. Jansson, *Random Number Generators* (Stockholm: Almqvist & Wiksell, 1966), Section 3A. Numerologists will be interested to learn that the number 3792 is self-reproducing in the four-digit middle-square method, since $3792^2 = 14379264$; furthermore (as Jansson has observed), it is "self-reproducing" in another sense, too, since its prime factorization is $3 \cdot 79 \cdot 2^4$!

11. The probability that $\lambda = 1$ and $\mu = 0$ is the probability that $X_1 = X_0$, namely $1/m$. The probability that $\lambda = 1, \mu = 1$, or $\lambda = 2, \mu = 0$, is the probability that $X_1 \neq X_0$ and that X_2 has a certain value, so it is $(1 - 1/m)(1/m)$. Similarly, the probability that the sequence has any given μ and λ is a function only of $\mu + \lambda$, namely

$$P(\mu, \lambda) = \frac{1}{m} \prod_{1 \le k < \mu + \lambda} \left(1 - \frac{k}{m}\right).$$

For the probability that $\lambda = 1$, we have

$$\sum_{\mu \ge 0} \frac{1}{m} \prod_{1 \le k \le \mu} \left(1 - \frac{k}{m}\right) = \frac{1}{m} Q(m),$$

where $Q(m)$ is defined in Section 1.2.11.3, Eq. (2). By Eq. (25) in that section, the probability is approximately $6\sqrt{\pi/2m} \approx 1.25/\sqrt{m}$. The chance of Algorithm K converging as it did is only about one in 80000; the author was decidedly unlucky. But see exercise 15 for further comments on the "colossalness."

12. $$\sum_{\substack{1\le\lambda\le m\\ 0\le\mu<m}} \lambda P(\mu,\lambda) = \frac{1}{m}\Big(1+3\Big(1-\frac{1}{m}\Big)+6\Big(1-\frac{1}{m}\Big)\Big(1-\frac{2}{m}\Big)+\cdots\Big)$$
$$= \frac{1+Q(m)}{2}.$$

(See the previous answer. In general if $f(a_0,a_1,\dots) = \sum_{n\ge0} a_n \prod_{1\le k\le n}(1-k/m)$ then $f(a_0,a_1,\dots) = a_0 + f(a_1,a_2,\dots) - f(a_1,2a_2,\dots)/m$; apply this identity with $a_n = (n+1)/2$.) Therefore the average value of λ (and, by symmetry of $P(\mu,\lambda)$, also of $\mu+1$) is approximately $\sqrt{\pi m/8}+\frac{1}{3}$. The average value of $\mu+\lambda$ is exactly $Q(m)$, approximately $\sqrt{\pi m/2}-\frac{1}{3}$. [For alternative derivations and further results, including asymptotic values for the moments, see A. Rapoport, *Bull. Math. Biophysics* **10** (1948), 145–157, and B. Harris, *Annals Math. Stat.* **31** (1960), 1045–1062; see also I. M. Sobol', *Theory of Probability and its Applications* **9** (1964), 333–338. Sobol' discusses the asymptotic period length for the more general sequence $X_{n+1} = f(X_n)$ if $n \not\equiv 0$ (modulo m); $X_{n+1} = g(X_n)$ if $n \equiv 0$ (modulo m); with both f and g random.]

13. [Paul Purdom and John Williams, *Trans. Amer. Math. Soc.* **133** (1968), 547–551.] Let T_{mn} be the number of functions that have n one-cycles and no cycles of length greater than one. Then

$$T_{mn} = \binom{m-1}{n-1} m^{m-n}.$$

(See Section 2.3.4.4.) *Any* function is such a function followed by a permutation of the n elements that were the one-cycles. Hence $\sum_{n\ge1} T_{mn}\, n! = m^m$.

Let P_{nk} be the number of permutations of n elements in which the longest cycle is of length k. Then the number of functions with a maximum cycle of length k is $\sum_{n\ge1} T_{mn}P_{nk}$. To get the average value of k, we compute $\sum_{k\ge1}\sum_{n\ge1} kT_{mn}P_{nk}$, which by the result of exercise 1.3.3–23 is $\sum_{n\ge1} T_{mn}\, n!(n+\frac{1}{2}+\epsilon_n)c$ where $c \approx .62433$ and $\epsilon_n \to 0$ as $n\to\infty$. Summing, we get the average value $cQ(m) + \frac{1}{2}c + \delta_m$, where $\delta_m \to 0$ as $m\to\infty$. (This is not substantially larger than the average value when X_0 is selected at random. The average value of $\max(\mu+\lambda)$ is still unknown.)

14. Let $c_r(m)$ be the number of functions with exactly r different final cycles. From the recurrence $c_1(m) = (m-1)! - \sum_{k>0}\binom{m}{k}(-1)^k(m-k)^k c_1(m-k)$, which comes by counting the number of functions whose image contains at most $m-k$ elements, we find the solution $c_1(m) = m^{m-1}Q(m)$. (Cf. exercise 1.2.11.3–16.) Another way to obtain the value of $c_1(m)$, which is perhaps more elegant and revealing, is given in exercise 2.3.4.4–17. The value of $c_r(m)$ may be determined by solving the recurrence

$$c_r(m) = \sum_{0<k<m} \binom{m-1}{k-1} c_1(k)c_{r-1}(m-k),$$

which has the solution

$$c_r(m) = m^{m-1}\Big(\frac{1}{0!}\begin{bmatrix}1\\r\end{bmatrix} + \frac{1}{1!}\begin{bmatrix}2\\r\end{bmatrix}\frac{m-1}{m} + \frac{1}{2!}\begin{bmatrix}3\\r\end{bmatrix}\frac{m-1}{m}\frac{m-2}{m} + \cdots\Big).$$

The desired average value can now be computed; it is (cf. exercise 12)

$$E_m = \frac{1}{m}\Big(H_1 + 2H_2\frac{m-1}{m} + 3H_3\frac{m-1}{m}\frac{m-2}{m} + \cdots\Big)$$
$$= 1 + \frac{1}{2}\frac{m-1}{m} + \frac{1}{3}\frac{m-1}{m}\frac{m-2}{m} + \cdots.$$

This latter formula was obtained by quite different means by Martin D. Kruskal, *AMM* **61** (1954), 392–397. Using the integral representation

$$E_m = \int_0^\infty \Big(\Big(1+\frac{x}{m}\Big)^m - 1\Big)e^{-x}\,\frac{dx}{x},$$

he proved the asymptotic relation $\lim_{m\to\infty}(E_m - \frac{1}{2}\ln m) = \frac{1}{2}(\gamma + \ln 2)$. For further results and references, see John Riordan, *Annals Math. Stat.* **33** (1962), 178–185.

15. The probability that $f(x) \neq x$ for all x is $(m-1)^m/m$, which is approximately $1/e$. The existence of a self-repeating value in an algorithm like Algorithm K is therefore not "colossal" at all—it occurs with probability $1 - 1/e \approx .63212$. The only "colossal" thing was that the author happened to hit such a value when X_0 was chosen at random (see exercise 11).

16. The sequence will repeat when a pair of successive elements occurs for the second time. The maximum period is m^2. (Cf. next exercise.)

17. After selecting $X_0, \ldots, X_{k-1}$ arbitrarily, let $X_{n+1} = f(X_n, \ldots, X_{n-k+1})$, where $0 \le x_1, \ldots, x_k < m$ implies that $0 \le f(x_1, \ldots, x_k) < m$. The maximum period is m^k. This is an obvious upper bound, but it is not obvious that it can be attained; for a proof that it can always be attained for suitable f, see exercises 3.2.2–17 and 3.2.2–21, and for the number of ways to attain it see exercise 2.3.4.2–23.

18. Same as exercise 7, but use the k-tuple of elements $(X_n, \ldots, X_{n-k+1})$ in place of the single element X_n.

SECTION 3.2.1

1. Take X_0 even, a even, c odd. Then X_n is odd for $n > 0$.

2. Let X_r be the first repeated value in the sequence. If X_r were equal to X_k for some k where $0 < k < r$, we could prove that $X_{r-1} = X_{k-1}$, since X_n uniquely determines X_{n-1} when a is prime to m. Hence $k = 0$.

3. If d is the greatest common divisor of a and m, the quantity aX_n can take on at most m/d values. The situation can be even worse; e.g., if $m = 2^e$ and if a is even, Eq. (6) shows that the sequence is eventually constant.

4. Induction on k.

5. If a is relatively prime to m, there is a number a' for which $aa' \equiv 1$ (modulo m). Then $X_{n-1} = (a'X_n - a'c) \bmod m$, and in general,

$$X_{n-k} = \big((a')^k X_n - c(a' + \cdots + (a')^k)\big) \bmod m$$
$$= \left((a')^k X_n - c\frac{(a')^{k+1} - a'}{a' - 1}\right) \bmod m$$

when $k > 0$, $n - k \ge 0$. If a is not relatively prime to m, it is not possible to determine X_{n-1} when X_n is given; multiples of $m/\gcd(a, m)$ may be added to X_{n-1} without changing the value of X_n. (See also exercise 3.2.1.3–7.)

SECTION 3.2.1.1

1. Let c' be a solution to the congruence $ac' \equiv c$ (modulo m). (Thus, $c' = a'c \bmod m$, if a' is the number in the answer to exercise 3.2.1–5.) Then we have

```
LDA  X
ADD  CPRIME
MUL  A        ∎
```

Overflow is possible on this addition operation. (From results derived later in the chapter, it is probably best to save a unit of time, taking $c = a$ and replacing the ADD instruction by "INCA 1". Then if $X_0 = 0$, overflow will not occur until the end of the period, so it won't occur in practice.)

2.
```
RANDM STJ  1F
      LDA  XRAND
      MUL  A
      SLAX 5
      ADD  C      (or, INCA c, if c is small)
      STA  XRAND
1H    JNOV *
      JMP  *-1
XRAND CON  X0
A     CON  a
C     CON  c      ∎
```

Note: Locations A and C should probably be named 2H and 3H to avoid conflict with other symbols, if this subroutine is to be used by other programmers.

3. See WM1 at the end of Program 4.2.3D.

4. Define the operation $x \underline{\bmod} 2^e = y$ iff $x \equiv y$ (modulo 2^e) and $-2^{e-1} \le y < 2^{e-1}$. The congruential sequence $\langle Y_n \rangle$ defined by

$$Y_0 = X_0 \underline{\bmod} 2^{32}, \qquad Y_{n+1} = (aY_n + c) \underline{\bmod} 2^{32}$$

is easy to compute on 370-style machines, since the lower half of the product of y and z is $(yz) \underline{\bmod} 2^{32}$ for all two's complement numbers y and z, and since addition ignoring overflow also delivers its result $\underline{\bmod} 2^{32}$. This sequence has all the randomness properties of the standard linear congruential sequence $\langle X_n \rangle$, since $Y_n \equiv X_n$ (modulo 2^{32}). Indeed, the two's complement representation of Y_n is *identical* to the binary representation of X_n, for all n. [G. Marsaglia and T. A. Bray first pointed this out in *CACM* **11** (1968), 757–759.]

5. (a) Subtraction: LDA X; SUB Y; JANN *+2; ADD M. (b) Addition: LDA X; SUB M; ADD Y; JANN *+2; ADD M. (Note that if m is more than half the word size, the instruction "SUB M" must precede the instruction "ADD Y".)

6. The sequences are not essentially different, since adding the constant $(m - c)$ has the same effect as subtracting the constant c. The operation must be combined with multiplication, so a subtractive process has little merit over the additive one (at least in MIX's case), except when it is necessary to avoid affecting the overflow toggle.

7. The prime factors of $z^k - 1$ appear in the factorization of $z^{kr} - 1$. If r is odd, the prime factors of $z^k + 1$ appear in the factorization of $z^{kr} + 1$. And $z^{2k} - 1$ equals $(z^k - 1)(z^k + 1)$.

8.
```
JOV  *+1    (Ensure that overflow is off.)
LDA  X
MUL  A
STX  TEMP
ADD  TEMP   Add lower half to upper half.
JNOV *+2    If ≥ w, subtract w − 1.
INCA 1      (Overflow is impossible in this step.)
```
∎

Note: Since addition on an e-bit ones'-complement computer is mod $(2^e - 1)$, it is possible to combine the techniques of exercises 4 and 8, producing $(yz) \bmod (2^e - 1)$ by adding together the two e-bit halves of the product yz, for all ones' complement numbers y and z regardless of sign.

9. The pairs $(0, w - 2)$, $(1, w - 1)$ are treated as equivalent in input and output:

```
JOV  *+1
LDA  X
MUL  A      aX = qw + r.
SLC  5      rA ← r, rX ← q.
STX  TEMP
ADD  TEMP
JNOV *+2    Get (r + q) mod (w − 2).
INCA 2      Overflow is impossible.
ADD  TEMP
JNOV *+3    Get (r + 2q) mod (w − 2).
INCA 2      Overflow is possible.
JOV  *-1
```
∎

SECTION 3.2.1.2

1. Period length m, by Theorem A. (Cf. exercise 3.)

2. Yes, these conditions imply the conditions in Theorem A, since the only prime divisor of 2^e is 2, and any odd number is relatively prime to 2^e. (In fact, the conditions of the exercise are *necessary* and sufficient.)

3. By Theorem A, we need $a \equiv 1$ (modulo 4) and $a \equiv 1$ (modulo 5). By Law D of Section 1.2.4, this is equivalent to $a \equiv 1$ (modulo 20).

4. We know $X_{2^{e-1}} \equiv 0$ (modulo 2^{e-1}) by using Theorem A in the case $m = 2^{e-1}$. Also using Theorem A for $m = 2^e$, we know that $X_{2^{e-1}} \not\equiv 0$ (modulo 2^e). It follows that $X_{2^{e-1}} = 2^{e-1}$. More generally, we can use Eq. 3.2.1–6 to prove that the second half of the period is essentially like the first half, since $X_{n+2^{e-1}} = (X_n + 2^{e-1}) \bmod 2^e$. (The quarters are similar too, see exercise 21.)

5. We need $a \equiv 1$ (modulo p) for $p = 3, 11, 43, 281, 86171$. By Law D of Section 1.2.4, this is equivalent to $a \equiv 1$ (modulo $3 \cdot 11 \cdot 43 \cdot 281 \cdot 86171$), so the *only* solution is the terrible multiplier $a = 1$.

6. (Cf. previous exercise.) $a \equiv 1$ (modulo $3 \cdot 7 \cdot 11 \cdot 13 \cdot 37$) implies that the solutions are $a = 1 + 111111k$, for $0 \le k \le 8$.

7. Using the notation of the proof of Lemma Q, μ is the smallest value such that $X_{\mu+\lambda} = X_\mu$; so it is the smallest value such that $Y_{\mu+\lambda} = Y_\mu$ and $Z_{\mu+\lambda} = Z_\mu$. This shows that $\mu = \max(\mu_1, \ldots, \mu_t)$. The highest achievable μ is $\max(e_1, \ldots, e_t)$, but nobody really wants to achieve it.

8. We have $a^2 \equiv 1$ (modulo 8); so $a^4 \equiv 1$ (modulo 16), $a^8 \equiv 1$ (modulo 32), etc. If $a \bmod 4 = 3$, then $a-1$ is twice an odd number; so $(a^{2^{e-1}}-1)/2 \equiv 0$ (modulo $2^{e+1}/2$), and this yields the desired result.

9. Substitute for X_n in terms of Y_n and simplify. If $X_0 \bmod 4 = 3$, the formulas of the exercise do not apply; but they do apply to the sequence $Z_n = (-X_n) \bmod 2^e$, which has essentially the same behavior.

10. Only $m = 1$, 2, 4, p^e, and $2p^e$, for odd primes p. In all other cases, the result of Theorem B is an improvement over Euler's theorem (exercise 1.2.4–28).

11. (a) Either $x+1$ or $x-1$ (not both) will be a multiple of 4, so $x \mp 1 = q2^f$, where q is odd and f is greater than 1. (b) In the given circumstances, $f < e$ and so $e \ge 3$. We have $\pm x \equiv 1$ (modulo 2^f) and $\pm x \not\equiv 1$ (modulo 2^{f+1}) and $f > 1$. Hence by applying Lemma P, we find that $(\pm x)^{2e-f-1} \not\equiv 1$ (modulo 2^e), while $x^{2e-f} = (\pm x)^{2e-f} \equiv 1$ (modulo 2^e). So the order is a divisor of 2^{e-f}, but not a divisor of 2^{e-f-1}. (c) 1 has order 1; $2^e - 1$ has order 2; the maximum period when $e \ge 3$ is therefore 2^{e-2}, and for $e \ge 4$ it is necessary to have $f = 2$, that is, $x \equiv 4 \pm 1$ (modulo 8).

12. If k is a proper divisor of $p-1$ and if $a^k \equiv 1$ (modulo p), then by Lemma P we have $a^{kp^{e-1}} \equiv 1$ (modulo p^e). Similarly, if $a^{p-1} \equiv 1$ (modulo p^2), we find that $a^{(p-1)p^{e-2}} \equiv 1$ (modulo p^e). So in these cases a is *not* primitive. Conversely, if $a^{p-1} \not\equiv 1$ (modulo p^2), Theorem 1.2.4F and Lemma P tell us that $a^{(p-1)p^{e-2}} \not\equiv 1$ (modulo p^e), but $a^{(p-1)p^{e-1}} \equiv 1$ (modulo p^e). So the order is a divisor of $(p-1)p^{e-1}$ but not of $(p-1)p^{e-2}$; it therefore has the form kp^{e-1}, where k divides $p-1$. But if a is primitive modulo p, the congruence $a^{kp^{e-1}} \equiv a^k \equiv 1$ (modulo p) implies that $k = p-1$.

13. Let λ be the order of a modulo p. By Theorem 1.2.4F, λ is a divisor of $p-1$. If $\lambda < p-1$, then $(p-1)/\lambda$ has a prime factor, q.

14. Let $0 < k < p$. If $a^{p-1} \equiv 1$ (modulo p^2), then $(a+kp)^{p-1} \equiv a^{p-1} + (p-1) \times a^{p-2}kp$ (modulo p^2); and this is $\not\equiv 1$, since $(p-1)a^{p-2}k$ is not a multiple of p. By exercise 12, $a + kp$ is primitive modulo p^e.

15. (a) If $\lambda_1 = p_1^{e_1} \ldots p_t^{e_t}$, $\lambda_2 = p_1^{f_1} \ldots p_t^{f_t}$, let $\kappa_1 = p_1^{g_1} \ldots p_t^{g_t}$, $\kappa_2 = p_1^{h_1} \ldots p_t^{h_t}$, where

$$\begin{aligned} g_j = e_j \quad &\text{and} \quad h_j = 0, \qquad \text{if} \quad e_j < f_j, \\ g_j = 0 \quad &\text{and} \quad h_j = f_j, \qquad \text{if} \quad e_j \ge f_j. \end{aligned}$$

Now $a_1^{\kappa_1}, a_2^{\kappa_2}$ have periods $\lambda_1/\kappa_1, \lambda_2/\kappa_2$, and the latter are relatively prime. Furthermore $(\lambda_1/\kappa_1)(\lambda_2/\kappa_2) = \lambda$, so it suffices to consider the case when λ_1 is relatively prime to λ_2, that is, when $\lambda = \lambda_1\lambda_2$. Now since $(a_1a_2)^\lambda \equiv 1$, we have $1 \equiv (a_1a_2)^{\lambda\lambda_1} \equiv a_2^{\lambda\lambda_1}$; hence $\lambda\lambda_1$ is a multiple of λ_2. This implies that λ is a multiple of λ_2, since λ_1 is relatively prime to λ_2. Similarly, λ is a multiple of λ_1; hence λ is a multiple of $\lambda_1\lambda_2$. But obviously $(a_1a_2)^{\lambda_1\lambda_2} \equiv 1$, so $\lambda = \lambda_1\lambda_2$.

(b) If a_1 has order $\lambda(m)$ and if a_2 has order λ, by part (a) $\lambda(m)$ must be a multiple of λ, otherwise we could find an element of higher order, namely of order $\operatorname{lcm}(\lambda, \lambda(m))$.

16. (a) $f(x) = (x-a)(x^{n-1} + (a+c_1)x^{n-2} + \cdots + (a^{n-1} + \cdots + c_{n-1})) + f(a)$. (b) The statement is clear when $n = 0$. If a is one root, $f(x) \equiv (x-a)q(x)$; therefore if a' is any other root,

$$0 \equiv f(a') \equiv (a'-a)q(a'),$$

and since $a'-a$ is not a multiple of p, a' must be a root of $q(x)$. So if $f(x)$ has more than n distinct roots, $q(x)$ has more than $n-1$ distinct roots. (c) $\lambda(p) \geq p-1$, since $f(x)$ must have degree $\geq p-1$ in order to possess so many roots. But by Theorem 1.2.4F, $\lambda(p) \leq p-1$.

17. By Lemma P, $11^5 \equiv 1$ (modulo 25), $11^5 \not\equiv 1$ (modulo 125), etc.; so the order of 11 is 5^{e-1} (modulo 5^e), not the maximum value $\lambda(5^e) = 4 \cdot 5^{e-1}$. But by Lemma Q the total period length is the least common multiple of the period modulo 2^e (namely 2^{e-2}) and the period modulo 5^e (namely 5^{e-1}), and this is $2^{e-2}5^{e-1} = \lambda(10^e)$. The period modulo 5^e may be 5^{e-1} or $2 \cdot 5^{e-1}$ or $4 \cdot 5^{e-1}$, without affecting the length of period modulo 10^e, since the least common multiple is taken. The values that are primitive modulo 5^e are those congruent to 2, 3, 8, 12, 13, 17, 22, 23 modulo 25 (cf. exercise 12), namely 3, 13, 27, 37, 53, 67, 77, 83, 117, 123, 133, 147, 163, 173, 187, 197.

18. According to Theorem C, $a \bmod 8$ must be 3 or 5. Knowing the period of a modulo 5 and modulo 25 allows us to apply Lemma P to determine admissible values of a mod 25. Period $= 4 \cdot 5^{e-1}$: 2, 3, 8, 12, 13, 17, 22, 23; period $= 2 \cdot 5^{e-1}$: 4, 9, 14, 19; period $= 5^{e-1}$: 6, 11, 16, 21. Each of these 16 values yields one value of a, $0 \leq a < 200$, with $a \bmod 8 = 3$, and another value of a with $a \bmod 8 = 5$.

19. One example appears in Table 3.3.4–1, line 26.

20. (a) We have $AY_n + X_0 \equiv AY_{n+k} + X_0$ (modulo m) iff $Y_n \equiv Y_{n+k}$ (modulo m'). (b)(i) Obvious. (ii) Theorem A. (iii) $(a^n-1)/(a-1) \equiv 0$ (modulo 2^e) iff $a^n \equiv 1$ (modulo 2^{e+1}); if $a \not\equiv -1$, the order of a modulo 2^{e+1} is twice its order modulo 2^e. (iv) $(a^n-1)/(a-1) \equiv 0$ (modulo p^e) iff $a^n \equiv 1$.

21. $X_{n+s} \equiv X_n + X_s$ by Eq. 3.2.1–6; and s is a divisor of m, since s is a power of p when m is a power of p. Hence a given integer q is a multiple of m/s iff $X_{qs} \equiv 0$ iff q is a multiple of $m/\gcd(X_s, m)$.

SECTION 3.2.1.3

1. $c = 1$ is always relatively prime to B^5; and every prime dividing $m = B^5$ is a divisor of B, so it divides $b = B^2$ to at least the second power.

2. Only 3, so the generator is not recommended in spite of its long period.

3. The potency is 18 in both cases (see next exercise).

4. Since $a \bmod 4 = 1$, we must have $a \bmod 8 = 1$ or 5, so $b \bmod 8 = 0$ or 4. If b is an odd multiple of 4, and if b_1 is a multiple of 8, clearly $b^s \equiv 0$ (modulo 2^e) implies that $b_1^s \equiv 0$ (modulo 2^e), so b_1 cannot have higher potency.

5. The potency is the smallest value of s such that $f_j s \geq e_j$ for all j.

6. The modulus must be divisible by 2^7 or by p^4 (for odd prime p) in order to have a potency as high as 4. The only values are $m = 2^{27} + 1$ and $10^9 - 1$.

7. $a' = (1 - b + b^2 - \cdots) \bmod m$, where the terms in b^s, b^{s+1}, etc., are dropped (if s is the potency).

8. Since X_n is always odd,

$$X_{n+2} = (2^{34} + 3 \cdot 2^{18} + 9)X_n \bmod 2^{35} = (2^{34} + 6X_{n+1} - 9X_n) \bmod 2^{35}.$$

Given Y_n and Y_{n+1}, the possibilities for

$$Y_{n+2} \approx (5 + 6(Y_{n+1} + \epsilon_1) - 9(Y_n + \epsilon_2)) \bmod 10,$$

with $0 \le \epsilon_1 < 1$, $0 \le \epsilon_2 < 1$, are limited and nonrandom.

Note: If the multiplier suggested in exercise 3 were, say, $2^{33} + 2^{18} + 2^2 + 1$, instead of $2^{23} + 2^{14} + 2^2 + 1$, we would similarly find $X_{n+2} - 10X_{n+1} + 25X_n \equiv$ constant (modulo 2^{35}). In general, we do not want $a \pm \delta$ to be divisible by high powers of 2 when δ is small, else we get "second order impotency." See Section 3.3.4 for a more detailed discussion.

The generator that appears in this exercise is discussed in an article by MacLaren and Marsaglia, *JACM* **12** (1965), 83–89. The deficiencies of such generators were first demonstrated by M. Greenberger, *CACM* **8** (1965), 177–179. Yet generators like this were still in widespread use more than ten years later (cf. the discussion of RANDU in Section 3.3.4).

SECTION 3.2.2

1. The method is useful only with great caution. In the first place, aU_n is likely to be so large that the addition of c/m that follows will lose almost all significance, and the "mod 1" operation will nearly destroy any vestiges of significance that might remain. We conclude that double-precision floating point arithmetic is necessary. Even with double precision, one must be sure that no rounding, etc., occurs to affect the numbers of the sequence in any way, since that would destroy the theoretical grounds for the good behavior of the sequence. (But see exercise 23.)

2. X_{n+1} equals either $X_{n-1} + X_n$ or $X_{n-1} + X_n - m$. If $X_{n+1} < X_n$ we must have $X_{n+1} = X_{n-1} + X_n - m$; hence $X_{n+1} < X_{n-1}$.

3. (a) The underlined numbers are $V[j]$ after step M3.

Output: initial		0	4	5	6	2	0	3	(2	7	4	1	6	3	0	5)	and repeats.
$V[0]$:	0	$\underline{4}$	$\underline{7}$	7	7	7	7	7	7	$\underline{4}$	$\underline{7}$	7	7	7	7	7	7	$\underline{4}$	$\underline{7}$ ...
$V[1]$:	3	3	3	3	3	3	3	$\underline{2}$	$\underline{5}$	5	5	5	5	5	5	$\underline{2}$	$\underline{5}$	5	5 ...
$V[2]$:	2	2	2	2	2	$\underline{0}$	$\underline{3}$	3	3	3	3	3	3	$\underline{0}$	$\underline{3}$	3	3	3	3 ...
$V[3]$:	5	5	5	$\underline{6}$	$\underline{1}$	1	1	1	1	1	1	$\underline{6}$	$\underline{1}$	1	1	1	1	1	1 ...
X:		4	7	6	1	0	3	2	5	4	7	6	1	0	3	2	5	4	7 ...
Y:		0	1	6	7	4	5	2	3	0	1	6	7	4	5	2	3	0	1 ...

So the potency has been reduced to 1! (See further comments in the answer to exercise 15.)

(b) The underlined numbers are $V[j]$ after step B2.

Output: initial		2 3 6 5 7 0 0 5 3 ... 4 6(3 0 ... 4 7)...
$V[0]$:	0	0 0 0 0 0 0 $\underline{5}$ $\underline{4}$ 4 ... 1 1 1 1 ... 1 1 ...
$V[1]$:	3	3 $\underline{6}$ $\underline{1}$ 1 1 1 1 1 1 ... 0 0 0 $\underline{4}$ ... 0 0 ...
$V[2]$:	2	$\underline{7}$ 7 7 7 $\underline{3}$ 3 3 3 $\underline{7}$ ... 6 $\underline{2}$ 2 2 ... 7 $\underline{2}$ ...
$V[3]$:	5	5 5 5 $\underline{0}$ 0 $\underline{2}$ 2 2 2 ... $\underline{3}$ 3 $\underline{5}$ 5 ... $\underline{3}$ 3 ...
X:	4	7 6 1 0 3 2 5 4 7 ... 3 2 5 4 ... 3 2 ...

In this case the output is considerably better than the input; it enters a repeating cycle of length 40 after 46 steps: 236570 05314 72632 40110 37564 76025 12541 73625 03746 (30175 24061 52317 46203 74531 60425 16753 02647). The cycle can be found easily by applying the method of exercise 3.1–7 to the above array until a column is repeated.

4. The low-order byte of many random sequences (e.g., linear congruential sequences with $m =$ word size) is much less random than the high-order byte. See Section 3.2.1.1.

5. The randomizing effect would be quite minimized, because $V[j]$ would always contain a number in a certain range, essentially $j/k \le V[j]/m < (j+1)/k$. However, some similar approaches could be used: we could take $Y_n = X_{n-1}$, or we could choose j from X_n by extracting some digits from about the middle instead of at the extreme left. None of these suggestions would produce a lengthening of the period analogous to the behavior of Algorithm B.

6. For example, if $\mathtt{X}_n/m < \frac{1}{2}$, then $\mathtt{X}_{n+1} = 2\mathtt{X}_n$.

7. [W. Mantel, *Nieuw Arehief voor Wiskunde* (2) **1** (1897), 172–184.]

The subsequence of X values:	00...01 00...10 . . . 10...00 CONTENTS(A)	becomes:	00...01 00...10 . . . 10...00 00...00 CONTENTS(A)

8. We may assume that $X_0 = 0$ and $m = p^e$, as in the proof of Theorem 3.2.1.2A. First suppose that the sequence has period length p^e; it follows that the period of the sequence mod p^f has length p^f, for $1 \le f \le e$, otherwise some residues mod p^f would never occur. Clearly, c is not a multiple of p, for otherwise each X_n would be a multiple of p. If $p \le 3$, it is easy to establish the necessity of conditions (iii) and (iv) by trial and error, so we may assume that $p \ge 5$. If $d \not\equiv 0$ (modulo p) then $dx^2 + ax + c \equiv d(x + a_1)^2 + c_1$ (modulo p^e) for some integers a_1 and c_1 and for all integers x; this quadratic takes the same value at the points x and $-x - 2a_1$, so it cannot assume all values modulo p^e. Hence $d \equiv 0$ (modulo p); and if $a \not\equiv 1$, we would have $dx^2 + ax + c \equiv x$ (modulo p) for some x, contradicting the fact that the sequence mod p has period length p.

To show the sufficiency of the conditions, we may assume by Theorem 3.2.1.2A and consideration of some trivial cases that $m = p^e$ where $e \ge 2$. If $p = 2$, we have $X_{n+p} \equiv X_n + pc$ (modulo p^2), by trial; and if $p = 3$, we have either $X_{n+p} \equiv X_n + pc$ (modulo p^2), for all n, or $X_{n+p} \equiv X_n - pc$ (modulo p^2), for all n. For $p \ge 5$, we can prove that $X_{n+p} \equiv X_n + pc$ (modulo p^2): Let $d = pr$, $a = 1 + ps$. Then if

$X_n \equiv cn + pY_n$ (modulo p^2), we must have $Y_{n+1} \equiv n^2c^2r + ncs + Y_n$ (modulo p); hence $Y_n \equiv \binom{n}{3}2c^2r + \binom{n}{2}(c^2r + cs)$ (modulo p). Thus $Y_p \bmod p = 0$, and the desired relation has been proved.

Now we can prove that the sequence $\langle X_n \rangle$ of integers defined in the "hint" satisfies the relation

$$X_{n+p^f} \equiv X_n + tp^f \quad (\text{modulo } p^{f+1}), \qquad n \geq 0,$$

for some t with $t \bmod p \neq 0$, and for all $f \geq 1$. This suffices to prove that the sequence $\langle X_n \bmod p^e \rangle$ has period length p^e, for the length of the period is a divisor of p^e but not a divisor of p^{e-1}. The above relation has already been established for $f = 1$, and for $f > 1$ it can be proved by induction in the following manner: Let

$$X_{n+p^f} \equiv X_n + tp^f + Z_np^{f+1} \quad (\text{modulo } p^{f+2});$$

then the quadratic law for generating the sequence, with $d = pr$, $a = 1 + ps$, yields $Z_{n+1} \equiv 2rtnc + st + Z_n$ (modulo p). It follows that $Z_{n+p} \equiv Z_n$ (modulo p); hence

$$X_{n+kp^f} \equiv X_n + k(tp^f + Z_np^{f+1}) \quad (\text{modulo } p^{f+2})$$

for $k = 1, 2, 3, \ldots$; setting $k = p$ completes the proof.

Notes: If $f(x)$ is a polynomial of degree higher than 2 and $X_{n+1} = f(X_n)$, the analysis is more complicated, although we can use the fact that $f(m + p^k) = f(m) + p^kf'(m) + p^{2k}f''(m)/2! + \cdots$ to prove that many polynomial recurrences give the maximum period. For example, Coveyou has proved that the period is $m = 2^e$ if $f(0)$ is odd, $f'(j) \equiv 1$, $f''(j) \equiv 0$, and $f(j+1) \equiv f(j)+1$ (modulo 4) for $j = 0, 1, 2, 3$. [*Studies in Applied Math.* **3** (1967), 70–111.]

9. Let $X_n = 4Y_n+2$; then the sequence Y_n satisfies the quadratic recurrence $Y_{n+1} = (4Y_n^2 + 5Y_n + 1) \bmod 2^{e-2}$.

10. *Case 1:* $X_0 = 0$, $X_1 = 1$; hence $X_n \equiv F_n$. We seek the smallest n for which $F_n \equiv 0$ and $F_{n+1} \equiv 1$ (modulo 2^e). Since $F_{2n} = F_n(F_{n-1} + F_{n+1})$, $F_{2n+1} = F_n^2 + F_{n+1}^2$, we find by induction on e that, for $e > 1$, $F_{3\cdot 2^{e-1}} \equiv 0$ and $F_{3\cdot 2^{e-1}+1} \equiv 2^e + 1$ (modulo 2^{e+1}). This implies that the period is a divisor of $3 \cdot 2^{e-1}$ but not a divisor of $3 \cdot 2^{e-2}$, so it is either $3 \cdot 2^{e-1}$ or 2^{e-1}. But $F_{2^{e-1}}$ is always odd (since only F_{3n} is even).

Case 2: $X_0 = a$, $X_1 = b$. Then $X_n \equiv aF_{n-1} + bF_n$; we need to find the smallest positive n with $a(F_{n+1} - F_n) + bF_n \equiv a$ and $aF_n + bF_{n+1} \equiv b$. This implies that $(b^2 - ab - a^2)F_n \equiv 0$, $(b^2 - ab - a^2)(F_{n+1} - 1) \equiv 0$; and $b^2 - ab - a^2$ is odd (i.e., prime to m) so the condition is equivalent to $F_n \equiv 0$, $F_{n+1} \equiv 1$.

Methods to determine the period of F_n for any modulus appear in an article by D. D. Wall, *AMM* **67** (1960), 525–532. Further facts about the Fibonacci sequence mod 2^e have been derived by B. Jansson [*Random Number Generators* (Stockholm: Almqvist & Wiksell, 1966), Section 3C1].

11. (a) We have $z^\lambda = 1 + f(z)u(z) + p^ev(z)$ for some $u(z), v(z)$, where $v(z) \not\equiv 0$ (modulo $f(z)$ and p). By the binomial theorem

$$z^{\lambda p} = 1 + p^{e+1}v(z) + p^{2e+1}v(z)^2(p-1)/2$$

plus further terms congruent to zero (modulo $f(z)$ and p^{e+2}). Since $p^e > 2$, we have $z^{\lambda p} \equiv 1+p^{e+1}v(z)$ (modulo $f(z)$ and p^{e+2}). If $p^{e+1}v(z) \equiv 0$ (modulo $f(z)$ and p^{e+2}),

there must exist polynomials $a(z)$ and $b(z)$ such that $p^{e+1}(v(z)+pa(z)) = f(z)b(z)$. Since $f(0)=1$, this implies that $b(z)$ is a multiple of p^{e+1} (by Gauss's Lemma 4.6.1G); hence $v(z) \equiv 0$ (modulo $f(z)$ and p), a contradiction.

(b) If $z^\lambda - 1 = f(z)u(z) + p^e v(z)$, then

$$G(z) = u(z)/(z^\lambda - 1) + p^e v(z)/f(z)(z^\lambda - 1);$$

hence $A_{n+\lambda} \equiv A_n$ (modulo p^e) for large n. Conversely, if $\langle A_n \rangle$ has the latter property then $G(z) = u(z) + v(z)/(1-z^\lambda) + p^e H(z)$, for some polynomials $u(z)$ and $v(z)$, and some power series $H(z)$, all with integer coefficients. This implies the identity $1-z^\lambda = u(z)f(z)(1-z^\lambda) + v(z)f(z) + p^e H(z)f(z)(1-z^\lambda)$; and $H(z)f(z)(1-z^\lambda)$ is a polynomial since the other terms of the equation are polynomials.

(c) It suffices to prove that $\lambda(p^e) \neq \lambda(p^{e+1})$ implies that $\lambda(p^{e+1}) = p\lambda(p^e) \neq \lambda(p^{e+2})$. Applying (a) and (b), we know that $\lambda(p^{e+2}) \neq p\lambda(p^e)$, and that $\lambda(p^{e+1})$ is a divisor of $p\lambda(p^e)$ but not of $\lambda(p^e)$. Hence if $\lambda(p^e) = p^f q$, where $q \bmod p \neq 0$, then $\lambda(p^{e+1})$ must be $p^{f+1}d$, where d is a divisor of q. But now $X_{n+p^{f+1}} \equiv X_n$ (modulo p^e); hence $p^{f+1}d$ is a multiple of $p^f q$, hence $d = q$. [*Note:* The hypothesis $p^e > 2$ is necessary; for example, let $a_1 = 4$, $a_2 = -1$, $k = 2$; then $\langle A_n \rangle = 1, 4, 15, 56, 209, 780, \ldots$; $\lambda(2) = 2$, $\lambda(4) = 4$, $\lambda(8) = 4$.]

(d) $g(z) = X_0 + (X_1 - a_1X_0)z + \cdots$
$$+(X_{k-1} - a_1X_{k-2} - a_2X_{k-3} - \cdots - a_{k-1}X_0)z^{k-1}.$$

(e) The derivation in (b) can be generalized to the case $G(z) = g(z)/f(z)$; then the assumption of period length λ implies that $g(z)(1-z^\lambda) \equiv 0$ (modulo $f(z)$ and p^e); we treated only the special case $g(z) = 1$ above. But both sides of this congruence can be multiplied by Hensel's $b(z)$, and we obtain $1 - z^\lambda \equiv 0$ (modulo $f(z)$ and p^e).

Note: A more "elementary" proof of the result in (c) can be given without using generating functions, using methods analogous to those in the answer to exercise 8: If $A_{\lambda+n} = A_n + p^e B_n$, for $n = r, r+1, \ldots, r+k-1$ and some integers B_n, then this same relation holds for *all* $n \geq r$ if we define $B_{r+k}, B_{r+k+1}, \ldots$ by the given recurrence relation. Since the resulting sequence of B's is some linear combination of shifts of the sequence of A's, we will have $B_{\lambda+n} \equiv B_n$ (modulo p^e) for all large enough values of n. Now $\lambda(p^{e+1})$ must be some multiple of $\lambda = \lambda(p^e)$; for all large enough n we have $A_{n+j\lambda} = A_n + p^e(B_n + B_{n+\lambda} + B_{n+2\lambda} + \cdots + B_{n+(j-1)\lambda}) \equiv A_n + jp^e B_n$ (modulo p^{2e}) for $j = 1, 2, 3, \ldots$. No k consecutive B's are multiples of p; hence $\lambda(p^{e+1}) = p\lambda(p^e) \neq \lambda(p^{e+2})$ follows immediately when $e \geq 2$. We still must prove that $\lambda(p^{e+2}) \neq p\lambda(p^e)$ when p is odd and $e = 1$; here we let $B_{\lambda+n} = B_n + pC_n$, and observe that $C_{n+\lambda} \equiv C_n$ (modulo p) when n is large enough. Then $A_{n+p} \equiv A_n + p^2\binom{p}{2}C_n$ (modulo p^3), and the proof is readily completed.

For the history of this problem, see Morgan Ward, *Trans. Amer. Math. Soc.* **35** (1933), 600–628; see also D. W. Robinson, *AMM* **73** (1966), 619–621.

12. The period length mod 2 can be at most 4; and the period length mod 2^{e+1} is at most twice the maximum length mod 2^e, by the considerations of the previous exercise. So the maximum conceivable period length is 2^{e+1}; this is achievable, for example, in the trivial case $a = 0$, $b = c = 1$.

13, 14. Clearly $Z_{n+\lambda} = Z_n$, so λ' is certainly a divisor of λ. Let the least common multiple of λ' and λ_1 be λ_1', and define λ_2' similarly. $X_n + Y_n \equiv Z_n \equiv Z_{n+\lambda_1'} \equiv X_n + Y_{n+\lambda_1'}$, so λ_1' is a multiple of λ_2. Similarly, λ_2' is a multiple of λ_1. This yields the desired result. (The result is "best possible" in the sense that sequences for which $\lambda' = \lambda_0$ can be constructed, as well as sequences for which $\lambda' = \lambda$.)

15. Algorithm M generates (X_{n+k}, Y_n) in step M1 and outputs $Z_n = X_{n+k-q_n}$ in step M3, for all sufficiently large n. Thus $\langle Z_n \rangle$ has a period of length λ', where λ' is the least positive integer such that $X_{n+k-q_n} = X_{n+\lambda'+k-q_{n+\lambda'}}$ for all large n. Since λ is a multiple of λ_1 and λ_2, it follows that λ' is a divisor of λ. (These observations are due to Alan G. Waterman.)

We also have $n + k - q_n \equiv n + \lambda' + k - q_{n+\lambda'}$ (modulo λ_1) for all large n, by the distinctness of the X's. The bound on $\langle q_n \rangle$ implies that $q_{n+\lambda'} = q_n + c$ for all large n, where $c \equiv \lambda'$ (modulo λ_1) and $|c| < \frac{1}{2}\lambda_1$. But c must be 0 since $\langle q_n \rangle$ is bounded. Hence $\lambda' \equiv 0$ (modulo λ_1), and $q_{n+\lambda'} = q_n$ for all large n; it follows that λ' is a multiple of λ_2 and λ_1, so $\lambda' = \lambda$.

Note: The answer to exercise 3.2.1.2–4 implies that when $\langle Y_n \rangle$ is a linear congruential sequence of maximum period modulo $m = 2^e$, the period length λ_2 will be at most 2^{e-2} when k is a power of 2.

16. There are several methods of proof.

(1) Using the theory of finite fields. In the field with 2^k elements let ξ satisfy $\xi^k = a_1\xi^{k-1} + \cdots + a_k$. Let $f(b_1\xi^{k-1} + \cdots + b_k) = b_k$, where each b_j is either zero or one; this is a linear function. If word X in the generation algorithm is $(b_1 b_2 \ldots b_k)_2$ before (10) is executed, and if $b_1\xi^{k-1} + \cdots + b_k\xi^0 = \xi^n$, then word X represents ξ^{n+1} after (10) is executed. Hence the sequence is $f(\xi^n)$, $f(\xi^{n+1})$, $f(\xi^{n+2})$, ...; and $f(\xi^{n+k}) = f(\xi^n\xi^k) = f(a_1\xi^{n+k-1} + \cdots + a_k\xi^n) = a_1 f(\xi^{n+k-1}) + \cdots + a_k f(\xi^n)$.

(2) Using brute force, or elementary ingenuity. We are given a sequence X_{nj}, $n \geq 0$, $1 \leq j \leq k$, satisfying

$$X_{(n+1)j} \equiv X_{n(j+1)} + a_j X_{n1}, \qquad 1 \leq j < k; \qquad X_{(n+1)k} \equiv a_k X_{n1} \quad \text{(modulo 2)}.$$

We must show that this implies $X_{nk} \equiv a_1 X_{(n-1)k} + \cdots + a_k X_{(n-k)k}$, for $n \geq k$. Indeed, it implies $X_{nj} \equiv a_1 X_{(n-1)j} + \cdots + a_k X_{(n-k)j}$ when $1 \leq j \leq k \leq n$. This is clear for $j = 1$, since $X_{n1} \equiv a_1 X_{(n-1)1} + X_{(n-1)2} \equiv a_1 X_{(n-1)1} + a_2 X_{(n-2)2} + X_{(n-2)3}$, etc. For $j > 1$, we have by induction

$$\begin{aligned} X_{nj} &\equiv X_{(n+1)(j-1)} - a_{j-1}X_{n1} \\ &\equiv \sum_{1\leq i\leq k} a_i X_{(n+1-i)(j-1)} - a_{j-1} \sum_{1\leq i\leq k} a_i X_{(n-i)1} \\ &\equiv \sum_{1\leq i\leq k} a_i (X_{(n+1-i)(j-1)} - a_{j-1} X_{(n-i)1}) \\ &\equiv a_1 X_{(n-1)j} + \cdots + a_k X_{(n-k)j}. \end{aligned}$$

This proof does not depend on the fact that operations were done modulo 2, or modulo any prime number.

17. (a) When the sequence terminates, the $(k-1)$-tuple $(X_{n+1}, \ldots, X_{n+k-1})$ occurs for the $(m+1)$st time. A given $(k-1)$-tuple $(X_{r+1}, \ldots, X_{r+k-1})$ can have only m distinct predecessors X_r, so one of these occurrences must be for $r = 0$. (b) Since the $(k-1)$-tuple $(0, \ldots, 0)$ occurs $(m+1)$ times, each possible predecessor appears, so the k-tuple $(a_1, 0, \ldots, 0)$ appears for all a_1, $0 \leq a_1 < m$. Let $1 \leq s < k$ and suppose we have proved that all k-tuples $(a_1, \ldots, a_s, 0, \ldots, 0)$ appear in the sequence when $a_s \neq 0$. By the construction, this k-tuple would not be in the sequence unless $(a_1, \ldots, a_s, 0, \ldots, 0, y)$ had appeared earlier for $1 \leq y < m$. Hence the $(k-1)$-tuple

$(a_1, \ldots, a_s, 0, \ldots, 0)$ has appeared m times, and all m possible predecessors appear; this means $(a, a_1, \ldots, a_s, 0, \ldots, 0)$ appears for $0 \le a < m$. The proof is now complete by induction.

The result also follows from Theorem 2.3.4.2D, using the directed graph of exercise 2.3.4.2–23; the set of arcs from $(x_1, \ldots, x_j, 0, \ldots, 0)$ to $(x_2, \ldots, x_j, 0, \ldots, 0, 0)$, where $x_j \neq 0$ and $1 \le j \le k$, forms an oriented subtree related neatly to Dewey decimal notation.

18. The third-most-significant bit of U_{n+1} is completely determined by the first and third bits of U_n, so only 32 of the 64 possible pairs $(\lfloor 8U_n \rfloor, \lfloor 8U_{n+1} \rfloor)$ occur. (*Notes:* If we had used, say, 11-bit numbers $U_n = (.X_{11n}X_{11n+1}\ldots X_{11n+10})_2$, the sequence *would* be satisfactory for many applications. If another constant appears in A having more "one" bits, the generalized spectral test might give some indication of its suitability. See exercise 3.3.4–24; we could examine ν_t in dimensions $t = 36, 37, 38, \ldots$.)

21. [*J. London Math. Soc.* **21** (1946), 169–172.] Any sequence of period length $m^k - 1$ with no k consecutive zeros leads to a sequence of period length m^k by inserting a zero in the appropriate place, as in exercise 7; conversely, we can start with a sequence of period length m^k and delete an appropriate zero from the period, to form a sequence of the other type. Let us call these "(m, k) sequences" of types A and B. The hypothesis assures us of the existence of (p, k) sequences of type A, for all primes p and all $k \ge 1$; hence we have (p, k) sequences of type B for all such p and k.

To get a (p^e, k) sequence of type B, let $e = qr$, where q is a power of p and r is not a multiple of p. Start with a (p, qrk) sequence of type A, namely $X_0, X_1, X_2, \ldots$; then (using the p-ary number system) the grouped digits $(X_0 \ldots X_{q-1})_p, (X_q \ldots X_{2q-1})_p, \ldots$ form a (p^q, rk) sequence of type A, since q is relatively prime to $p^{qrk} - 1$ and the sequence therefore has a period length of $p^{qrk} - 1$. This leads to a (p^q, rk) sequence $\langle Y_n \rangle$ of type B; and $(Y_0Y_1 \ldots Y_{r-1})_{p^q}, (Y_rY_{r+1} \ldots Y_{2r-1})_{p^q}, \ldots$ is a (p^{qr}, k) sequence of type B by a similar argument, since r is relatively prime to p^{qk}.

To get an (m, k) sequence of type B for arbitrary m, we can combine (p^e, k) sequences for each of the prime power factors of m using the Chinese remainder theorem; but a simpler method is available. Let $\langle X_n \rangle$ be an (r, k) sequence of type B, and let $\langle Y_n \rangle$ be an (s, k) sequence of type B, where r and s are relatively prime; then $\langle sX_n + Y_n \rangle$ is an (rs, k) sequence of type B.

A simple, uniform construction that yields $(2, k)$ sequences for arbitrary k has been discovered by A. Lempel [*IEEE Trans.* **C–19** (1970), 1204–1209].

22. By the Chinese remainder theorem, we can find constants $a_1, \ldots, a_k$ having desired residues mod each prime divisor of m. If $m = p_1p_2 \ldots p_t$, the period length will be $\operatorname{lcm}(p_1^k - 1, \ldots, p_t^k - 1)$. In fact, we can achieve reasonably long periods for arbitrary m (not necessarily squarefree), as shown in exercise 11.

23. Period length at least $2^{55} - 1$; possibly faster than (7), see exercise 3.2.1.1–5. Furthermore, R. Brent has pointed out that the calculations can be done exactly on floating point numbers in $[0, 1)$.

24. Run the sequence backwards. In other words, if $Z_n = Y_{-n}$ we have $Z_n = (Z_{n-m+k} + Z_{n-m}) \bmod 2$.

25. This actually would be slower and more complicated, unless it can be used to save subroutine-calling overhead in high-level languages. (See the FORTRAN program in Section 3.6.)

27. Let $J_n = \lfloor kX_n/m \rfloor$. *Lemma:* After the $(k^2 + 7k - 2)/2$ consecutive values

$$0^{k+2}\,1\,0^{k+1}\,2\,0^{k}\ldots(k-1)\,0^{3}$$

occur in the $\langle J_n \rangle$ sequence, Algorithm B will have $V[i] < m/k$ for $0 \le j < k$, and also $Y < m/k$. *Proof.* Let S_n be the set of positions i such that $V[i] < m/k$ just before X_n is generated, and let j_n be the index such that $V[j_n] \leftarrow X_n$. If $j_n \notin S_n$ and $J_n = 0$, then $S_{n+1} = S_n \cup \{j_n\}$; if $j_n \in S_n$ and $J_n = 0$, then $S_{n+1} = S_n$ and $j_{n+1} = 0$. After $k+2$ successive 0's, we must therefore have $0 \in S_n$ and $j_{n+1} = 0$. Then after "$1\,0^{k+1}$" we must have $\{0,1\} \subseteq S_n$ and $j_{n+1} = 0$; after "$2\,0^{k}$" we must have $\{0,1,2\} \subseteq S_n$ and $j_{n+1} = 0$; and so on.

Corollary: For $\lambda \ge 2(k^2+7k-2)k^{(k^2+7k-2)/2}$, either Algorithm B yields a period of length λ or the sequence $\langle X_n \rangle$ is poorly distributed. *Proof.* The probability that any given length-l pattern of J's does not occur in a random sequence of length λ is less than $(1 - k^{-l})^{\lambda/l} < \exp(-k^{-l}\lambda/l)$. For $l = (k^2 + 7k - 2)/2$ this is at most e^{-4}; hence the stated pattern should appear. After it does, the subsequent behavior of Algorithm B will be the same each time it reaches this part of the period.

SECTION 3.3.1

1. There are $k = 11$ categories, so the line $\nu = 10$ should be used.

2. $\frac{2}{49}, \frac{3}{49}, \frac{4}{49}, \frac{5}{49}, \frac{6}{49}, \frac{9}{49}, \frac{6}{49}, \frac{5}{49}, \frac{4}{49}, \frac{3}{49}, \frac{2}{49}$.

3. $V = 7\frac{173}{240}$, only very slightly higher than that obtained from the good dice! There are two reasons why we do not detect the weighting: (a) The new probabilities (cf. exercise 2) are not really very far from the old ones in Eq. (1). The sum of the two dice tends to smooth out the probabilities; if we considered instead each of the 36 possible pairs of values, and counted these, we would probably detect the difference quite rapidly (assuming that the two dice are distinguishable). (b) A far more important reason is that n is too small for a significant difference to be detected. If the same experiment is done for large enough n, the faulty dice will be discovered (see exercise 12).

4. $p_s = \frac{1}{12}$ for $2 \le s \le 12$ and $s \ne 7$; $p_7 = \frac{1}{6}$. The value of V is $16\frac{1}{2}$, which falls between the 75% and 95% entries in Table 1; so it is reasonable, in spite of the fact that not too many sevens actually turned up.

5. $K_{20}^{+} = 1.15$; $K_{20}^{-} = 0.215$; these do not differ significantly from random behavior (being at about the 94% and 86% levels), but they are mighty close. (The data values in this exercise come from Appendix A, Table 1.)

6. The probability that $X_j \le x$ is $F(x)$, so we have the binomial distribution discussed in Section 1.2.10. $F_n(x) = s/n$ with probability $\binom{n}{s} F(x)^s(1 - F(x))^{n-s}$; the mean is $F(x)$; the standard deviation is $\sqrt{F(x)(1 - F(x))/n}$. [Cf. Eq. 1.2.10–19. This suggests that a slightly better statistic would be to define

$$K_n^{+} = \sqrt{n} \max_{-\infty < x < \infty} (F_n(x) - F(x))/\sqrt{F(x)(1 - F(x))};$$

see exercise 22. We can calculate the mean and standard deviation of $F_n(x) - F_n(y)$, for $x < y$, and obtain the covariance of $F_n(x), F_n(y)$. Using these facts, it can be

shown that for large values of n the function $F_n(x)$ behaves as a "Brownian motion," and techniques from this branch of probability theory may be used to study it. The situation is exploited in articles by J. L. Doob and M. D. Donsker, *Annals Math. Stat.* **20** (1949), 393–403 and **23** (1952), 277–281; this is generally regarded as the most enlightening way to study the KS tests.]

7. ((Cf. Eq. (13).) Take $j = n$ to see that K_n^+ is never negative and it can get as high as $\sqrt{n}$. Similarly, take $j = 1$ to make the same observations about K_n^-.

8. The new KS statistic was computed for 20 observations. The distribution of K_{10}^+ was used as $F(x)$ when the KS statistic was computed.

9. The idea is erroneous, because all of the observations must be *independent*. There is a relation between the statistics K_n^+ and K_n^- on the same data, so each test should be judged separately. (A high value of one tends to give a low value of the other.) Similarly, the entries in Figs. 2 and 5 (which show 15 tests for each generator) do not show 15 independent observations, because the maximum-of-5 test is not independent of the maximum-of-4 test. The three tests of each horizontal row are independent (because they were done on different parts of the sequence), but the five tests in a column are somewhat correlated. The net effect of this is that the 95-percent probability levels, etc., which apply to one test, cannot legitimately be applied to a whole group of tests on the same data. Moral: When testing a random number generator, we may expect it to "pass" each of several tests, e.g., the frequency test, maximum test, run test; but an array of data from several different tests should not be considered as a unit since the tests themselves may not be independent. The K_n^+ and K_n^- statistics should be considered as two separate tests; a good source of random numbers will pass both of the tests.

10. Each Y_s is doubled, and np_s is doubled, so the numerators of (6) are quadrupled while the denominators only double. Hence the new value of V is twice as high as the old one.

11. The empirical distribution function stays the same; the values of K_n^+ and K_n^- are multiplied by $\sqrt{2}$.

12. Let $Z_s = (Y_s - nq_s)/\sqrt{nq_s}$. The value of V is n times

$$\sum_{1 \le s \le k} (q_s - p_s + \sqrt{q_s/n}Z_s)^2/p_s,$$

and the latter quantity stays bounded away from zero as n increases (since $Z_s n^{-1/4}$ is bounded with probability 1). Hence the value of V will increase to a value that is extremely improbable under the p_s assumption.

For the KS test, let $F(x)$ be the assumed distribution, $G(x)$ the actual distribution, and let $h = \max|G(x) - F(x)|$. Take n large enough so that $|F_n(x) - G(x)| > h/2$ occurs with very small probability; then $|F_n(x) - F(x)|$ will be improbably high under the assumed distribution $F(x)$.

13. (The "max" notation should really be replaced by "sup" since a least upper bound is meant; however, "max" was used in the text to avoid confusing too many readers by the less familiar "sup" notation.) For convenience, let $X_0 = -\infty$, $X_{n+1} = +\infty$. When $X_j \le x < X_{j+1}$, we have $F_n(x) = j/n$; therefore $\max(F_n(x) - F(x)) = j/n - F(X_j)$ and $\max(F(x) - F_n(x)) = F(X_{j+1}) - j/n$ in this interval. As j varies

from 0 to n, all real values of x are considered; this proves that

$$K_n^+ = \sqrt{n} \max_{0\le j\le n}\left(\frac{j}{n} - F(X_j)\right);$$
$$K_n^- = \sqrt{n} \max_{1\le j\le n+1}\left(F(X_j) - \frac{j-1}{n}\right).$$

These are equivalent to (13), since the extra term under the maximum signs is nonpositive and it must be redundant by exercise 7.

14. The logarithm of the left-hand side simplifies to

$$-\sum_{1\le s\le k} Y_s \ln\left(1+\frac{Z_s}{\sqrt{np_s}}\right) + \frac{1-k}{2}\ln(2\pi n) - \frac{1}{2}\sum_{1\le s\le k}\ln p_s - \frac{1}{2}\sum_{1\le s\le k}\ln\left(1+\frac{Z_s}{\sqrt{np_s}}\right) + O\left(\frac{1}{n}\right),$$

and this quantity simplifies further (upon expanding $\ln(1 + Z_s/\sqrt{np_s})$ and realizing that $\sum_{1\le s\le k} Z_s\sqrt{np_s} = 0$) to

$$-\frac{1}{2}\sum_{1\le s\le k} Z_s^2 + \frac{1-k}{2}\ln(2\pi n) - \frac{1}{2}\ln(p_1\ldots p_k) + O\left(\frac{1}{\sqrt{n}}\right).$$

15. The corresponding Jacobian determinant is easily evaluated by (a) removing the factor r^{n-1} from the determinant, (b) expanding the resulting determinant by the cofactors of the row containing "$\cos\theta_1 \ -\sin\theta_1\ 0\ldots 0$" (each of the cofactor determinants may be evaluated by induction), and (c) recalling that $\sin^2\theta_1 + \cos^2\theta_1 = 1$.

16. $$\int_0^{z\sqrt{2x}+y} \exp\left(-\frac{u^2}{2x}+\cdots\right)du = ye^{-z^2} + O\left(\frac{1}{\sqrt{x}}\right) + \int_0^{z\sqrt{2x}} \exp\left(-\frac{u^2}{2x}+\cdots\right)du.$$

The latter integral is

$$\int_0^{z\sqrt{2x}} e^{-u^2/2x}\,du + \frac{1}{3x^2}\int_0^{z\sqrt{2x}} e^{-u^2/2x}u^3\,du + O\left(\frac{1}{\sqrt{x}}\right).$$

When all is put together, the final result is

$$\frac{\gamma(x+1, x+z\sqrt{2x}+y)}{\Gamma(x+1)} = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z\sqrt{2}} e^{-u^2/2}\,du + \frac{e^{-z^2}}{\sqrt{2\pi x}}(y - \tfrac{2}{3} - \tfrac{2}{3}z^2) + O\left(\frac{1}{x}\right).$$

If we set $z\sqrt{2} = x_p$ and write

$$\frac{1}{\sqrt{2\pi}}\int_{-\infty}^{z\sqrt{2}} e^{-u^2/2}\,du = p, \qquad x+1 = \frac{\nu}{2}, \qquad \gamma\left(\frac{\nu}{2}, \frac{t}{2}\right)\Big/\Gamma\left(\frac{\nu}{2}\right) = p,$$

where $t/2 = x + z\sqrt{2x} + y$, we can solve for y to obtain $y = \frac{2}{3}(1+z^2) + O(1/\sqrt{x})$, which is consistent with the above analysis. The solution is therefore $t = \nu + 2\sqrt{\nu}z + \frac{4}{3}z^2 - \frac{2}{3} + O(1/\sqrt{\nu})$.

17. (a) Change of variable, $x_j \leftarrow x_j + t$. (b) Induction on n; by definition,

$$P_{n0}(x-t) = \int_n^x P_{(n-1)0}(x_n - t)\,dx_n.$$

(c) The left-hand side is

$$\int_n^{x+t} dx_n \ldots \int_{k+1}^{x_{k+2}} dx_{k+1} \qquad \text{times} \qquad \int_t^k dx_k \int_t^{x_k} dx_{k-1} \ldots \int_t^{x_2} dx_1.$$

(d) From (b), (c) we have

$$P_{nk}(x) = \sum_{0 \le r \le k} \frac{(r-t)^r}{r!} \frac{(x+t-r)^{n-r-1}}{(n-r)!}(x+t-n).$$

The numerator in (24) is $P_{n\lfloor t \rfloor}(n)$.

18. We may assume that $F(x) = x$ for $0 \le x \le 1$, as remarked in the text's derivation of (24). If $0 \le X_1 \le \cdots \le X_n \le 1$, let $Z_j = 1 - X_{n+1-j}$. We have $0 \le Z_1 \le \cdots \le Z_n \le 1$; and K_n^+ evaluated for $X_1, \ldots, X_n$ equals K_n^- evaluated for $Z_1, \ldots, Z_n$. This symmetrical relation gives a one-to-one correspondence between sets of equal volume for which K_n^+ and K_n^- fall in a given range.

23. Let m be any number $\ge n$. (a) If $\lfloor mF(X_i) \rfloor = \lfloor mF(X_j) \rfloor$ and $i > j$, then $i/n - F(X_i) > j/n - F(X_j)$. (b) Start with $a_k = 1.0$, $b_k = 0.0$, and $c_k = 0$ for $0 \le k < m$. Then do the following for each observation X_j: Set $Y \leftarrow F(X_j)$, $k \leftarrow \lfloor mY \rfloor$, $a_k \leftarrow \min(a_k, Y)$, $b_k \leftarrow \max(b_k, Y)$, $c_k \leftarrow c_k + 1$. (Assume that $F(X_j) < 1$ so that $k < m$.) Then set $j \leftarrow 0$, $r^+ \leftarrow r^- \leftarrow 0$, and for $k = 0, 1, \ldots, m-1$ (in this order) do the following whenever $c_k > 0$: Set $r^- \leftarrow \max(r^-, a_k - j/n)$, $j \leftarrow j + c_k$, $r^+ \leftarrow \max(r^+, j/n - b_k)$. Finally set $K_n^+ \leftarrow \sqrt{n}\,r^+$, $K_n^- \leftarrow \sqrt{n}\,r^-$. The time required is $O(m+n)$, and the precise value of n need not be known in advance. (If the estimate $(k + \frac{1}{2})/m$ is used for a_k and b_k, so that only the values c_k are actually computed for each k, we obtain estimates of K_n^+ and K_n^- good to within $\frac{1}{2}\sqrt{n}/m$, even when $m < n$.) [*ACM Trans. Math. Software* **3** (1977), 60–64.]

SECTION 3.3.2

1. The observations for a chi-square test must be independent, and in the second sequence successive observations are manifestly dependent, since the second component of one equals the first component of the next.

2. Form t-tuples $(Y_{jt}, \ldots, Y_{jt+t-1})$, for $0 \le j < n$, and count how many of these equal each possible value. Apply the chi-square test with $k = d^t$ and with probability $1/d^t$ in each category. The number of observations, n, should be at least $5d^t$.

3. The probability that j values are examined, i.e., the probability that U_{j-1} is the nth element of the sequence lying in the range $\alpha \le U_{j-1} < \beta$, is easily seen to be

$$\binom{j-1}{n-1} p^n (1-p)^{j-n},$$

by enumeration of the possible places in which the other $n-1$ occurrences can appear and by evaluating the probability of such a pattern. The generating function is $G(z) = (pz/(1-(1-p)z))^n$, which makes sense since the given distribution is the n-fold convolution of the same thing for $n = 1$. Hence the mean and variance are proportional to n; the number of U's to be examined is now easily found to have the characteristics (min n, ave n/p, max ∞, dev $\sqrt{n(1-p)}/p$). A more detailed discussion of this probability distribution when $n = 1$ may be found in the answer to exercise 3.4.1–17; see also the considerably more general results of exercise 2.3.4.2–26.

4. The probability of a gap of length $\ge r$ is the probability that r consecutive U's lie outside the given range, i.e., $(1-p)^r$. The probability of a gap of length exactly r is the above value for length $\ge r$ minus the value for length $\ge (r+1)$.

5. As N goes to infinity, so does n (with probability 1), hence this test is just the same as the gap test described in the text except for the length of the very last gap. And the text's gap test certainly is asymptotic to the chi-square distribution stated, since the length of each gap is independent of the length of the others. [*Notes:* A quite complicated proof of this result by E. Bofinger and V. J. Bofinger appears in *Annals Math. Stat.* **32** (1961), 524–534. Their paper is noteworthy because it discusses several interesting variations of the gap test; they show, for example, that the quantity

$$\sum_{0 \le r \le t} \frac{(Y_r - (Np)p_r)^2}{(Np)p_r}$$

does *not* approach a chi-square distribution, although others had suggested this statistic as a "stronger" test because Np is the expected value of n.]

7. 5, 3, 5, 6, 5, 5, 4.

8. See exercise 10, with $w = d$.

9. (Change d to w in steps C1 and C4.) We have

$$p_r = \frac{d(d-1)\ldots(d-w+1)}{d^r} \left\{ {r-1 \atop w-1} \right\}, \qquad \text{for } w \le r < t;$$

$$p_t = 1 - \frac{d!}{d^{t-1}} \left(\frac{1}{0!} \left\{ {t-1 \atop d} \right\} + \cdots + \frac{1}{(d-w)!} \left\{ {t-1 \atop w} \right\} \right).$$

10. As in exercise 3, we really need consider only the case $n = 1$. The generating function for the probability that a coupon set has length r is

$$G(z) = \frac{d!}{(d-w)!} \sum_{r>0} \left\{ {r-1 \atop w-1} \right\} \left(\frac{z}{d}\right)^r = z^w \left(\frac{d-1}{d-z}\right) \cdots \left(\frac{d-w+1}{d-(w-1)z}\right)$$

by the previous exercise and Eq. 1.2.9–28. The mean and variance are readily computed using Theorem 1.2.10A and exercise 3.4.1–17. We find that

$$\text{mean}(G) = w + \left(\frac{d}{d-1} - 1\right) + \cdots + \left(\frac{d}{d-w+1} - 1\right) = d(H_d - H_{d-w}) = \mu;$$
$$\text{var}(G) = d^2(H_d^{(2)} - H_{d-w}^{(2)}) - d(H_d - H_{d-w}) = \sigma^2.$$

The number of U's examined, as the search for a coupon set is repeated n times, therefore has the characteristics (min wn, ave μn, max ∞, dev $\sigma\sqrt{n}$).

11. $\big|\,1\,\big|\,2\,\big|\,9\;\;8\;\;5\;\;3\,\big|\,6\,\big|\,7\;\;0\,\big|\,4\,\big|$.

12. Algorithm R (*Data for run test*).

R1. [Initialize.] Set $j \leftarrow -1$, and set COUNT[1] $\leftarrow$ COUNT[2] $\leftarrow \cdots \leftarrow$ COUNT[6] $\leftarrow 0$. Also set $U_n \leftarrow U_{n-1}$, for convenience in terminating the algorithm.

R2. [Set r zero.] Set $r \leftarrow 0$.

R3. [Is $U_j < U_{j+1}$?] Increase r and j by 1. If $U_j < U_{j+1}$, repeat this step.

R4. [Record the length.] If $r \geq 6$, increase COUNT[6] by one, otherwise increase COUNT[r] by one.

R5. [Done?] If $j < n-1$, return to step R2. ∎

13. There are $(p+q+1)\binom{p+q}{p}$ ways to have $U_{i-1} \gtrless U_i < \cdots < U_{i+p-1} \gtrless U_{i+p} < \cdots < U_{i+p+q-1}$; subtract $\binom{p+q+1}{p+1}$ of these in which $U_{i-1} < U_i$, and subtract $\binom{p+q+1}{1}$ for those in which $U_{i+p-1} < U_{i+p}$; then add in 1 for the case that both $U_{i-1} < U_i$ and $U_{i+p-1} < U_{i+p}$, since this case has been subtracted out twice. (This is a special case of the "inclusion-exclusion" principle, which is explained further in Section 1.3.3.)

14. A run of length r occurs with probability $1/r! - 1/(r+1)!$, assuming distinct U's.

15. This is always true of $F(X)$ when F is continuous and S has distribution F; see Section 3.3.1C.

16. (a) $Z_{jt} = \max(Z_{j(t-1)}, Z_{(j+1)(t-1)})$. If the $Z_{j(t-1)}$ are stored in memory, it is therefore a simple matter to transform this array into the set of Z_{jt} with no auxiliary storage required. (b) With his "improvement," each of the V's should indeed have the stated distribution, but the observations are no longer independent. In fact, when U_j is a relatively large value, all of Z_{jt}, $Z_{(j-1)t}$, ..., $Z_{(j-t+1)t}$ will be equal to U_j; so we almost have the effect of repeating the same data t times (and that would multiply V by t, cf. exercise 3.3.1–10).

17. (b) By Lagrange's identity, the difference is $\sum_{0\leq k<j<n}(U'_kV'_j - U'_jV'_k)^2$, and this is certainly positive. (c) Therefore if $D^2 = N^2$, we must have $U'_kV'_j - U'_jV'_k = 0$, for all pairs j, k. This means that the matrix

$$\begin{pmatrix} U'_1 & U'_2 \ldots U'_{n-1} \\ V'_1 & V'_2 \ldots V'_{n-1} \end{pmatrix}$$

has rank < 2, so its rows are linearly dependent. (A more elementary proof can be given, using the fact that $U'_0V'_j - U'_jV'_0 = 0$ for $1 \leq j < n$ implies the existence of constants α, β such that $\alpha U'_j + \beta V'_j = 0$ for all j, provided that U'_0 and V'_0 are not both zero; the latter case can be avoided by a suitable renumbering.)

18. (a) The numerator is $-(U_0 - U_1)^2$, the denominator is $(U_0 - U_1)^2$. (b) The numerator in this case is $-(U_0^2 + U_1^2 + U_2^2 - U_0U_1 - U_1U_2 - U_2U_0)$; the denominator is $2(U_0^2 + \cdots - U_2U_0)$. (c) The denominator always equals $\sum_{0\le j<k<n}(U_j - U_k)^2$, by exercise 1.2.3–30 or 1.2.3–31.

21. The successive values of $c_{r-1} = s - 1$ in step P2 are 2, 3, 7, 6, 4, 2, 2, 1, 0; hence $f = 886862$.

22. $1024 = 6! + 2\cdot 5! + 2\cdot 4! + 2\cdot 3! + 2\cdot 2! + 0\cdot 1!$, so we want the successive values of $s - 1$ in step P2 to be 0, 0, 0, 1, 2, 2, 2, 2, 0; working backwards, the permutation is $(9, 6, 5, 2, 3, 4, 0, 1, 7, 8)$.

SECTION 3.3.3

1. $y((x/y)) + \frac{1}{2}y - \frac{1}{2}y\delta(x/y)$.

2. See exercises 1.2.4–38 and 1.2.4–39(a), (b), (g).

3. $f(x) = \sum_{n\ge 1}(-\sin 2\pi nx)/n$, which converges for all x. (The representation in Eq. (24) may be considered a "finite" Fourier series, for the case when x is rational.)

4. $d = 2^{10}\cdot 5$. Note that we have $X_{n+1} < X_n$ with probability $\frac{1}{2} + \epsilon$, where

$$|\epsilon| < d/(2\cdot 10^{10}) = 1/(2\cdot 5^9);$$

hence *every* potency-10 generator is respectable from the standpoint of Theorem P.

5. An intermediate result:

$$\sum_{0\le x<m} \frac{x}{m}\frac{s(x)}{m} = \frac{1}{12}\sigma(a, m, c) + \frac{m}{4} - \frac{c}{2m} - \frac{x'}{2m}.$$

6. (a) Use induction and the formula

$$\left(\!\!\left(\frac{hj+c}{k}\right)\!\!\right) - \left(\!\!\left(\frac{hj+c-1}{k}\right)\!\!\right) = \frac{1}{k} - \frac{1}{2}\delta\left(\frac{hj+c}{k}\right) - \frac{1}{2}\delta\left(\frac{hj+c-1}{k}\right).$$

(b) Use the fact that $-\left(\!\!\left(\frac{h'j}{k}\right)\!\!\right) = -\left(\!\!\left(\frac{j}{hk} - \frac{k'j}{h}\right)\!\!\right) = \left(\!\!\left(\frac{k'j}{h}\right)\!\!\right) - \frac{j}{hk} + \frac{1}{2}\delta\left(\frac{k'j}{h}\right)$.

7. Take $m = h$, $n = k$, $k = 2$ in the second formula of exercise 1.2.4–45:

$$\sum_{0<j<k}\left(\frac{hj}{k} - \left(\!\!\left(\frac{hj}{k}\right)\!\!\right) + \frac{1}{2}\right)\left(\frac{hj}{k} - \left(\!\!\left(\frac{hj}{k}\right)\!\!\right) - \frac{1}{2}\right) + 2\sum_{0<j<h}\left(\frac{kj}{h} - \left(\!\!\left(\frac{kj}{h}\right)\!\!\right) + \frac{1}{2}\right)j = kh(h-1).$$

The sums on the left simplify, and by standard manipulations we get

$$h^2k - hk - \frac{h}{2} + \frac{h^2}{6k} + \frac{k}{12} + \frac{1}{4} - \frac{h}{6}\sigma(h, k, 0) - \frac{h}{6}\sigma(k, h, 0) + \frac{1}{12}\sigma(1, k, 0) = h^2k - hk.$$

Since $\sigma(1, k, 0) = (k-1)(k-2)/k$, this reduces to the reciprocity law.

8. See *Duke Math. J.* **21** (1954), 391–397.

9. Begin with the handy identity $\sum_{0\le k<r}\lfloor kp/r\rfloor\lfloor kq/r\rfloor+\sum_{0\le k<p}\lfloor kq/p\rfloor\lfloor kr/p\rfloor+\sum_{0\le k<q}\lfloor kr/q\rfloor\lfloor kp/q\rfloor=(p-1)(q-1)(r-1)$ for which a simple geometric proof is possible. [U. Dieter, *Abh. Math. Sem. Univ. Hamburg* **21** (1957), 109–125.]

10. Obviously $\sigma(k-h,k,c)=-\sigma(h,k,-c)$, cf. (8). Replace j by $k-j$ in definition (16), to deduce that $\sigma(h,k,c)=\sigma(h,k,-c)$.

11. (a) $\displaystyle\sum_{0\le j<dk}\left(\!\left(\frac{j}{dk}\right)\!\right)\left(\!\left(\frac{hj+c}{k}\right)\!\right)=\sum_{\substack{0\le i<d\\0\le j<k}}\left(\!\left(\frac{ik+j}{dk}\right)\!\right)\left(\!\left(\frac{hj+c}{k}\right)\!\right)$; use (10) to sum on i. (b) $\displaystyle\left(\!\left(\frac{hj+c+\theta}{k}\right)\!\right)=\left(\!\left(\frac{hj+c}{k}\right)\!\right)+\frac{\theta}{k}-\frac{1}{2}\delta\left(\frac{hj+c}{k}\right)$; now sum.

12. Since $((\frac{hj+c}{k}))$ runs through the same values as $((\frac{j}{k}))$ in some order, Cauchy's inequality implies that $\sigma(h,k,c)^2\le\sigma(h,k,0)^2$; and $\sigma(1,k,0)$ may be summed directly, cf. exercise 7.

13. $\displaystyle\sigma(h,k,c)+\frac{3(k-1)}{k}=\frac{12}{k}\sum_{0<j<k}\frac{\omega^{-cj}}{(\omega^{-hj}-1)(\omega^{j}-1)}+\frac{6}{k}(c \bmod k)-6\left(\!\left(\frac{h'c}{k}\right)\!\right)$,

if $hh'\equiv 1$ (modulo k).

14. $(2^{38}-3\cdot 2^{20}+5)/(2^{70}-1)\approx 2^{-32}$. An extremely satisfactory global value, in spite of the local nonrandomness!

15. Replace c^2 where it appears in (19) by $\lfloor c\rfloor\lceil c\rceil$.

16. The hinted identity is equivalent to $m_1=p_rm_{r+1}+p_{r-1}m_{r+2}$ for $1\le r\le t$; this follows by induction, cf. also exercise 4.5.3–32. Now replace c_j by $\sum_{j\le r\le t}b_rm_{r+1}$ and compare coefficients of b_ib_j on both sides of the identity to be proved.

Note: For all exponents $e\ge 1$ we have

$$\sum_{1\le j\le t}(-1)^{j+1}\frac{c_j^e}{m_jm_{j+1}}=\frac{1}{m_1}\sum_{1\le j\le t}(-1)^{j+1}b_j\frac{(c_j^e-c_{j+1}^e)}{c_j-c_{j+1}}p_{j-1}$$

by a similar argument.

17. During this algorithm we will have $k=m_j$, $h=m_{j+1}$, $c=c_j$, $p=p_j-1$, $p'=p_{j-2}$, $s=(-1)^{j+1}$ for $j=1,2,\ldots,t+1$.

D1. [Initialize.] Set $A\leftarrow 0$, $B\leftarrow h$, $p\leftarrow 1$, $p'\leftarrow 0$, $s\leftarrow 1$.

D2. [Divide.] Set $a\leftarrow\lfloor k/h\rfloor$, $b\leftarrow\lfloor c/h\rfloor$, $r\leftarrow c \bmod h$. (Now $a=a_j$, $b=b_j$, and $r=c_{j+1}$.)

D3. [Accumulate.] Set $A\leftarrow A-(a-6b)s$, $B\leftarrow B+6bp(c+r)s$. If $r\ne 0$ or $c=0$, set $A\leftarrow A-3s$. If $h=1$, set $B\leftarrow B+ps$. (This subtracts $3e(m_{j+1},c_j)$ and also takes care of the $\sum(-1)^{j+1}/m_jm_{j+1}$ terms.)

D4. [Prepare for next iteration.] Set $c\leftarrow r$, $s\leftarrow -s$; set $r\leftarrow k-ah$, $k\leftarrow h$, $h\leftarrow r$; set $r\leftarrow ap+p'$, $p'\leftarrow p$, $p\leftarrow r$. If $h>0$, return to D2. ∎

At the conclusion of this algorithm, p will be equal to the original value k_0 of k, so the desired answer will be $A+B/p$. The final value of p' will be h' if $s < 0$, otherwise p' will be $k_0 - h'$. It would be possible to maintain B in the range $0 \le B < k_0$, by making appropriate adjustments to A, thereby requiring only single-precision operations (with double-precision products and dividends) if k_0 is a single-precision number.

18. A moment's thought shows that the formula

$$S(h,k,c,z) = \sum_{0\le j<k}(\lfloor j/k\rfloor - \lfloor (j-z)/k\rfloor)(((hj+c)/k))$$

is in fact valid for all z, not only when $k \ge z$. Writing $\lfloor j/k\rfloor - \lfloor (j-z)/k\rfloor = \frac{z}{k} + ((\frac{j-z}{k})) - ((\frac{j}{k})) + \frac{1}{2}\delta_{j0} - \frac{1}{2}\delta(\frac{j-z}{k})$ and carrying out the sums yields $S(h,k,c,z) = zd((c/d))/k + \frac{1}{12}\sigma(h,k,hz+c) - \frac{1}{12}\sigma(h,k,c) + \frac{1}{2}((c/k)) - \frac{1}{2}(((hz+c)/k))$, where $d = \gcd(h,k)$. [This formula allows us to express the probability that $X_{n+1} < X_n < \alpha$ in terms of generalized Dedekind sums, given α.]

19. The desired probability is

$$\begin{aligned}
&\sum_{0\le x<m}(\lfloor (x-\alpha)/m\rfloor - \lfloor (x-\beta)/m\rfloor)(\lfloor (s(x)-\alpha')/m\rfloor - \lfloor (s(x)-\beta')/m\rfloor)/m\\
&\quad= \sum_{0\le x<m}\Big(\tfrac{\beta-\alpha}{m} + ((\tfrac{x-\beta}{m})) - ((\tfrac{x-\alpha}{m})) + \tfrac{1}{2}\delta(\tfrac{x-\alpha}{m}) - \tfrac{1}{2}\delta(\tfrac{x-\beta}{m})\Big)\\
&\qquad\times\big(\tfrac{\beta'-\alpha'}{m} + ((\tfrac{ax+c-\beta'}{m})) - ((\tfrac{ax+c-\alpha'}{m})) + \tfrac{1}{2}\delta(\tfrac{ax+c-\alpha'}{m}) - \tfrac{1}{2}\delta(\tfrac{ax+c-\beta'}{m})\big)/m\\
&\quad= \tfrac{\beta-\alpha}{m}\tfrac{\beta'-\alpha'}{m} + \tfrac{1}{12m}(\sigma(a,m,c+a\alpha-\alpha') - \sigma(a,m,c+a\alpha-\beta')\\
&\qquad\qquad + \sigma(a,m,c+a\beta-\beta') - \sigma(a,m,c+a\beta-\alpha')) + \epsilon,
\end{aligned}$$

where $|\epsilon| \le 2.5/m$.

[This approach is due to U. Dieter. The discrepancy between the true probability and the ideal value $\frac{\beta-\alpha}{m}\frac{\beta'-\alpha'}{m}$ is bounded by $\sum_{1\le j\le t} a_j/4m$, according to Theorem K; conversely, by choosing α, β, α', β' appropriately we will obtain a discrepancy of at least half this bound when there are large partial quotients, by the fact that Theorem K is "best possible." Note that when $a \approx \sqrt{m}$ the discrepancy cannot exceed $O(1/\sqrt{m}\,)$, so even the locally nonrandom generator of exercise 14 will look good on the serial test over the full period; it appears that we should insist on an *extremely* small discrepancy.]

20. $\sum_{0\le x<m}\lceil (x-s(x))/m\rceil\lceil (s(x)-s(s(x)))/m\rceil/m = \sum_{0\le x<m}((x-s(x))/m + (((bx+c)/m)) + \frac{1}{2})((s(x)-s(s(x)))/m + ((a(bx+c)/m)) + \frac{1}{2})/m$; and $x/m = ((x/m)) + \frac{1}{2} - \frac{1}{2}\delta(x/m)$, $s(x)/m = (((ax+c)/m)) + \frac{1}{2} - \frac{1}{2}\delta((ax+c)/m)$, $s(s(x))/m = (((a^2x + ac + c)/m)) + \frac{1}{2} - \frac{1}{2}\delta((a^2x+ac+c)/m)$. Let $s(x') = s(x'') = 0$ and $d = \gcd(b,m)$. The sum now reduces to

$$\begin{aligned}
&\frac{1}{4} + \frac{1}{12m}(S_1 - S_2 + S_3 - S_4 + S_5 - S_6 + S_7 - S_8 + S_9) + \frac{d}{2m}\Big(\big((\tfrac{c}{d})\big) + \big((\tfrac{ac}{d})\big)\Big)\\
&\quad+ \frac{1}{2m}\Big(\big((\tfrac{x'-x''}{m})\big) - \big((\tfrac{x'}{m})\big) + \big((\tfrac{x''}{m})\big) + \big((\tfrac{ac+c}{m})\big) - \big((\tfrac{ac}{m})\big) - \big((\tfrac{c}{m})\big) - \frac{1}{2}\Big),
\end{aligned}$$

where $S_1 = \sigma(a,m,c)$, $S_2 = \sigma(a^2,m,ac+c)$, $S_3 = \sigma(ab,m,ac)$, $S_4 = \sigma(1,m,0) = (m-1)(m-2)/m$, $S_5 = \sigma(a,m,c)$, $S_6 = \sigma(b,m,c)$, $S_7 = -\sigma(a'-1,m,a'c)$, and

$S_8 = -\sigma(a'(a'-1), m, (a')^2 c)$, if $a'a \equiv 1$ (modulo m); and finally

$$\begin{aligned} S_9 &= 12 \sum_{0\le x<m} \left(\!\!\left(\frac{bx+c}{m}\right)\!\!\right)\left(\!\!\left(\frac{a(bx+c)}{m}\right)\!\!\right) \\ &= 12d \sum_{0\le x<m/d} \left(\!\!\left(\frac{x+c_0/d}{m/d}\right)\!\!\right)\left(\!\!\left(\frac{a(x+c_0/d}{m/d}\right)\!\!\right) \\ &= 12d \sum_{0\le x<m/d} \left(\!\!\left(\frac{x}{m/d}\right)\!\!\right) + \frac{c_0}{m} - \frac{1}{2}\delta_{x0}\left(\!\!\left(\frac{a(x+c_0/d)}{m/d}\right)\!\!\right) \\ &= d\left(\sigma(ad, m, ac_0) + 12\frac{c_0}{m}\left(\!\!\left(\frac{ac_0}{d}\right)\!\!\right) - 6\left(\!\!\left(\frac{ac_0}{m}\right)\!\!\right)\right) \end{aligned}$$

where $c_0 = c \bmod d$. The grand total will be near $\frac{1}{6}$ when d is small and when the fractions a/m, $(a^2 \bmod m)/m$, $(ab \bmod m)/m$, b/m, $(a'-1)/m$, $(a'(a'-1) \bmod m)/m$, $((ad) \bmod m)/m$ all have small partial quotients. (Note that $a' - 1 \equiv -b + b^2 - \cdots$, cf. exercise 3.2.1.3–7.)

21. $C = (s - (\frac{1}{2})^2)/(\frac{1}{3} - (\frac{1}{2})^2)$, where

$$\begin{aligned} s_n &= \int_{x_n}^{x_{n+1}} x\{ax+\theta\}\,dx = \frac{1}{a^2}\left(\frac{1}{3} - \frac{\theta}{2} + \frac{n}{2}\right), \qquad \text{if } x_n = \frac{n-\theta}{a}; \\ s &= \int_0^1 x\{ax+\theta\}\,dx = s_0 + s_1 + \cdots + s_{a-1} + \int_{-\theta/a}^0 (ax+\theta)\,dx \\ &= \frac{1}{3a} - \frac{\theta}{2a} + \frac{a-1}{4a} + \frac{\theta^2}{2a}. \end{aligned}$$

Therefore $C = (1 - 6\theta + 6\theta^2)/a$.

22. Let $[u, v)$ denote the set $\{\, x \mid u \le x < v \,\}$. We have $s(x) < x$ in the disjoint intervals

$$\left[\frac{1-\theta}{a}, \frac{1-\theta}{a-1}\right), \quad \left[\frac{2-\theta}{a}, \frac{2-\theta}{a-1}\right), \quad \ldots, \quad \left[\frac{a-\theta}{a}, 1\right),$$

which have total length

$$1 + \sum_{0<j\le a-1} \left(\frac{j-\theta}{a-1}\right) - \sum_{0<j\le a} \left(\frac{j-\theta}{a}\right) = 1 + \frac{a}{2} - \theta - \frac{a+1}{2} + \theta = \frac{1}{2}.$$

23. We have $s(s(x)) < s(x) < x$ when x is in $\left[\frac{k-\theta}{a}, \frac{k-\theta}{a-1}\right)$ and $ax + \theta - k$ is in $\left[\frac{j-\theta}{a}, \frac{j-\theta}{a-1}\right)$, for $0 < j \le k < a$; or when x is in $\left[\frac{a-\theta}{a}, 1\right)$ and $ax + \theta - a$ is either in $\left[\frac{j-\theta}{a}, \frac{j-\theta}{a-1}\right)$ for $0 < j \le \lfloor a\theta \rfloor$ or in $\left[\frac{\lfloor a\theta\rfloor + 1 - \theta}{a}, \theta\right)$. The desired probability is

$$\begin{aligned} &\sum_{0<j\le k<a} \frac{j-\theta}{a^2(a-1)} + \sum_{0<j\le \lfloor a\theta\rfloor} \frac{j-\theta}{a^2(a-1)} + \frac{1}{a^2}\max(0, \{a\theta\} + \theta - 1) \\ &\qquad = \frac{1}{6} + \frac{1}{6a} - \frac{\theta}{2a} + \frac{1}{a^2}\left(\frac{\lfloor a\theta\rfloor(\lfloor a\theta\rfloor + 1 - 2\theta)}{2(a-1)} + \max(0, \{a\theta\} + \theta - 1)\right), \end{aligned}$$

which is $\frac{1}{6} + (1 - 3\theta + 3\theta^2)/6a + O(1/a^2)$ for large a. Note that $1 - 3\theta + 3\theta^2 \ge \frac{1}{4}$, so θ can't be chosen to make this probability come out right.

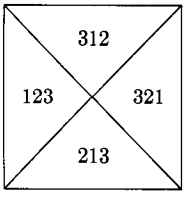


Fig. A–1. Permutation regions for Fibonacci generator.

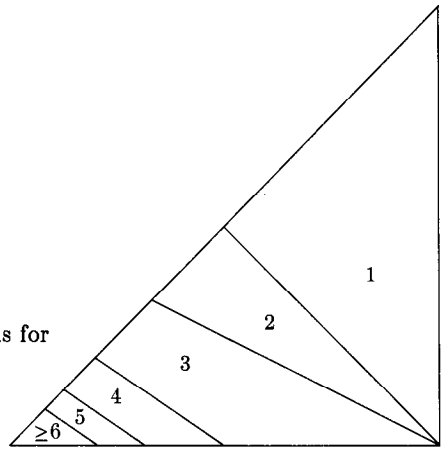


Fig. A–2. Run-length regions for Fibonacci generator.

24. Proceed as in the previous exercise; the sum of the interval lengths is

$$\sum_{0<j_1\le\cdots\le j_{t-1}<a} \frac{j_1}{a^{t-1}(a-1)} = \frac{1}{a^{t-1}(a-1)}\binom{a+t-2}{t}.$$

To compute the average length, let p_k be the probability of a run of length $\ge k$; the average is

$$\sum_{k\ge1} p_k = \sum_{k\ge1}\binom{a+k-2}{k}\frac{1}{a^{k-1}(a-1)} = \left(\frac{a}{a-1}\right)^a - \frac{a}{a-1}.$$

The value for a truly random sequence would be $e-1$; and our value is $e-1+(e/2-1)/a+O(1/a^2)$. [*Note:* The same result holds for an ascending run, since we have $U_n > U_{n+1}$ if and only if $1-U_n < 1-U_{n+1}$. This would lead us to suspect that runs in linear congruential sequences might be slightly longer than normal, so the run test should be applied to such generators.]

25. x must be in the interval $[(k+\alpha'-\theta)/a, (k+\beta'-\theta)/a)$ for some k, and also in the interval $[\alpha,\beta)$. Let $k_0 = \lceil a\alpha+\theta-\beta'\rceil$, $k_1 = \lceil a\beta+\theta-\beta'\rceil$. With due regard to boundary conditions, we get the probability

$$(k_1-k_0)(\beta'-\alpha')/a + \max(0,\beta-(k_1+\alpha'-\theta)/a) - \max(0,\alpha-(k_0+\alpha'-\theta)/a).$$

This is $(\beta-\alpha)(\beta'-\alpha')+\epsilon$, where $|\epsilon| < 2(\beta'-\alpha')/a$.

26. See Fig. A–1; the orderings $U_1 < U_3 < U_2$ and $U_2 < U_3 < U_1$ are impossible; the other four each have probability $\frac{1}{4}$.

27. $U_n = \{F_{n-1}U_0 + F_nU_1\}$. We need to have both $F_{k-1}U_0 + F_kU_1 < 1$ and $F_kU_0 + F_{k+1}U_1 > 1$. The half-unit-square in which $U_0 > U_1$ is broken up as shown in Fig. A–2, with various values of k indicated. The probability for a run of length k is $\frac{1}{2}$, if $k=1$; $1/F_{k-1}F_{k+1} - 1/F_kF_{k+2}$, if $k>1$. The corresponding probabilities for a random sequence are $2k/(k+1)! - 2(k+1)/(k+2)!$; the following table compares the first few values.

k:	1	2	3	4	5
Probability in Fibonacci case:	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{10}$	$\frac{1}{24}$	$\frac{1}{65}$
Probability in random case:	$\frac{1}{3}$	$\frac{5}{12}$	$\frac{11}{60}$	$\frac{19}{360}$	$\frac{29}{2520}$

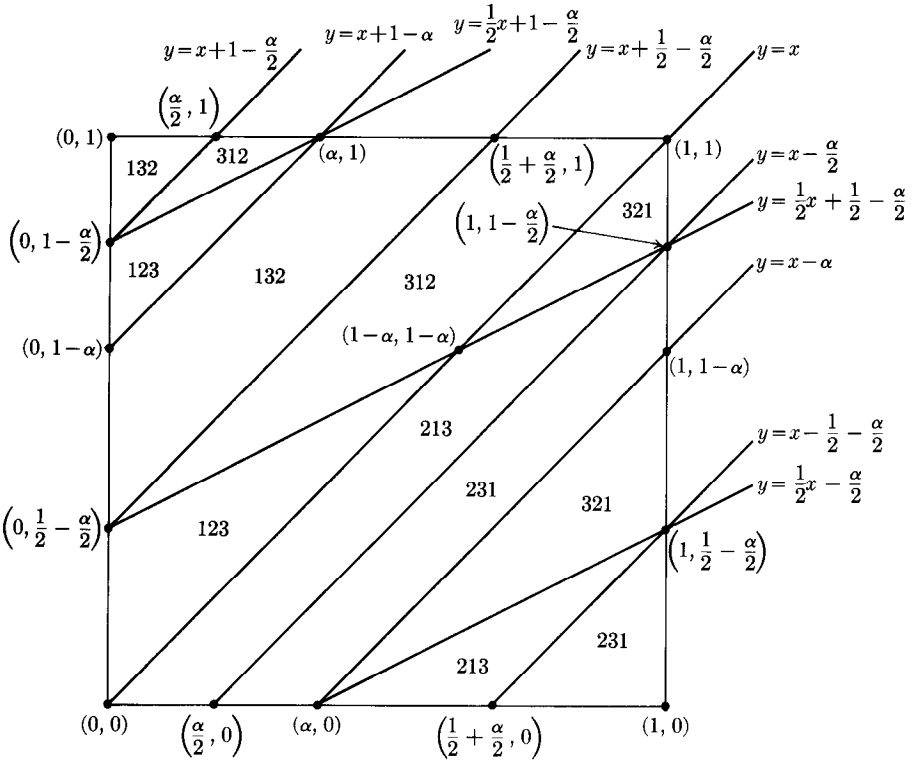


Fig. A–3. Permutation regions for a generator with potency 2; $\alpha = (a-1)c/m$.

28. Fig. A–3 shows the various regions in the general case. The "213" region means $U_2 < U_1 < U_3$, if U_1 and U_2 are chosen at random; the "321" region means that $U_3 < U_2 < U_1$, etc. The probabilities for 123 and 321 are $\frac{1}{4} - \alpha/2 + \alpha^2/2$; the probabilities for all other cases are $\frac{1}{8} + \alpha/4 - \alpha^2/4$. To have all equal to $\frac{1}{6}$, we must have $1 - 6\alpha + 6\alpha^2 = 0$. [This exercise establishes a theorem due to J. N. Franklin, *Math. Comp.* **17** (1963), 28–59, Theorem 13; other results of Franklin's paper are related to exercises 22 and 23.]

SECTION 3.3.4

1. ν_1 is always m and $\mu_1 = 2$, for generators of maximum period.

2. Let V be the matrix whose rows are $V_1, \ldots, V_t$. To minimize $Y \cdot Y$, subject to the condition that $Y \neq (0, \ldots, 0)$ and VY is an integer column vector X, is equivalent to minimizing $(V^{-1}X)\cdot(V^{-1}X)$, subject to the condition that X is a nonzero integer column vector. The columns of V^{-1} are $U_1, \ldots, U_t$.

3. $a^2 \equiv 2a - 1$ and $a^3 \equiv 3a - 2$ (modulo m). By considering all short solutions of (15), we find that $\nu_3^2 = 6$ and $\nu_4^2 = 4$, for the respective vectors $(1, -2, 1)$ and $(1, -1, -1, 1)$, except in the following cases: $m = 2^e q$, q odd, $e \geq 3$, $a \equiv 2^{e-1}$ (modulo 2^e), $a \equiv 1$ (modulo q), $\nu_3^2 = \nu_4^2 = 2$; $m = 3^e q$, $\gcd(3, q) = 1$, $e \geq 2$, $a \equiv 1 \pm 3^{e-1}$ (modulo 3^e), $a \equiv 1$ (modulo q), $\nu_4^2 = 2$; $m = 9$, $a = 4$ or 7, $\nu_2^2 = \nu_3^2 = 5$.

4. (a) The unique choice for (x_1, x_2) is $\frac{1}{m}(y_1 u_{22} - y_2 u_{21}, -y_1 u_{12} + y_2 u_{11})$, and this is $\equiv \frac{1}{m}(y_1 u_{22} + y_2 a u_{22}, -y_1 u_{12} - y_2 a u_{12}) \equiv (0, 0)$ (modulo 1); i.e., x_1 and x_2 are integers. (b) When $(x_1, x_2) \neq (0, 0)$, we have $(x_1 u_{11} + x_2 u_{21})^2 + (x_1 u_{12} + x_2 u_{22})^2 = x_1^2(u_{11}^2 + u_{12}^2) + x_2^2(u_{21}^2 + u_{22}^2) + 2x_1 x_2(u_{11} u_{21} + u_{12} u_{22})$, and by hypothesis this is $\geq (x_1^2 + x_2^2 - |x_1 x_2|)(u_{11}^2 + u_{12}^2) \geq u_{11}^2 + u_{12}^2$. [Note that this is a stronger result than Lemma A, which tells us only that $x_1^2 \leq (u_{11}^2 + u_{12}^2)(u_{21}^2 + u_{22}^2)/m^2$ and that $x_2^2 \leq (u_{11}^2 + u_{12}^2)^2/m^2$, where the latter can be ≥ 1. The idea is essentially Gauss's notion of a reduced binary quadratic form, *Disq. Arith.* (Leipzig: 1801), §171.]

5. Conditions (30) remain invariant; hence h cannot be zero in step S2, when a is relatively prime to m. Since h always decreases in that step, S2 eventually terminates with $u^2 + v^2 \geq s$. Note that $pp' \leq 0$ throughout the calculation.

The hinted inequality surely holds the first time step S2 is performed. The integer q' that minimizes $(h' - qh)^2 + (p' - qp')^2$ is $q' = \text{round}((h'h + p'p)/(h^2 + p^2))$, by (24). If $(h' - q'h)^2 + (p' - q'p)^2 < h^2 + p^2$ we must have $q' \neq 0$, $q' \neq -1$, hence $(p' - q'p)^2 \geq p^2$, hence $(h' - q'h)^2 < h^2$, i.e., $|h' - q'h| < h$, i.e., q' is q or $q + 1$. We have $hu + pv \geq h(h' - q'h) + p(p' - q'p) \geq -\frac{1}{2}(h^2 + p^2)$, so if $u^2 + v^2 < s$ the next iteration of step S2 will preserve the assumption in the hint. If $u^2 + v^2 \geq s > (u - h)^2 + (v - p)^2$, we have $2|h(u - h) + p(v - p)| = 2(h(h - u) + p(p - v)) = (u - h)^2 + (v - p)^2 + h^2 + p^2 - (u^2 + v^2) \leq (u - h)^2 + (v - p)^2 \leq h^2 + p^2$, hence $(u-h)^2+(v-p)^2$ is minimal by exercise 4. Finally if both u^2+v^2 and $(u-h)^2+(v-p)^2$ are $\geq s$, let $u' = h' - q'h$, $v' = p' - q'p$; then $2|hu' + pv'| \leq h^2 + p^2 \leq u'^2 + v'^2$, and $h^2 + p^2$ is minimal by exercise 4.

6. If $u^2+v^2 \geq s > (u-h)^2+(v-p)^2$ in the previous answer, we have $(v-p)^2 > v^2$, hence $(u - h)^2 < u^2$; and if $q = a_j$, so that $h' = a_j h + u$, we must have $a_{j+1} = 1$. It follows that $\nu_2^2 = \min_{0 \leq j < t}(m_j^2 + p_{j-1}^2)$, in the notation of exercise 3.3.3–16.

Now we have $m_0 = m_j p_j + m_{j+1} p_{j-1} = a_j m_j p_{j-1} + m_j p_{j-2} + m_{j+1} p_{j-1} < (a_j + 1 + 1/a_j) m_j p_{j-1} \leq (A + 1 + 1/A) m_j p_{j-1}$, and $m_j^2 + p_{j-1}^2 \geq 2 m_j p_{j-1}$, hence the result.

7. We shall prove, using condition (19), that $U_j \cdot U_k = 0$ for all $k \neq j$ iff $V_j \cdot V_k = 0$ for all $k \neq j$. Assume that $U_j \cdot U_k = 0$ for all $k \neq j$, and let $U_j = \alpha_1 V_1 + \cdots + \alpha_t V_t$. Then $U_j \cdot U_k = \alpha_k$ for all k, hence $U_j = \alpha_j V_j$, and $V_j \cdot V_k = \alpha_j^{-1}(U_j \cdot V_k) = 0$ for all $k \neq j$. A symmetric argument proves the converse.

8. Clearly $\nu_{t+1} \leq \nu_t$ (a fact used implicitly in Algorithm S, since s is not changed when t increases). For $t = 2$ this is equivalent to $(m\mu_2/\pi)^{1/2} \geq (\frac{3}{4} m \mu_3/\pi)^{1/3}$, i.e., $\mu_3 \leq \frac{4}{3}\sqrt{m/\pi}\,\mu_2^{3/2}$. This reduces to $\frac{4}{3} 10^{-4}/\sqrt{\pi}$ with the given parameters, but for large m and fixed μ_2 the bound (39) is better.

9. Let $f(y_1, \ldots, y_t) = \theta$; then $\gcd(y_1, \ldots, y_t) = 1$, so there is an integer matrix W of determinant 1 having $(w_1, \ldots, w_t)$ as its first row. (Prove the latter fact by induction on the magnitude of the smallest nonzero entry in the row.) Now if $X = (x_1, \ldots, x_t)$ is a row vector, we have $XW = X'$ iff $X = X'W^{-1}$, and W^{-1} is an integer matrix of determinant 1, hence the form g defined by WU satisfies $g(x_1, \ldots, x_t) = f(x_1', \ldots, x_t')$; furthermore $g(1, 0, \ldots, 0) = \theta$.

Without loss of generality, assume that $f = g$. If now S is any orthogonal matrix, the matrix US defines the same form as U, since $(XUS)(XUS)^T = (XU)(XU)^T$. Choosing S so that its first column is a multiple of V_1 and its other columns are any suitable vectors, we have

$$US = \begin{pmatrix} \alpha_1 & \alpha_2 & \ldots & \alpha_t \\ 0 & & & \\ \vdots & & U' & \\ 0 & & & \end{pmatrix}$$

for some $\alpha_1, \alpha_2, \ldots, \alpha_t$ and some $(t-1) \times (t-1)$ matrix U'. Hence $f(x_1, \ldots, x_t) = (\alpha_1 x_1 + \cdots + \alpha_t x_t)^2 + h(x_2, \ldots, x_t)$. It follows that $\alpha_1 = \sqrt{\theta}$ [in fact, $\alpha_j = (U_1 \cdot U_j)/\sqrt{\theta}$ for $1 \le j \le t$], and that h is a positive definite quadratic form defined by U', where $\det U' = (\det U)/\sqrt{\theta}$. By induction on t, there are integers $(x_2, \ldots, x_t)$ with $h(x_2, \ldots, x_t) \le (\frac{4}{3})^{(t-2)/2} |\det U|^{2/(t-1)}/\theta^{1/(t-1)}$, and for these integer values we can choose x_1 so that $|x_1 + (\alpha_2 x_2 + \cdots + \alpha_t x_t)/\alpha_1| \le \frac{1}{2}$, i.e., $(\alpha_1 x_1 + \cdots + \alpha_t x_t)^2 \le \frac{1}{4}\theta$. Hence $\theta \le f(x_1, \ldots, x_t) \le \frac{1}{4}\theta + (\frac{4}{3})^{(t-2)/2} |\det U|^{2/(t-1)}/\theta^{1/(t-1)}$ and the desired inequality follows immediately.

[*Note:* For $t = 2$ the result is best possible. For general t, Hermite's theorem implies that $\mu_t \le \pi^{t/2}(4/3)^{t(t-1)/4}/(t/2)!$. A fundamental theorem due to Minkowski ("Every t-dimensional convex set symmetric about the origin with volume $\ge 2^t$ contains a nonzero integer point") gives $\mu_t \le 2^t$; this is stronger than Hermite's theorem for $t \ge 9$. Even stronger results are known, cf. (41).]

10. Since y_1 and y_2 are relatively prime, we can solve $u_1 y_2 - u_2 y_1 = m$; furthermore $(u_1 + q y_1) y_2 - (u_2 + q y_2) y_1 = m$ for all q, so we can ensure that $2|u_1 y_1 + u_2 y_2| \le y_1^2 + y_2^2$ by choosing an appropriate integer q. Now $y_2(u_1 + a u_2) \equiv y_2 u_1 - y_1 u_2 \equiv 0$ (modulo m), and y_2 must be relatively prime to m, hence $u_1 + a u_2 \equiv 0$. Finally let $|u_1 y_1 + u_2 y_2| = \alpha m$, $u_1^2 + u_2^2 = \beta m$, $y_1^2 + y_2^2 = \gamma m$; we have $0 \le \alpha \le \frac{1}{2}\gamma$, and it remains to be shown that $\alpha \le \frac{1}{2}\beta$ and $\beta\gamma \ge 1$. The identity $(u_1 y_2 - u_2 y_1)^2 + (u_1 y_1 + u_2 y_2)^2 = (u_1^2 + u_2^2)(y_1^2 + y_2^2)$ implies that $1 + \alpha^2 = \beta\gamma$. If $\alpha > \frac{1}{2}\beta$, we have $2\alpha\gamma > 1 + \alpha^2$, i.e., $\gamma - \sqrt{\gamma^2 - 1} < \alpha \le \frac{1}{2}\gamma$. But $\frac{1}{2}\gamma < \sqrt{\gamma^2 - 1}$ implies that $\gamma^2 > \frac{4}{3}$, a contradiction.

11. Since a is odd, $y_1 + y_2$ must be even. To avoid solutions with y_1 and y_2 both even, let $y_1 = x_1 + x_2$, $y_2 = x_1 - x_2$, and solve $x_1^2 + x_2^2 = m/\sqrt{3} - \epsilon$, with (x_1, x_2) relatively prime and x_1 even; the corresponding multiplier a will be the solution to $(x_2 - x_1)a \equiv x_2 + x_1$ (modulo 2^e). It is not difficult to prove that $a \equiv 1$ (modulo 2^{k+1}) iff $x_1 \equiv 0$ (modulo 2^k), so we get the best potency when $x_1 \bmod 4 = 2$. The problem reduces to finding relatively prime solutions to $x_1^2 + x_2^2 = N$ where N is a large integer of the form $4k+1$. By factoring N over the Gaussian integers, we can see that solutions exist if and only if each prime factor of N (over the usual integers) has the form $4k + 1$.

According to a famous theorem of Fermat, every prime p of the form $4k + 1$ can be written $p = u^2 + v^2 = (u + iv)(u - iv)$, v even, in a unique way except for the signs of u and v. The numbers u and v can be calculated efficiently by solving $x^2 \equiv -1$ (modulo p), then calculating $u + iv = \gcd(x + i, p)$ by Euclid's algorithm over the Gaussian integers. [We can take $x = n^{(p-1)/4} \bmod p$ for almost half of all integers n. This application of a Euclidean algorithm is essentially the same as finding the least nonzero $u^2 + v^2$ such that $u \pm xv \equiv 0$ (modulo p).] If the prime factorization of N is $p_1^{e_1} \ldots p_r^{e_r} = (u_1 + iv_1)^{e_1}(u_1 - iv_1)^{e_1} \ldots (u_r + iv_r)^{e_r}(u_r - iv_r)^{e_r}$, we get 2^{r-1} distinct solutions to $x_1^2 + x_2^2 = N$, $\gcd(x_1, x_2) = 1$, x_1 even, by letting $|x_2| + i|x_1| = (u_1 + iv_1)^{e_1}(u_2 \pm iv_2)^{e_2} \ldots (u_r \pm iv_r)^{e_r}$; and all such solutions are obtained in this way.

Note: When $m = 10^e$, a similar procedure can be used, but it is five times as much work since we must keep trying until finding a solution with $x_1 \equiv 0$ (modulo 10). For example, when $m = 10^{10}$ we have $\lfloor m/\sqrt{3} \rfloor = 5773502691$, and $5773502689 = 53 \cdot 108934013 = (7+2i)(7-2i)(2203+10202i)(2203-10202i)$. Of the two solutions $|x_2| + i|x_1| = (7+2i)(2203+10202i)$ or $(7+2i)(2203-10202i)$, the former gives $|x_1| = 67008$ (no good) and the latter gives $|x_1| = 75820$, $|x_2| = 4983$ (which is usable). Line 9 of Table 1 was obtained by taking $x_1 = 75820$, $x_2 = -4983$.

Line 20 of the table was obtained as follows: $\lfloor 2^{35}/\sqrt{3} \rfloor = 19837604196$; we drop down to 19837604193, which is divisible by 3 so it is ineligible. Similarly, 19837604189 is divisible by 19, and 19837604185 by 7, and 19837604181 by 3; but 19837604177 is prime and equals $131884^2 + 49439^2$. The corresponding multiplier is 1175245817; a better one could be found if we continued searching. The multiplier on line 24 is the best of the first sixteen multipliers found by this procedure when $m = 2^{32}$.

12. $U_j' \cdot U_j' = U_j \cdot U_j + 2\sum_{i \neq j} q_i (U_i \cdot U_j) + \sum_{i \neq j} \sum_{k \neq j} q_i q_k (U_i \cdot U_k)$. The partial derivative with respect to q_k is twice the left-hand side of (26). If the minimum can be achieved, these partial derivatives must all vanish.

13. $u_{11} = 1$, $u_{21} =$ irrational, $u_{12} = u_{22} = 0$.

14. After three Euclidean steps we find $\nu_2^2 = 5^2 + 5^2$, then S4 produces

$$U = \begin{pmatrix} -5 & 5 & 0 \\ -18 & -2 & 0 \\ 1 & -2 & 1 \end{pmatrix}, \qquad V = \begin{pmatrix} -2 & 18 & 38 \\ -5 & -5 & -5 \\ 0 & 0 & 100 \end{pmatrix}.$$

Transformations $(j, q_1, q_2, q_3) = (1, *, 0, 2)$, $(2, -4, *, 1)$, $(3, 0, 0, *)$, $(1, *, 0, 0)$ result in

$$U = \begin{pmatrix} -3 & 1 & 2 \\ -5 & -8 & -7 \\ 1 & -2 & 1 \end{pmatrix}, \qquad V = \begin{pmatrix} -22 & -2 & 18 \\ -5 & -5 & -5 \\ 9 & -31 & 29 \end{pmatrix}, \qquad Z = (0 \quad 0 \quad 1).$$

Thus $\nu_3 = \sqrt{6}$, as we already knew from exercise 3.

15. The largest achievable q in (11), minus the smallest achievable, plus 1, is $|u_1| + \cdots + |u_t| - \delta$, where $\delta = 1$ if $u_i u_j < 0$ for some i and j, otherwise $\delta = 0$. For example if $t = 5$, $u_1 > 0$, $u_2 > 0$, $u_3 > 0$, $u_4 = 0$, and $u_5 < 0$, the largest achievable value is $q = u_1 + u_2 + u_3 - 1$ and the smallest is $q = u_5 + 1 = -|u_5| + 1$.

[Note that the number of hyperplanes is unchanged when c varies, hence the same answer applies to the problem of covering L instead of L_0. However, the stated formula is *not* always exact for covering L_0, since the hyperplanes that intersect the unit hypercube may not all contain points of L_0. In the example above, we can never achieve the value $q = u_1 + u_2 + u_3 - 1$ in L_0 if $u_1 + u_2 + u_3 > m$; it is achievable iff there is a solution to $m - u_1 - u_2 - u_3 = x_1 u_1 + x_2 u_2 + x_3 u_3 + x_4 |u_5|$ in nonnegative integers (x_1, x_2, x_3, x_4). It may be true that the stated limits are always achievable when $|u_1| + \cdots + |u_t|$ is minimal, but this does not appear to be obvious.]

16. It suffices to determine all solutions to (15) having minimum $|u_1| + \cdots + |u_t|$, subtracting 1 if any one of these solutions has components of opposite sign.

Instead of positive definite quadratic forms, we work with the somewhat similar function $f(x_1, \ldots, x_t) = |x_1 U_1 + \cdots + x_t U_t|$, defining $|Y| = |y_1| + \cdots + |y_t|$. Inequality (21) can be replaced by $|x_k| \leq (\max_{1 \leq j \leq t} |v_{kj}|)\, f(y_1, \ldots, y_t)$.

Thus a workable algorithm can be obtained as follows. Replace steps S1 through S3 by: "Set $U \leftarrow (m)$, $V \leftarrow (1)$, $r \leftarrow 1$, $s \leftarrow m$, $t \leftarrow 1$." (Here U and V are 1×1

matrices; thus the two-dimensional case will be handled by the general method. A special procedure for $t = 2$ could, of course, be devised.) In steps S4 and S8, set $s \leftarrow \min(s, |U_k|)$. In step S8, set $z_k \leftarrow \lfloor \max_{1\le j\le t} |v_{kj}|\, s/m \rfloor$. In step S10, set $s \leftarrow \min(s, |Y| - \delta)$; and in step S11, output $s = N_t$. Otherwise leave the algorithm as it stands, since it already produces suitably short vectors. [*Math. Comp.* **29** (1975), 827–833.]

17. When $k > t$ in S10, and if $Y \cdot Y \le s$, output Y and $-Y$; furthermore if $Y \cdot Y < s$, take back the previous output of vectors for this t. [In the author's experience preparing Table 1, there was exactly one vector (and its negative) output for each ν_t, except when $y_1 = 0$ or $y_t = 0$.]

18. (a) Let $x = m$, $y = (1 - m)/3$, $v_{ij} = y + x\delta_{ij}$, $u_{ij} = -y + \delta_{ij}$. Then $V_j \cdot V_k = \frac{1}{3}(m^2 - 1)$ for $j \ne k$, $V_k \cdot V_k = \frac{2}{3}(m^2 + \frac{1}{2})$, $U_j \cdot U_j = \frac{1}{3}(m^2 + 2)$, $z_k \approx \sqrt{\frac{2}{9}}\, m$. (This example satisfies (28) with $a = 1$ and works for all $m \equiv 1$ (modulo 3).)

(b) Interchange the rôles of U and V in step S5. Also set $s \leftarrow \min(s, U_i \cdot U_i)$ for all U_i that change. For example, when $m = 64$ this transformation with $j = 1$, applied to the matrices of (a), reduces

$$V = \begin{pmatrix} 43 & -21 & -21 \\ -21 & 43 & -21 \\ -21 & -21 & 43 \end{pmatrix}, \quad U = \begin{pmatrix} 22 & 21 & 21 \\ 21 & 22 & 21 \\ 21 & 21 & 22 \end{pmatrix}$$

to

$$V = \begin{pmatrix} 1 & 1 & 1 \\ -21 & 43 & -21 \\ -21 & -21 & 43 \end{pmatrix}, \quad U = \begin{pmatrix} 22 & 21 & 21 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix}.$$

[Since the transformation can increase the length of V_j, an algorithm that incorporates both transformations must be careful to avoid infinite looping. See also exercise 23.]

19. No, since a product of non-identity matrices with all off-diagonal elements non-negative and all diagonal elements 1 cannot be the identity.

[However, looping would be possible if a subsequent transformation with $q = -1$ were performed when $-2V_i \cdot V_j = V_j \cdot V_j$; the rounding rule must be asymmetric with respect to sign if non-shortening transformations are allowed.]

20. Use the ordinary spectral test for a and $m = 2^{e-2}$; cf. exercise 3.2.1.2–9. [On intuitive grounds, the same answer should apply also when $a \bmod 8 = 3$.]

21. $X_{4n+4} \equiv X_{4n}$ (modulo 4), so it is now appropriate to let $V_1 = (4, 4a^2, 4a^3)/m$, $V_2 = (0, 1, 0)$, $V_3 = (0, 0, 1)$ define the corresponding lattice L_0.

24. Let $m = p$; an analysis paralleling the text can be given. For example, when $t = 4$ we have $X_{n+3} = ((a^2 + b)X_{n+1} + abX_n) \bmod n$, and we want to minimize $u_1^2 + u_2^2 + u_3^2 + u_4^2 \ne 0$ such that $u_1 + bu_3 + abu_4 \equiv u_2 + au_3 + (a^2 + b)u_4 \equiv 0$ (modulo m).

Replace steps S1 through S3 by the operations of setting

$$U \leftarrow \begin{pmatrix} m & 0 \\ 0 & m \end{pmatrix}, \quad V \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad R \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad s \leftarrow m^2, \quad t \leftarrow 2,$$

and outputting $\nu_2 = m$. Replace step S4 by

S4′. [Advance t.] If $t = T$, the algorithm terminates. Otherwise set $t \leftarrow t+1$ and $R \leftarrow R\left(\begin{smallmatrix} 0 & b \\ 1 & a \end{smallmatrix}\right) \bmod m$. Set U_t to the new row $(-r_{12}, -r_{22}, 0, \ldots, 0, 1)$ of t elements, and set $u_{it} \leftarrow 0$ for $1 \le i < t$. Set V_t to the new row $(0, \ldots, 0, m)$. For $1 \le i < t$, set $q \leftarrow \text{round}((v_{i1}r_{12} + v_{i2}r_{22})/m)$, $v_{it} \leftarrow v_{i1}r_{12} + v_{i2}r_{22} - qm$, and $U_t \leftarrow U_t + qU_i$. Finally set $s \leftarrow \min(s, U_t \cdot U_t)$, $k \leftarrow t$, $j \leftarrow 1$.

[A similar generalization applies to all sequences of length $p^k - 1$ defined by Eq. 3.2.2–8. Additional numerical examples have been given by A. Grube, *Zeitschrift für angewandte Math. und Mechanik* **53** (1973), T223–T225.]

25. The given sum is at most two times the quantity $\sum_{0 \le k \le m/2d} r(dk) = 1 + \frac{1}{d} f(m/d)$, where

$$f(m) = \frac{1}{m} \sum_{1 \le k \le m/2} \csc(\pi k/m)$$
$$= \frac{1}{m} \int_1^{m/2} \csc(\pi x/m)\, dx + O\Big(\frac{1}{m}\Big) = \frac{1}{\pi} \ln \tan\Big(\frac{\pi}{2m} x\Big)\Big|_1^{m/2} + O\Big(\frac{1}{m}\Big).$$

[When $d = 1$, we have $\sum_{0 \le k < m} r(k) = (2/\pi)\ln m + 1 + (2/\pi)\ln(2e/\pi) + O(1/m)$.]

26. When $m = 1$, we cannot use (52) since k will be zero. If $\gcd(q, m) = d$, the same derivation goes through with m replaced by m/d. Suppose we have $m = p_1^{e_1} \ldots p_r^{e_r}$ and $\gcd(a-1, m) = p_1^{f_1} \ldots p_r^{f_r}$ and $d = p_1^{d_1} \ldots p_r^{d_r}$. If m is replaced by m/d, then s is replaced by $p_1^{\max(0, e_1 - f_1 - d_1)} \ldots p_r^{\max(0, e_r - f_r - d_r)}$.

27. It is convenient to use the following functions: $\rho(x) = 1$ if $x = 0$, $\rho(x) = x$ if $0 < x \le m/2$, $\rho(x) = m - x$ if $m/2 < x < m$; $\text{trunc}(x) = \lfloor x/2 \rfloor$ if $0 \le x \le m/2$, $\text{trunc}(x) = m - \lfloor (m-x)/2 \rfloor$ if $m/2 < x < m$; $L(x) = 0$ if $x = 0$, $L(x) = \lfloor \lg x \rfloor + 1$ if $0 < x \le m/2$, $L(x) = -(\lfloor \lg(m-x) \rfloor + 1)$ if $m/2 < x < m$; and $l(x) = \max(1, 2^{|x|-1})$. Note that $l(L(x)) \le \rho(x) < 2l(L(x))$ and $\rho(x) \le m \sin(\pi x/m) < \pi\rho(x)$ for $0 < x < m$.

Say that a vector $(u_1, \ldots, u_t)$ is *bad* if it is nonzero and satisfies (15); and let $\rho_{\max}$ be the maximum value of $\rho(u_1) \ldots \rho(u_t)$ over all bad $(u_1, \ldots, u_t)$. The vector $(u_1, \ldots, u_t)$ is said to be in class $(L(u_1), \ldots, L(u_t))$. Thus there are at most $(2 \lg m + 1)^t$ classes, and class $(L_1, \ldots, L_t)$ contains at most $l(L_1) \ldots l(L_t)$ vectors. Our proof is based on showing that the bad vectors in each fixed class contribute only $O(1/\rho_{\max})$ to $\sum r(u_1, \ldots, r_t)$; this proves even more than was asked, since $1/\rho_{\max} \le \pi^t r_{\max}$.

Let $\mu = \lfloor \lg \rho_{\max} \rfloor$. The *$\mu$-fold truncation operator* on a vector is defined to be the following operation repeated μ times: "Let j be minimal such that $\rho(u_j) > 1$, and replace u_j by $\text{trunc}(u_j)$; but do nothing if $\rho(u_j) = 1$ for all j." (This operation essentially throws away one bit of information about $(u_1, \ldots, u_t)$.) If $(u_1', \ldots, u_t')$ and $(u_1'', \ldots, u_t'')$ are two vectors of the same class having the same μ-fold truncation, we say they are *similar*; in this case it follows that $\rho(u_1' - u_1'') \ldots \rho(u_t' - u_t'') < 2^\mu \le \rho_{\max}$. For example, any two vectors of the form $((1x_2x_1)_2, 0, m - (1x_3)_2, (101x_5x_4)_2, (1101)_2)$ are similar when m is large and $\mu = 5$; the μ-fold truncation operator successively removes x_1, x_2, x_3, x_4, x_5. Since the difference of two bad vectors satisfies (15), it is impossible for two unequal bad vectors to be similar. Therefore class $(L_1, \ldots, L_t)$ can contain at most $\max(1, l(L_1) \ldots l(L_t)/2^\mu)$ bad vectors. If class $(L_1, \ldots, L_t)$ contains exactly one bad vector $(u_1, \ldots, u_t)$, we have $r(u_1, \ldots, u_t) \le r_{\max} \le 1/\rho_{\max}$; if it contains $\le l(L_1) \ldots l(L_t)/2^\mu$ bad vectors, each of them has $r(u_1, \ldots, u_t) \le 1/\rho(u_1) \ldots \rho(u_t) \le 1/l(L_1) \ldots l(L_t)$.

28. Let $\zeta = e^{2\pi i/(m-1)}$ and let $S_{kl} = \sum_{0\le j<m-1} \omega^{x_{j+l}}\zeta^{jk}$. The analog of (51) is $|S_{k0}| = \sqrt{m}$, hence the analog of (53) is $|N^{-1}\sum_{0\le n<N}\omega^{x_n}| = O((\sqrt{m}\log m)/N)$. The analogous theorem now states that

$$D_N^{(t)} = O\left(\frac{\sqrt{m}(\log m)^{t+1}}{N}\right) + O\big((\log m)^t\, r_{\max}\big), \qquad D_{m-1}^{(t)} = O\big((\log m)^t\, r_{\max}\big).$$

In fact, $D_{m-1}^{(t)} \le \frac{m-2}{m-1}\sum r(u_1,\ldots,u_t)$ [summed over nonzero solutions of (15)] + $\frac{1}{m-1}\sum r(u_1,\ldots,u_t)$ [summed over all nonzero $(u_1,\ldots,u_t)$]. The latter sum is $O(\log m)^t$ by exercise 25 with $d=1$, and the former sum is treated as in exercise 27.

Let us now consider the quantity $R(a) = \sum r(u_1,\ldots,u_t)$ summed over nonzero solutions of (15). Since m is prime, each $(u_1,\ldots,u_t)$ can be a solution to (15) for at most $t-1$ values of a, hence $\sum_{0<a<m} R(a) \le (t-1)\sum r(u_1,\ldots,u_t) = O(t(\log m)^t)$. It follows that the average value of $R(a)$ taken over all $\varphi(m-1)$ primitive roots is $O(t(\log m)^t/\varphi(m-1))$.

Note: In general $1/\varphi(n) = O(\log\log n/n)$; we have therefore proved that *for all prime m and for all T there exists a primitive root a modulo m such that the linear congruential sequence $(1,a,0,m)$ has discrepancy $D_{m-1}^{(t)} = O(m^{-1}T(\log m)^T\log\log m)$ for $1\le t\le T$.* This method of proof does *not* extend to a similar result for linear congruential generators of period 2^e modulo 2^e, since for example the vector $(1,-3,3,-1)$ solves (15) for about $2^{2e/3}$ values of a.

29. $u_1^2+\cdots+u_t^2 \ge \nu_t^2 \ge 2(t-1)$ implies that $\rho(u_1)\ldots\rho(u_t) \ge \sqrt{\nu_t^2-(t-1)} \ge \nu_t/\sqrt{2}$, in the notation of exercise 27.

30. We wish to minimize $q|aq-mp|$ for $1\le q<m$ and $0\le p<a$. In the notation of exercise 4.5.3–42, we have $aq_n - mp_n = (-1)^n Q_{s-n-1}(a_{n+2},\ldots,a_s)$ for $0\le n\le s$. In the range $q_{n-1}\le q<q_n$ we have $|aq-mp| \ge |aq_{n-1}-mp_{n-1}|$; consequently $q|aq-mp| \ge q_{n-1}|aq_{n-1}-mp_{n-1}|$, and the minimum is $\min_{0\le n<s} q_n|aq_n-mp_n| = \min_{0\le n<s} Q_n(a_1,\ldots,a_n)Q_{s-n-1}(a_{n+2},\ldots,a_s)$. By exercise 4.5.3–32 we have $m = Q_n(a_1,\ldots,a_n)a_{n+1}Q_{s-n-1}(a_{n+2},\ldots,a_s) + Q_n(a_1,\ldots,a_n)Q_{s-n-2}(a_{n+3},\ldots,a_s) + Q_{n-1}(a_1,\ldots,a_{n-1})Q_{s-n-1}(a_{n+2},\ldots,a_s)$; and our problem is essentially that of maximizing the quantity $m/Q_n(a_1,\ldots,a_n)Q_{s-n-1}(n+2,\ldots,a_s)$, which lies between a_{n+1} and $a_{n+1}+2$.

31. Equivalently, the conjecture is that all large m can be written $m = Q_n(a_1,\ldots,a_n)$ for some n and some $a_i\in\{1,2,3\}$. For fixed n the 3^n numbers $Q_n(a_1,\ldots,a_n)$ have an average value of order $(1+\sqrt{2}\,)^n$, and their standard deviation is of order $(2.51527)^n$; so the conjecture is almost surely true. S. K. Zaremba conjectured in 1972 that all m can be represented with $a_i\le 5$; T. W. Cusick made some progress on this problem in *Mathematika* **24** (1977), 166–172. It appears that only the cases $m=54$ and $m=150$ require $a_i=5$, and the largest m's that require 4's are 2052, 2370, 5052, and 6234; at least, the author has found representations with $a_i\le 3$ for all other integers less than 2000000. When we require $a_i\le 2$, the average of $Q_n(a_1,\ldots,a_n)$ is $\frac{4}{5}2^n + \frac{1}{5}(-2)^{-n}$, while the standard deviation grows as $(2.04033)^n$. The density of such numbers in the author's experiments (which considered 2^6 blocks of 2^{14} numbers each, for $m\le 2^{20}$) appears to vary between .50 and .65.

[See I. Borosh and H. Niederreiter, to appear, for a computational method that finds multipliers with small partial quotients. They have found 2-bounded solutions with $m=2^e$ for $25\le e\le 35$.]

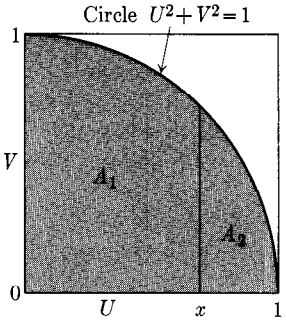


Fig. A–4. Region of "acceptance" for the algorithm of exercise 6.

SECTION 3.4.1

1. $\alpha + (\beta - \alpha)U$.

2. Let $U = X/m$; $\lfloor kU \rfloor = r$ iff $r \le kX/m < r + 1$ iff $mr/k \le X < m(r + 1)/k$ iff $\lceil mr/k \rceil \le X < \lceil m(r + 1)/k \rceil$. The exact probability is given by the formula $(1/m)(\lceil m(r + 1)/k \rceil - \lceil mr/k \rceil) = 1/k + \epsilon$, where $|\epsilon| < 1/m$.

3. If full-word random numbers are given, the result will be sufficiently random as in exercise 2. But if a linear congruential sequence is used, k must be relatively prime to the modulus m, lest the numbers have a very short period, by the results of Section 3.2.1.1. For example, if $k = 2$ and m is even, the numbers will at best be alternately 0 and 1. The method is slower than (1) in nearly every case, so it is not recommended.

4. $\max(X_1, X_2) \le x$ if and only if $X_1 \le x$ and $X_2 \le x$; $\min(X_1, X_2) \ge x$ if and only if $X_1 \ge x$ and $X_2 \ge x$. The probability that two independent events both happen is the product of the individual probabilities.

5. Obtain independent uniform deviates U_1, U_2. Set $X \leftarrow U_2$. If $U_1 \ge p$, set $X \leftarrow \max(X, U_3)$, where U_3 is a third uniform deviate. If $U_1 \ge p + q$, also set $X \leftarrow \max(X, U_4)$, where U_4 is a fourth uniform deviate. This method can obviously be generalized to any polynomial, and indeed even to infinite power series (as shown for example in Algorithm S, which uses minimization instead of maximization).

We could also proceed as follows (suggested by M. D. MacLaren): If $U_1 < p$, set $X \leftarrow U_1/p$; otherwise if $U_1 < p + q$, set $X \leftarrow \max((U_1 - p)/q, U_2)$; otherwise set $X \leftarrow \max((U_1 - p - q)/r, U_2, U_3)$. This method requires less time than the other to obtain the uniform deviates, although it involves further arithmetical operations and it is slightly less stable numerically.

6. $F(x) = A_1/(A_1 + A_2)$, where A_1 and A_2 are the areas in Fig. A–4; so

$$F(x) = \frac{\int_0^x \sqrt{1 - y^2}\, dy}{\int_0^1 \sqrt{1 - y^2}\, dy} = \frac{2}{\pi} \arcsin x + \frac{2}{\pi} x\sqrt{1 - x^2}.$$

The probability of termination at step 2 is $p = \pi/4$, each time step 2 is encountered, so the number of executions of step 2 has the geometric distribution. The characteristics of this number are (min 1, ave $4/\pi$, max ∞, dev $(4/\pi)\sqrt{1 - \pi/4}$), by exercise 17.

7. If $k = 1$, then $n_1 = n$ and the problem is trivial. Otherwise it is always possible to find $i \neq j$ such that $n_i \leq n \leq n_j$. Fill B_i with n_i cubes of color C_i and $n - n_i$ of color C_j, then decrease n_j by $n - n_i$ and eliminate color C_i. We are left with the same sort of problem but with k reduced by 1; by induction, it's possible.

The following algorithm can be used to compute the P and Y tables: Form a list of pairs $(p_1, 1) \ldots (p_k, k)$ and sort it by first components, obtaining a list $(q_1, a_1) \ldots (q_k, a_k)$ where $q_1 \leq \cdots \leq q_k$ and $n = k$. Repeat the following operation until $n = 0$: Set $P[a_1 - 1] \leftarrow kq_1$ and $Y[a_1 - 1] \leftarrow x_{a_n}$. Delete (q_1, a_1) and (q_n, a_n), then insert the new entry $(q_n - (1/k - q_1), a_n)$ into its proper place in the list and decrease n by 1.

(If $p_j \leq 1/k$ the algorithm will never put x_j in the Y table; this fact is used implicitly in Algorithm M. The algorithm attempts to maximize the probability that $V < P_K$ in (3), by always robbing from the richest remaining element and giving it to the poorest. However, it is very difficult to determine the absolute minimum of this probability, since such a task is at least as difficult as the "bin-packing problem"; cf. Chapter 7.)

8. Replace P_j by $(j + P_j)/k$ for $0 \leq j < k$.

9. Consider the sign of $f''(x) = \sqrt{2/\pi}\,(x^2 - 1)e^{-x^2/2}$.

10. Let $S_j = (j - 1)/5$ for $1 \leq j \leq 16$ and $p_{j+15} = F(S_{j+1}) - F(S_j) - p_j$ for $1 \leq j \leq 15$; also let $p_{31} = 1 - F(3)$ and $p_{32} = 0$. (Eq. (15) defines $p_1, \ldots, p_{15}$.) The algorithm of exercise 7 can now be used with $k = 32$ to compute P_j and Y_j, after which we will have $1 \leq Y_j \leq 15$ for $1 \leq j \leq 32$. Set $P_0 \leftarrow P_{32}$ (which is 0) and $Y_0 \leftarrow Y_{32}$. Then set $Z_j \leftarrow 1/(5 - 5P_j)$ and $Y_j \leftarrow \frac{1}{5}Y_j - Z_j$ for $0 \leq j < 32$; $Q_j \leftarrow 1/(5P_j)$ for $1 \leq j \leq 15$.

Let $h = \frac{1}{5}$ and $f_{j+15}(x) = \sqrt{2/\pi}\left(e^{-x^2/2} - e^{-j^2/50}\right)/p_{j+15}$ for $S_j \leq x \leq S_j + h$. Then let $a_j = f_{j+15}(S_j)$ for $1 \leq j \leq 5$, $b_j = f_{j+15}(S_j)$ for $6 \leq j \leq 15$; also $b_j = -hf'_{j+15}(S_j + h)$ for $1 \leq j \leq 5$, and $a_j = f_{j+15}(x_j) + (x_j - S_j)b_j/h$ for $6 \leq j \leq 15$, where x_j is the root of the equation $f'_{j+15}(x_j) = -b_j/h$. Finally set $D_{j+15} \leftarrow a_j/b_j$ for $1 \leq j \leq 15$ and $E_{j+15} \leftarrow 25/j$ for $1 \leq j \leq 5$, $E_{j+15} \leftarrow 1/(e^{(2j-1)/50} - 1)$ for $6 \leq j \leq 15$.

Table 1 was computed while making use of the following intermediate values: $(p_1, \ldots, p_{31}) = (.156, .147, .133, .116, .097, .078, .060, .044, .032, .022, .014, .009, .005, .003, .002, .002, .005, .007, .009, .010, .009, .009, .008, .006, .005, .004, .002, .002, .001, .001, .003)$; $(x_6, \ldots, x_{15}) = (1.115, 1.304, 1.502, 1.700, 1.899, 2.099, 2.298, 2.497, 2.697, 2.896)$; $(a_1, \ldots, a_{15}) = (7.5, 9.1, 9.5, 9.8, 9.9, 10.0, 10.0, 10.1, 10.1, 10.1, 10.1, 10.2, 10.2, 10.2, 10.2)$; $b_1, \ldots, b_{15}) = (14.9, 11.7, 10.9, 10.4, 10.1, 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.7, 10.8, 10.9)$.

11. Let $g(t) = e^{9/2}te^{-t^2/2}$ for $t \geq 3$. Since $G(x) = \int_0^x g(t)\,dt = 1 - e^{-(x^2-9)/2}$, a random variable X with density g can be computed by setting $X \leftarrow G^{-1}(1 - V) = \sqrt{9 - 2\ln V}$. Now $e^{-t^2/2} \leq (t/3)e^{-t^2/2}$ for $t \geq 3$, so we obtain a valid rejection method if we accept X with probability $f(X)/cg(X) = 3/X$.

12. We have $f'(x) = xf(x) - 1 < 0$ for $x \geq 0$, since $f(x) = x^{-1} - e^{x^2/2}\int_x^\infty e^{-t^2/2}\,dt/t^2$ for $x > 0$. Let $x = a_{j-1}$ and $y^2 = x^2 + 2\ln 2$; then

$$\sqrt{2/\pi}\int_y^\infty e^{-t^2/2}\,dt = \tfrac{1}{2}\sqrt{2/\pi}\,e^{-x^2/2}f(y) < \tfrac{1}{2}\sqrt{2/\pi}\,e^{-x^2/2}f(x) = 2^{-j},$$

hence $y > a_j$.

13. Take $b_j = \mu_j$; consider now the problem with $\mu_j = 0$ for each j. In matrix notation, if $Y = AX$, where $A = (a_{ij})$, we need $AA^T = C = (c_{ij})$. (In other notation, if $Y_j = \sum a_{jk}X_k$, then the average value of Y_iY_j is $\sum a_{ik}a_{jk}$.) If this matrix equation can be solved for A, it can be solved when A is triangular, since $A = BU$ for some orthogonal matrix U and some triangular B, and $BB^T = C$. The desired triangular solution can be obtained by solving the equations $a_{11}^2 = c_{11}$, $a_{11}a_{21} = c_{12}$, $a_{21}^2 + a_{22}^2 = c_{22}$, $a_{11}a_{31} = c_{13}$, $a_{21}a_{31} + a_{22}a_{32} = c_{23}$, ..., successively for a_{11}, a_{21}, a_{22}, a_{31}, a_{32}, etc. [*Note:* The covariance matrix must be positive semidefinite, since the average value of $(\sum y_jY_j)^2$ is $\sum c_{ij}y_iy_j$, which must be nonnegative. And there is always a solution when C is positive semidefinite, since $C = U^{-1}\mathrm{diag}(\lambda_1, \ldots, \lambda_n)U$, where the eigenvalues λ_j are nonnegative, and $U^{-1}\mathrm{diag}(\sqrt{\lambda_1}, \ldots, \sqrt{\lambda_n})U$ is a solution.]

14. $F(x/c)$ if $c > 0$, a step function if $c = 0$, and $1 - F(x/c)$ if $c < 0$.

15. Distribution $\int_{-\infty}^{\infty} F_1(x-t)\,dF_2(t)$. Density $\int_{-\infty}^{\infty} f_1(x-t)f_2(t)\,dt$. This is called the *convolution* of the given distributions.

16. It is clear that $f(t) \le cg(t)$ for all t as required. Since $\int_0^\infty g(t)\,dt = 1$ we have $g(t) = Ct^{a-1}$ for $0 \le t < 1$, Ce^{-t} for $t \ge 1$, where $C = ae/(a+e)$. A random variable with density g is easy to obtain as a mixture of two distributions, $G_1(x) = x^a$ for $0 \le x < 1$, and $G_2(x) = 1 - e^{1-x}$ for $x \ge 1$:

G1. [Initialize.] Set $p \leftarrow e/(a+e)$. (This is the probability that G_1 should be used.)

G2. [Generate G deviate.] Generate independent uniform deviates U, V, where $V \neq 0$. If $U < p$, set $X \leftarrow V^{1/a}$ and $q \leftarrow e^{-X}$; otherwise set $X \leftarrow 1 - \ln V$ and $q \leftarrow X^{a-1}$. (Now X has density g, and $q = f(X)/cg(X)$.)

G3. [Reject?] Generate a new uniform deviate U. If $U \ge q$, return to G2. ∎

The average number of iterations is $c = (a+e)/(e\,\Gamma(a+1)) < 1.4$.

It is possible to streamline this procedure in several ways. First, we can replace V by an exponential deviate Y of mean 1, generated by Algorithm S, say, and then we set $X \leftarrow e^{-Y/a}$ or $X \leftarrow 1 + Y$ in the two cases. Moreover, if we set $q \leftarrow pe^{-X}$ in the first case and $q \leftarrow p + (1-p)X^{a-1}$ in the second, we can use the original U instead of a newly generated one in step G3. Finally if $U < p/e$ we can accept $V^{1/a}$ immediately, avoiding the calculation of q about 30 percent of the time.

17. (a) $F(x) = 1 - (1-p)^{\lfloor x \rfloor}$, for $x \ge 0$. (b) $G(z) = pz/(1-(1-p)z)$. (c) Mean $1/p$, standard deviation $\sqrt{1-p}/p$. To do the latter calculation, observe that if $H(z) = q + (1-q)z$, then $H'(1) = 1 - q$ and $H''(1) + H'(1) - (H'(1))^2 = q(1-q)$, so the mean and variance of $1/H(z)$ are $q - 1$ and $q(q-1)$, respectively. (See Section 1.2.10.) In this case, $q = 1/p$; the extra factor z in the numerator of $G(z)$ increases the mean by one.

18. Set $N \leftarrow N_1 + N_2 - 1$, where N_1 and N_2 independently have the geometric distribution for probability p. (Consider the generating function.)

19. Set $N \leftarrow N_1 + \cdots + N_t - t$, where the N_j have the geometric distribution for p. (This is the number of failures before the tth success, when a sequence of independent trials are made each of which succeeds with probability p.)

For $t = p = \frac{1}{2}$, and in general when the mean value (namely $t(1-p)/p$) of the distribution is small, we can simply evaluate the probabilities $p_n = \binom{t-1+n}{n} p^t(1-p)^n$ consecutively for $n = 0, 1, 2, \ldots$ as in the following algorithm:

B1. [Initialize.] Set $N \leftarrow 0$, $q \leftarrow p^t$, $r \leftarrow q$, and generate a random uniform deviate U. (We will have $q = p_N$ and $r = p_0 + \cdots + p_N$ during this algorithm, which stops as soon as $U < r$.)

B2. [Search.] If $U \geq r$, set $N \leftarrow N+1$, $q \leftarrow q(1-p)(t-1+N)/N$, $r \leftarrow r+q$, and repeat this step. ∎

[An interesting technique for the negative binomial distribution, for arbitrarily large real values of t, has been suggested by R. Léger: First generate a random gamma deviate X of order t, then let N be a random Poisson deviate of mean $X(1-p)/p$.]

20. R1 $= 1 + (1 - A/R) \cdot$ R1. When R2 is performed, the algorithm terminates with probability I/R; when R3 is performed, it goes to R1 with probability E/R. We have

R1	R/A	R/A	R/A	R/A
R2	0	R/A	0	R/A
R3	0	0	R/A	$R/A - I/A$
R4	R/A	$R/A - I/A$	$R/A - E/A$	$R/A - I/A - E/A$

21. $R = \sqrt{8/e} \approx 1.71153$; $A = \sqrt{\pi/2} \approx 1.25331$. Since

$$\int u\sqrt{a-bu}\,du = (a-bu)^{3/2}(\tfrac{2}{5}(a-bu) - \tfrac{2}{3})/b^2,$$

we have $I = \int_0^{a/b} u\sqrt{a-bu}\,du = \frac{4}{15}a^{5/2}/b^2$ where $a = 4(1+\ln c)$ and $b = 4c$; when $c = e^{1/4}$, I has its maximum value $\frac{5}{6}\sqrt{5/e} \approx 1.13020$. Finally the following integration formulas are needed for E:

$$\int \sqrt{bu-au^2}\,du = \tfrac{1}{8}b^2a^{-3/2}\arcsin(2ua/b-1) + \tfrac{1}{4}ba^{-1}\sqrt{bu-au^2}\,(2ua/b-1),$$

$$\int \sqrt{bu+au^2}\,du = -\tfrac{1}{8}b^2a^{-3/2}\ln\bigl(\sqrt{bu+au^2} + u\sqrt{a} + b/2\sqrt{a}\bigr) + \tfrac{1}{4}ba^{-1}\sqrt{bu+au^2}\,(2ua/b+1),$$

where $a, b > 0$. Let the test in step R3 be "$X^2 \geq 4e^{x-1}/U - 4x$"; then the exterior region hits the top of the rectangle when $u = r(x) = (e^x - \sqrt{e^{2x} - 2ex})/2ex$. (Incidentally, $r(x)$ reaches its maximum value at $x = 1/2$, a point where it is *not* differentiable!) We have $E = \int_0^{r(x)}(\sqrt{2/e} - \sqrt{bu-au^2})\,du$ where $b = 4e^{x-1}$ and $a = 4x$. The maximum value of E occurs near $x = -.35$, where we have $E \approx .29410$.

22. (Solution by G. Marsaglia.) Consider the "continuous Poisson distribution" defined by $G(x) = \int_\mu^\infty e^{-t}t^{x-1}\,dt/\Gamma(x)$, for $x > 0$; if X has this distribution then $\lfloor X \rfloor$ is Poisson distributed, since $G(x+1) - G(x) = e^{-\mu}\mu^x/x!$. If μ is large, G is approximately normal, hence $G^{-1}(F_\mu(x))$ is approximately linear, where $F_\mu(x)$ is the distribution function for a normal deviate with mean and variance μ; i.e., $F_\mu(x) = F((x-\mu)/\sqrt{\mu})$, where $F(x)$ is the normal distribution function (10). Let $g(x)$ be an efficiently computable function such that $|G^{-1}(F_\mu(x)) - g(x)| < \epsilon$ for $-\infty < x < \infty$; we can now generate Poisson deviates efficiently as follows: Generate a normal deviate X, and set $Y \leftarrow g(\mu + \sqrt{\mu}X)$, $N \leftarrow \lfloor Y \rfloor$, $M \leftarrow \lfloor Y + \frac{1}{2} \rfloor$. If $|Y - M| > \epsilon$, output N; otherwise output $M-1$ or M, according as $G^{-1}(F(X)) < M$ or not.

This approach applies also to the binomial distribution, with

$$G(x) = \int_p^1 u^{x-1}(1-u)^{n-x}\,du\,\Gamma(t+1)/\Gamma(x)\Gamma(t+1-x),$$

since $\lfloor G^{-1}(U) \rfloor$ is binomial with parameters (t,p) and G is approximately normal.

[See also the alternative method proposed by Ahrens and Dieter in *Computing* (1980), to appear.]

23. Yes. The second method calculates $|\cos 2\theta|$, where θ is uniformly distributed between 0 and $\pi/2$. (Let $U = r\cos\theta$, $V = r\sin\theta$.)

25. $\frac{21}{32} = (.10101)_2$. In general, the binary representation is formed by using 1 for $\vee$ and 0 for $\wedge$, from left to right, then suffixing 1. This technique [cf. K. D. Tocher, *J. Roy. Stat. Soc.* **B-16** (1954), 49] can lead to efficient generation of independent bits having a given probability p, and it can also be applied to the geometric and binomial distributions.

26. (a) True, $\sum_k \Pr(N_1 = k)\Pr(N_2 = n-k) = e^{-\mu_1-\mu_2}(\mu_1+\mu_2)^n/n!$. (b) False, unless $\mu_2 = 0$; otherwise $N_1 - N_2$ might be negative.

27. Let the binary representation of p be $(.b_1b_2b_3\ldots)_2$, and proceed according to the following rules:

B1. [Initialize.] Set $m \leftarrow t$, $N \leftarrow 0$, $j \leftarrow 1$. (During this algorithm, m represents the number of simulated uniform deviates whose relation to p is still unknown, since they match p in their leading $j-1$ bits; and N is the number of simulated deviates known to be less than p.)

B2. [Look at next column of bits.] Generate a random integer M with the binomial distribution $(m, \frac{1}{2})$. (Now M represents the number of unknown deviates that fail to match b_j.) Set $m \leftarrow m - M$, and if $b_j = 1$ set $N \leftarrow N + M$.

B3. [Done?] If $m = 0$, or if the remaining bits $(.b_{j+1}b_{j+2}\ldots)_2$ of p are all zero, the algorithm terminates. Otherwise, set $j \leftarrow j+1$ and return to step B2. ∎

[When $b_j = 1$ for infinitely many j, the average number of iterations A_t satisfies

$$A_0 = 0; \qquad A_n = 1 + \frac{1}{2^n}\sum_k \binom{n}{k} A_k, \quad \text{for } n \geq 1.$$

Letting $A(z) = \sum A_n z^n/n!$, we have $A(z) = e^z - 1 + A(\frac{1}{2}z)e^{z/2}$. Therefore $A(z)e^{-z} = 1 - e^{-z} + A(\frac{1}{2}z)e^{-z/2} = \sum_{k\geq 0}(1-e^{-z/2^k}) = 1 - e^{-z} - \sum_{n\geq 1}(-z)^n/(n!(2^n-1))$, and

$$A_m = 1 + \sum_{k\geq 1}\binom{n}{k}\frac{(-1)^{k+1}}{2^k-1} = 1 + \frac{V_{n+1}}{n+1} = \lg n + \frac{\gamma}{\ln 2} + \frac{1}{2} + f_0(n) + O(n^{-1})$$

in the notation of exercise 5.2.2–48.]

28. Generate a random point $(y_1,\ldots,y_n)$ on the unit sphere, and let $\rho = \sqrt{\sum a_k y_k^2}$. Generate an independent uniform deviate U, and if $\rho^{n+1}U < K\sqrt{\sum a_k^2 y_k^2}$, output the point $(y_1/\rho,\ldots,y_n/\rho)$; otherwise start over. Here $K^2 = \min\{(\sum a_k y_k^2)^{n+1}/(\sum a_k^2 y_k^2) \mid \sum y_k^2 = 1\} = a_n^{n-1}$ if $na_n \geq a_1$, $((n+1)/(a_1+a_n))^{n+1}(a_1a_n/n)^n$ otherwise.

29. Let $X_{n+1} = 1$, then set $X_k \leftarrow X_{k+1}U_k^{1/k}$ or $X_k \leftarrow X_{k+1}e^{-Y_k/k}$ for $k = n, n-1, \ldots, 1$, where U_k is uniform or Y_k is exponential. [*ACM Trans. Math. Software*, to appear.]

SECTION 3.4.2

1. There are $\binom{N-t}{n-m}$ ways to pick $n-m$ records from the last $N-t$; $\binom{N-t-1}{n-m-1}$ ways to pick $n-m-1$ from $N-t-1$ after selecting the $(t+1)$st item.

2. Step S3 will never go to step S5 when the number of records left to be examined is equal to $n-m$.

3. We should not confuse "conditional" and "unconditional" probabilities. The quantity m depends randomly on the selections that took place among the first t elements; if we take the average over all possible choices that could have occurred among these elements, we will find that $(n-m)/(N-t)$ is exactly n/N on the average. For example, consider the second element; if the first element was selected in the sample (this happens with probability n/N), the second element is selected with probability $(n-1)/(N-1)$; if the first element was not selected, the second is selected with probability $n/(N-1)$. The overall probability of selecting the second element is $(n/N)((n-1)/(N-1)) + (1-n/N)(n/(N-1)) = n/N$.

4. From the algorithm,

$$p(m, t+1) = \Big(1 - \frac{n-m}{N-t}\Big)p(m,t) + \frac{n-(m-1)}{N-t}\,p(m-1,t).$$

The desired formula can be proved by induction on t. In particular, $p(n, N) = 1$.

5. In the notation of exercise 4, the probability that $t = k$ at termination is $q_k = p(n,k) - p(n,k-1) = \binom{k-1}{n-1}/\binom{N}{n}$. The average is $\sum_{0\le k\le N} kq_k = (N+1)n/(n+1)$.

6. Similarly, $\sum_{0\le k\le N} k(k+1)q_k = (N+2)(N+1)n/(n+2)$; the variance is therefore $(N+1)(N-n)n/(n+2)(n+1)^2$.

7. Suppose the choice is $1 \le x_1 < x_2 < \cdots < x_n \le N$. Let $x_0 = 0$, $x_{n+1} = N+1$. The choice is obtained with probability $p = \prod_{1\le t\le N} p_t$, where

$$p_t = \frac{N-(t-1)-n+m}{N-(t-1)}, \qquad \text{for } x_m < t < x_{m+1};$$
$$p_t = \frac{n-m}{N-(t-1)}, \qquad \text{for } t = x_{m+1}.$$

The denominator of the product p is $N!$; the numerator contains the terms $N-n$, $N-n-1, \ldots, 1$ for those t's that are not x's, and the terms $n, n-1, \ldots, 1$ for those t's that are x's. Hence $p = (N-n)!n!/N!$. *Example:* $n = 3$, $N = 8$, $(x_1, x_2, x_3) = (2, 3, 7)$; $p = \frac{5}{8}\frac{3}{7}\frac{2}{6}\frac{2}{5}\frac{4}{4}\frac{3}{3}\frac{2}{2}\frac{1}{1}\frac{1}{1}$.

9. The reservoir gets seven records: 1, 2, 3, 5, 9, 13, 16. The final sample consists of records 2, 5, 16.

10. Delete step R6 and the variable m. Replace the I table by a table of records, initialized to the first n records in step R1, and with the new record replacing the Mth table entry in step R4.

11. Arguing as in Section 1.2.10, which considers the special case $n = 1$, we see that the generating function is

$$G(z) = z^n\left(\frac{1}{n+1} + \frac{n}{n+1}z\right)\left(\frac{2}{n+2} + \frac{n}{n+2}z\right)\ldots\left(\frac{N-n}{N} + \frac{n}{N}z\right).$$

The mean is $n + \sum_{n<t\le N}(n/t) = n(1 + H_N - H_n)$; and the variance turns out to be $n(H_N - H_n) - n^2(H_N^{(2)} - H_n^{(2)})$.

12. (Note that $\pi^{-1} = (b_t t)\ldots(b_3 3)(b_2 2)$, so we seek an algorithm that goes from the representation of π to that for π^{-1}.) Set $b_j \leftarrow j$ for $1 \le j \le t$. Then for $j = 2$, 3, ..., t (in this order) interchange $b_j \leftrightarrow b_{a_j}$. Finally for $j = t$, ..., 3, 2 (in this order) set $b_{a_j} \leftarrow b_j$. (The algorithm is based on the fact that $(a_t t)\pi_1 = \pi_1(b_t t)$.)

13. Renumbering the deck 0, 1, ..., $2n - 2$, we find that s takes card number x into card number $(2x) \bmod (2n-1)$, while c takes card x into $(x+1) \bmod (2n-1)$. We have (c followed by s) $= cs = sc^2$. Therefore any product of c's and s's can be transformed into the form $s^i c^k$. Also $2^{\varphi(2n-1)} \equiv 1$ (modulo $(2n-1)$); since $s^{\varphi(2n-1)}$ and c^{2n-1} are the identity permutation, at most $(2n-1)\varphi(2n-1)$ arrangements are possible. (The *exact* number of different arrangements is $(2n-1)k$, where k is the order of 2 modulo $(2n-1)$. For if $s^k = c^j$, then c^j fixes the card 0, so $s^k = c^j =$ identity.) For further details, see *SIAM Review* **3** (1961), 293–297.

14. Set $Y_j \leftarrow j$ for $t - n < j \le t$. Then for $j = t$, $t-1$, ..., $t - n + 1$ do the following operations: Set $k \leftarrow \lfloor jU \rfloor + 1$. If $k > t - n$ then set $X_j \leftarrow Y_k$ and $Y_k \leftarrow Y_j$; otherwise if $k = X_i$ for some $i > j$ (a symbol table algorithm could be used), then set $X_j \leftarrow Y_i$ and $Y_i \leftarrow Y_j$; otherwise set $X_j \leftarrow k$. (The idea is to let $Y_{t-n+1}, \ldots, Y_j$ represent $X_{t-n+1}, \ldots, X_j$, and if $i > j$ and $X_i \le t - n$ also to let Y_i represent X_{X_i}, in the execution of Algorithm P.)

15. We may assume that $n \le \frac{1}{2}N$, otherwise it suffices to find the $N - n$ elements *not* in the sample. Using a hash table of size $2n$, the idea is to generate random numbers between 1 and N, storing them in the table and discarding duplicates, until n distinct numbers have been generated. The average number of random numbers generated is $N/N + N/(N-1) + \cdots + N/(N-n+1) < 2n$, by exercise 3.3.2–10, and the average time to process each number is $O(1)$. We want to output the results in increasing order, and this can be done as follows: Using an ordered hash table (exercise 6.4–66) with linear probing, the hash table will appear as if the values had been inserted in increasing order. Thus if we use a monotonic hash address such as $\lceil nk/N \rceil$ for the key k, it will be a simple matter to output the keys in sorted order by making at most two passes over the table.

16. In most cases it will be best simply to select n records one at a time by Walker's alias method, with probabilities proportional to their weights, rejecting an element that occurs more than once. But if some weights are significantly larger than others, the following algorithm will sometimes be better [cf. Wong and Easton, *SIAM J. Comput.* **9** (1980), 111–113]. Begin with a "complete binary tree" of weights x_i, where $x_i = x_{2i} + x_{2i+1}$ for $1 \le i < N$ and $x_{N+i-1} = w_i$ for $1 \le i \le N$; then do the following operation n times: "Set $j \leftarrow 1$, and generate $X = Ux_1$. If $X < x_{2j}$, set $j \leftarrow 2j$, otherwise set $X \leftarrow X - x_{2j}$ and $j \leftarrow 2j + 1$; repeat this until $j \ge N$. Then select element $j - N + 1$, set $i \leftarrow \lfloor j/2 \rfloor$, and while $i \ge 1$ set $x_i \leftarrow x_i - x_j$ and $i \leftarrow \lfloor i/2 \rfloor$. Finally set $x_j \leftarrow 0$."

SECTION 3.5

1. A b-ary sequence, yes (cf. exercise 2); a $[0, 1)$ sequence, no (since only finitely many values are assumed by the elements).

2. It is 1-distributed and 2-distributed, but not 3-distributed (the binary number 111 never appears).

3. Cf. exercise 3.2.2–17; repeat the sequence there with a period of length 27.

4. The sequence begins $\frac{1}{3}, \frac{2}{3}, \frac{2}{3}, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{2}{3}, \frac{2}{3}, \frac{2}{3}, \frac{2}{3}, \frac{2}{3}, \frac{2}{3}, \frac{2}{3}$, etc. When $n = 1, 3, 7, 15, \ldots$ we have $\nu(n) = 1, 1, 5, 5, \ldots$ so that $\nu(2^{2k-1} - 1) = \nu(2^{2k} - 1) = (2^{2k} - 1)/3$; hence $\nu(n)/n$ oscillates between $\frac{1}{3}$ and approximately $\frac{2}{3}$, and no limit exists. The probability is undefined.

[The methods of Section 4.2.4 show, however, that a numerical value *can* meaningfully be assigned to

$$\Pr(U_n < \frac{1}{2}) = \Pr(\text{leading digit of the radix-4 representation of } n+1 \text{ is } 1),$$

namely $\log_4 2 = \frac{1}{2}$.]

5. If $\nu_1(n)$, $\nu_2(n)$, $\nu_3(n)$, $\nu_4(n)$ are the counts corresponding to the four probabilities, we have $\nu_1(n) + \nu_2(n) = \nu_3(n) + \nu_4(n)$ for all n. So the desired result follows by addition of limits.

6. By exercise 5 and induction,

$$\Pr(S_j(n) \text{ for some } j,\ 1 \le j \le k) = \sum_{1 \le j \le k} \Pr(S_j(n)).$$

As $k \to \infty$, the latter is a monotone sequence bounded by 1, so it converges; and

$$\underline{\Pr}(S_j(n) \text{ for some } j \ge 1) \ge \sum_{1 \le j \le k} \Pr(S_j(n))$$

for all k. For a counterexample to equality, it is not hard to arrange things so that $S_j(n)$ is always true for *some* j, yet $\Pr(S_j(n)) = 0$ for *all* j.

7. Let $p_i = \sum_{j \ge 1} \Pr(S_{ij}(n))$. The result of the preceding exercise can be generalized to $\underline{\Pr}(S_j(n) \text{ for some } j \ge 1) \ge \sum_{j \ge 1} \underline{\Pr}(S_j(n))$, for *any* disjoint statements $S_j(n)$. So we have $1 = \Pr(S_{ij}(n) \text{ for some } i, j \ge 1) \ge \sum_{i \ge 1} \underline{\Pr}(S_{ij}(n) \text{ for some } j \ge 1) \ge \sum_{i \ge 1} p_i = 1$, and hence $\underline{\Pr}(S_{ij}(n) \text{ for some } j \ge 1) = p_i$. Given $\epsilon > 0$, let I be large enough so that $\sum_{1 \le i \le I} p_i \ge 1 - \epsilon$. Let

$$\phi_i(N) = (\text{number of } n < N \text{ with } S_{ij}(n) \text{ true for some } j \ge 1)/N.$$

Clearly $\sum_{1 \le i \le I} \phi_i(N) \le 1$, and for all large enough N we have $\sum_{2 \le i \le I} \phi_i(N) \ge \sum_{2 \le i \le I} p_i - \epsilon$; hence $\phi_1(N) \le 1 - \phi_2(N) - \cdots - \phi_I(N) \le 1 - p_2 - \cdots - p_I + \epsilon \le 1 - (1 - \epsilon - p_1) + \epsilon = p_1 + 2\epsilon$. This proves that $\overline{\Pr}(S_{1j}(n) \text{ for some } j \ge 1) \le p_1 + 2\epsilon$; hence $\Pr(S_{1j}(n) \text{ for some } j \ge 1) = p_1$, and the desired result holds for $i = 1$. By symmetry of the hypotheses, it holds for any value of i.

8. Add together the probabilities for j, $j + d$, $j + 2d$, ... in Definition E.

9. $\limsup_{n\to\infty}(a_n+b_n)\le\limsup_{n\to\infty}a_n+\limsup_{n\to\infty}b_n$; hence we find that

$$\limsup_{n\to\infty}((y_{1n}-\alpha)^2+\cdots+(y_{mn}-\alpha)^2)\le m\alpha^2-2m\alpha^2+m\alpha^2=0,$$

and this can happen only if each $(y_{jn}-\alpha)$ tends to zero.

10. In the evaluation of the sum in Eq. (22).

11. $\langle U_{2n}\rangle$ is k-distributed if $\langle U_n\rangle$ is $(2,2k)$-distributed.

12. Let $f(x_1,\ldots,x_k)=1$ if $u\le\max(x_1,\ldots,x_k)<v$; $f(x_1,\ldots,x_k)=0$ otherwise. Then apply Theorem B.

13. Let

$$\begin{aligned}p_k&=\Pr(U_n\text{ begins a gap of length }k-1)\\&=\Pr(U_{n-1}\in[\alpha,\beta),\ U_n\notin[\alpha,\beta),\ \ldots,\ U_{n+k-2}\notin[\alpha,\beta),\ U_{n+k-1}\in[\alpha,\beta))\\&=p^2(1-p)^{k-1}.\end{aligned}$$

It remains to translate this into the probability that $f(n)-f(n-1)=k$. Let $\nu_k(n)=$ (number of $j\le n$ with $f(j)-f(j-1)=k$); let $\mu_k(n)=$ (number of $j\le n$ with U_j the beginning of a gap of length $k-1$); and let $\mu(n)$ similarly count the number of $1\le j\le n$ with $U_j\in[\alpha,\beta)$. We have $\mu_k(f(n))=\nu_k(n)$, $\mu(f(n))=n$. As $n\to\infty$, we must have $f(n)\to\infty$, hence

$$\nu_k(n)/n=(\mu_k(f(n))/f(n))\cdot(f(n)/\mu(f(n)))\to p_k/p=p(1-p)^{k-1}.$$

[We have only made use of the fact that the sequence is $(k+1)$-distributed.]

14. Let

$$\begin{aligned}p_k&=\Pr(U_n\text{ begins a run of length }k)\\&=\Pr(U_{n-1}>U_n<\cdots<U_{n+k-1}>U_{n+k})\\&=\frac{1}{(k+2)!}\left(\binom{k+2}{1}\binom{k+1}{1}-\binom{k+2}{1}-\binom{k+2}{1}+1\right)\\&=\frac{k}{(k+1)!}-\frac{k+1}{(k+2)!}\end{aligned}$$

(cf. exercise 3.3.2–13). Now proceed as in the previous exercise to transfer this to $\Pr(f(n)-f(n-1)=k)$. [We have assumed only that the sequence is $(k+2)$-distributed.]

15. For $s,t\ge0$ let

$$\begin{aligned}p_{st}&=\Pr(X_{n-2t-3}=X_{n-2t-2}\ne X_{n-2t-1}\ne\cdots\ne X_{n-1}\\&\qquad\qquad\text{and }X_n=\cdots=X_{n+s}\ne X_{n+s+1})\\&=2^{-s-2t-3};\end{aligned}$$

for $t\ge0$ let $q_t=\Pr(X_{n-2t-2}=X_{n-2t-1}\ne\cdots\ne X_{n-1})=2^{-2t-1}$. By exercise 7, $\Pr(X_n$ is not the beginning of a coupon set$)=\sum_{t\ge0}q_t=\frac{2}{3}$; $\Pr(X_n$ is the beginning of coupon set of length $s+2)=\sum_{t\ge0}p_{st}=\frac{1}{3}\cdot2^{-s-1}$. Now proceed as in exercise 13.

16. (Solution by R. P. Stanley.) Whenever the subsequence $S = (b-1), (b-2), \ldots, 1, 0, 0, 1, \ldots, (b-2), (b-1)$ appears, a coupon set must end at the right of S, since some coupon set is completed in the first half of S. We now proceed to calculate the probability that a coupon set begins at position n by manipulating the probabilities that the last prior appearance of S ends at position $n-1$, $n-2$, etc., as in exercise 15.

18. Proceed as in the proof of Theorem A to calculate $\underline{\Pr}$ and $\overline{\Pr}$.

19. (Solution by T. Herzog.) Yes; e.g., the sequence $\langle U_{\lfloor n/2 \rfloor} \rangle$ when $\langle U_n \rangle$ satisfies R4 (or even its weaker version), cf. exercise 33.

21. $\Pr(Z_n \in M_1, \ldots, Z_{n+k-1} \in M_k) = p(M_1) \ldots p(M_k)$, for all $M_1, \ldots, M_k \in \mathcal{M}$.

22. If the sequence is k-distributed, the limit is zero by integration and Theorem B. Conversely, note that if $f(x_1, \ldots, x_k)$ has an absolutely convergent Fourier series

$$f(x_1, \ldots, x_k) = \sum_{-\infty < c_1, \ldots, c_k < \infty} a(c_1, \ldots, c_k) \exp(2\pi i(c_1 x_1 + \cdots + c_k x_k)),$$

we have $\lim_{N \to \infty} \frac{1}{N} \sum_{0 \le n < N} f(U_n, \ldots, U_{n+k-1}) = a(0, \ldots, 0) + \epsilon_r$, where

$$|\epsilon_r| \le \sum_{|c_1|, \ldots, |c_k| > r} |a(c_1, \ldots, c_k)|,$$

so ϵ_r can be made arbitrarily small. Hence this limit is equal to

$$a(0, \ldots, 0) = \int_0^1 \cdots \int_0^1 f(x_1, \ldots, x_k)\, dx_1 \ldots dx_k,$$

and Eq. (8) holds for all sufficiently smooth functions f. The remainder of the proof shows that the function in (9) can be approximated by smooth functions to any desired accuracy.

23. See *AMM* **75** (1968), 260–264.

24. This follows immediately from exercise 22.

25. If the sequence is equidistributed, the denominator in Corollary S approaches $\frac{1}{12}$, and the numerator approaches the quantity in this exercise.

26. See *Math. Comp.* **17** (1963), 50–54. [Consider also the following example by A. G. Waterman: Let $\langle U_n \rangle$ be an equidistributed $[0, 1)$ sequence and $\langle X_n \rangle$ an ∞-distributed binary sequence. Let $V_n = U_{\lceil \sqrt{n} \rceil}$ or $1 - U_{\lceil \sqrt{n} \rceil}$ according as X_n is 0 or 1. Then $\langle V_n \rangle$ is equidistributed and white, but $\Pr(V_n = V_{n+1}) = \frac{1}{2}$. Let $W_n = (V_n - \epsilon_n) \bmod 1$ where $\langle \epsilon_n \rangle$ is any sequence that decreases monotonically to 0; then $\langle W_n \rangle$ is equidistributed and white, yet $\Pr(W_n < W_{n+1}) = \frac{3}{4}$.]

28. Let $\langle U_n \rangle$ be ∞-distributed, and consider the sequence $\langle \frac{1}{2}(X_n + U_n) \rangle$. This is 3-distributed, using the fact that $\langle U_n \rangle$ is $(16, 3)$-distributed.

29. If $x = x_1x_2\ldots x_t$ is any binary number, we can consider the number $\nu_x^E(n)$ of times $X_p\ldots X_{p+t-1} = x$, where $1 \le p \le n$ and p is even. Similarly, let $\nu_x^O(n)$ count the number of times when p is odd. Let $\nu_x^E(n) + \nu_x^O(n) = \nu_x(n)$. Now

$$\nu_0^E(n) = \sum \nu^E_{0**\ldots*}(n) \approx \sum \nu^O_{*0*\ldots*}(n) \approx \sum \nu^E_{**0\ldots*}(n) \approx \cdots \approx \sum \nu^O_{***\ldots0}(n)$$

where the ν's in these summations have $2k$ subscripts, $2k-1$ of which are asterisks (meaning that they are being summed over—each sum is taken over 2^{2k-1} combinations of zeros and ones), and where "$\approx$" denotes approximate equality (except for an error of at most $2k$ due to end conditions). Therefore we find that

$$\frac{1}{n}2k\nu_0^E(n) = \frac{1}{n}\Big(\sum \nu_{*0*\ldots*}(n) + \cdots + \sum \nu_{***\ldots0}(n)\Big) + \frac{1}{n}\sum_x (r(x) - s(x))\nu_x^E(n) + O\Big(\frac{1}{n}\Big),$$

where $x = x_1\ldots x_{2k}$ contains $r(x)$ zeros in odd positions and $s(x)$ zeros in even positions. By $(2k)$-distribution, the parenthesized quantity tends to $k(2^{2k-1})/2^{2k} = k/2$. The remaining sum is clearly a maximum if $\nu_x^E(n) = \nu_x(n)$ when $r(x) > s(x)$, and $\nu_x^E(n) = 0$ when $r(x) < s(x)$. So the maximum of the right-hand side becomes

$$\frac{k}{2} + \sum_{0\le s<r\le k} (r-s)\binom{k}{r}\binom{k}{s}\Big/2^{2k} = \frac{k}{2} + k\binom{2k-1}{k}\Big/2^{2k}.$$

Now $\overline{\Pr}(X_{2n} = 0) \le \limsup_{n\to\infty} \nu_0^E(2n)/n$, so the proof is complete. *Note:*

$$\sum_{r,s}\binom{n}{r}\binom{n}{s}\max(r,s) = 2n2^{2n-2} + n\binom{2n-1}{n};$$
$$\sum_{r,s}\binom{n}{r}\binom{n}{s}\min(r,s) = 2n2^{2n-2} - n\binom{2n-1}{n}.$$

30. Let $f(x_1, x_2, \ldots, x_{2k}) = \operatorname{sign}(x_1 - x_2 + x_3 - x_4 + \cdots - x_{2k})$. Construct a directed graph with 2^{2k} nodes labeled $(E; x_1, \ldots, x_{2k-1})$ and $(O; x_1, \ldots, x_{2k-1})$, where each x is either 0 or 1. Let there be $1 + f(x_1, x_2, \ldots, x_{2k})$ directed arcs from $(E; x_1, \ldots, x_{2k-1})$ to $(O; x_2, \ldots, x_{2k})$, and $1 - f(x_1, x_2, \ldots, x_{2k})$ directed arcs leading from $(O; x_1, \ldots, x_{2k-1})$ to $(E; x_2, \ldots, x_{2k})$. We find that each node has the same number of arcs leading into it as there are leading out; for example, $(E; x_1, \ldots, x_{2k-1})$ has $1 - f(0, x_1, \ldots, x_{2k-1}) + 1 - f(1, x_1, \ldots, x_{2k-1})$ leading in and $1 + f(x_1, \ldots, x_{2k-1}, 0) + 1 + f(x_1, \ldots, x_{2k-1}, 1)$ leading out, and $f(x, x_1, \ldots, x_{2k-1}) = -f(x_1, \ldots, x_{2k-1}, x)$. Drop all nodes that have no paths leading either in or out, i.e., $(E; x_1, \ldots, x_{2k-1})$ if $f(0, x_1, \ldots, x_{2k-1}) = +1$, or $(O; x_1, \ldots, x_{2k-1})$ if $f(1, x_1, \ldots, x_{2k-1}) = -1$. The resulting directed graph is seen to be connected, since we can get from any node to $(E; 1, 0, 1, 0, \ldots, 1)$ and from this to any desired node. By Theorem 2.3.4.2G, there is a cyclic path traversing each arc; this path has length 2^{2k+1}, and we may assume that it starts at node $(E; 0, \ldots, 0)$. Construct a cyclic sequence with $X_1 = \cdots = X_{2k-1} = 0$, and $X_{n+2k-1} = x_{2k}$ if the nth arc of the path is from $(E; x_1, \ldots, x_{2k-1})$ to $(O; x_2, \ldots, x_{2k})$ or from $(O; x_1, \ldots, x_{2k-1})$ to $(E; x_2, \ldots, x_{2k})$. For example, the graph for $k = 2$ is shown in Fig. A–5; the arcs of

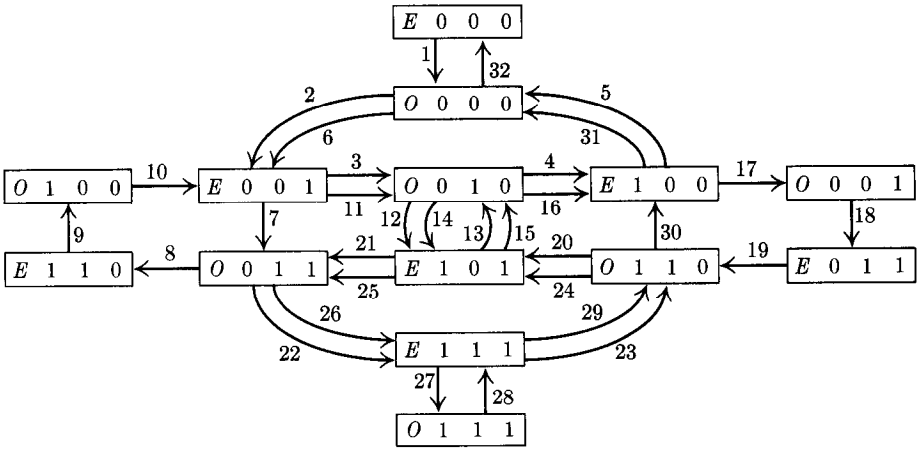


Fig. A–5. Directed graph for the construction in exercise 30.

the cyclic path are numbered from 1 to 32, and the cyclic sequence is

$$(00001000110010101001101110111110)(00001\ldots).$$

Note that $\Pr(X_{2n} = 0) = \frac{11}{16}$ in this sequence. The sequence is clearly $(2k)$-distributed, since each $(2k)$-tuple $x_1x_2\ldots x_{2k}$ occurs $1 + f(x_1, \ldots, x_{2k}) + 1 - f(x_1, \ldots, x_{2k}) = 2$ times in the cycle. The fact that $\Pr(X_{2n} = 0)$ has the desired value comes from the fact that the maximum value on the right-hand side in the proof of the preceding exercise has been achieved by this construction.

31. Use Algorithm W with rule $\mathcal{R}_1$ selecting the entire sequence. [For a generalization of this type of nonrandom behavior in R5-sequences, see Jean Ville, *Étude Critique de la notion de Collectif* (Paris, 1939), 55–62. Perhaps R6 is also too weak, from this standpoint.]

32. If $\mathcal{R}, \mathcal{R}'$ are computable subsequence rules, so is $\mathcal{R}'' = \mathcal{R}\mathcal{R}'$ defined by the following functions: $f''_n(x_0, \ldots, x_{n-1}) = 1$ iff $\mathcal{R}$ defines the subsequence $x_{r_1}, \ldots, x_{r_k}$ of $x_0, \ldots, x_{n-1}$, where $k \geq 0$ and $0 \leq r_1 < \cdots < r_k < n$ and $f'_k(x_{r_1}, \ldots, x_{r_k}) = 1$.

Now $\langle X_n\rangle\mathcal{R}\mathcal{R}'$ is $(\langle X_n\rangle\mathcal{R})\mathcal{R}'$. The result follows immediately.

33. Given $\epsilon > 0$, find N_0 such that $N > N_0$ implies that both $|\nu_r(N)/N - p| < \epsilon$ and $|\nu_s(N)/N - p| < \epsilon$. Then find N_1 such that $N > N_1$ implies that t_N is r_M or s_M for some $M > N_0$. Now $N > N_1$ implies that

$$\left|\frac{\nu_t(N)}{N} - p\right| = \left|\frac{\nu_r(N_r) + \nu_s(N_s)}{N} - p\right| = \left|\frac{\nu_r(N_r) - pN_r + \nu_s(N_s) - pN_s}{N_r + N_s}\right| < \epsilon.$$

34. For example, if the binary representation of t is $(1\,0^{b-2}\,1\,0^{a_1}\,1\,1\,0^{a_2}\,1\ldots1\,0^{a_k})_2$, where "$0^a$" stands for a sequence of a consecutive zeros, let the rule $\mathcal{R}_t$ accept U_n if and only if $\lfloor bU_{n-k}\rfloor = a_1, \ldots, \lfloor bU_{n-1}\rfloor = a_k$.

35. Let $a_0 = s_0$ and $a_{m+1} = \max\{\, s_k \mid 0 \leq k < 2^{a_m} \,\}$. Construct a subsequence rule that selects element X_n if and only if $n = s_k$ for some $k < 2^{a_m}$, when n is in the range $a_m \leq n < a_{m+1}$. Then $\lim_{m\to\infty} \nu(a_m)/a_m = \frac{1}{2}$.

36. Let b and k be arbitrary but fixed integers greater than 1. Let $Y_n = \lfloor bU_n \rfloor$. An arbitrary infinite subsequence $\langle Z_n \rangle = \langle Y_{s_n} \rangle \mathcal{R}$ determined by algorithms $\mathcal{S}$ and $\mathcal{R}$ (as in the proof of Theorem M) corresponds in a straightforward but notationally hopeless manner to algorithms $\mathcal{S}'$ and $\mathcal{R}'$ that inspect X_t, X_{t+1}, ..., X_{t+s} and/or select X_t, X_{t+1}, ..., $X_{t+\min(k-1,s)}$ of $\langle X_n \rangle$ if and only if $\mathcal{S}$ and $\mathcal{R}$ inspect and/or select Y_s, where $U_s = (0.X_tX_{t+1}\ldots X_{t+s})_2$. Algorithms $\mathcal{S}'$ and $\mathcal{R}'$ determine an infinite 1-distributed subsequence of $\langle X_n \rangle$ and in fact (as in exercise 32) this subsequence is ∞-distributed so it is $(k, 1)$-distributed. Hence we find that $\underline{\Pr}(Z_n = a)$ and $\overline{\Pr}(Z_n = a)$ differ from $1/b$ by less than $1/2^k$.

[The result of this exercise is true if "R6" is replaced consistently by "R4" or "R5"; but it is false if "R1" is used, since $X_{\binom{n}{2}}$ might be identically zero.]

37. For $n \geq 2$ replace U_{n^2} by $\frac{1}{2}(U_{n^2} + \delta_n)$, where $\delta_n = 0$ or 1 according as the set $\{U_{(n-1)^2+1}, \ldots, U_{n^2-1}\}$ contains an even or odd number of elements less than $\frac{1}{2}$. [*Advances in Math.* **14** (1974), 333–334.]

39. See *Acta Arithmetica* **21** (1972), 45–50. The best possible value of c is unknown.

40. If every one-digit change to a random table yields a random table, all tables are random (or none are). If we don't allow degrees of randomness, the answer must therefore be, "Not always."

SECTION 3.6

```
1. RANDI STJ  9F          Store exit location.
          STA  8F         Store value of k.
          LDA  XRAND      rA ← X.
          MUL  7F         rAX ← aX.
          INCX 1009       rX ← (aX + c) mod m.
          JOV  *+1        Ensure that overflow is off.
          SLAX 5          rA ← (aX + c) mod m.
          STA  XRAND      Store X.
          MUL  8F         rA ← ⌊kX/m⌋.
          INCA 1          Add 1, so that 1 ≤ Y ≤ k.
   9H     JMP  *          Return.
   XRAND  CON  1          Value of X; X_0 = 1.
   8H     CON  0          Temp storage of k.
   7H     CON  3141592621 The multiplier a.  ∎
```

2. Putting a random number generator into a program makes the results essentially unpredictable to the programmer. If the behavior of the machine on each problem were known in advance, few programs would ever be written. As Turing has said, the actions of a computer quite often *do* surprise its programmer, especially when a program is being debugged.

So the world had better watch out.

7. In fact, you only need the 2-bit values $\lfloor X_n/2^{16} \rfloor \bmod 4$; see D. E. Knuth, "Deciphering a linear congruential encryption," to appear. See also J. Reeds, *Cryptologia* **1** (1977), 20–26, **3** (1979), 83–95, for solutions to related problems.

SECTION 4.1

1. 1010, 1011, 1000, ..., 11000, 11001, 11110.

2. (a) -110001, $-11.001001001001\ldots$, $11.0010010000111111\ldots$.
(b) 11010011, 1101.001011001011... , 111.011001000100000....
(c) $\overline{1}11\overline{1}1$, $\overline{1}0.0\overline{1}10110\overline{1}1011\ldots$, $10.011\overline{1}111\overline{1}\ldots$.
(d) -9.4, $-\ldots 7582417582413$, $\ldots 562951413$.

3. $(1010113.2)_{2i}$.

4. (a) Between rA and rX. (b) The remainder in rX has radix point between bytes 3 and 4; the quotient in rA has radix point one byte to the right of the least significant portion of the register.

5. It has been subtracted from $999\ldots9 = 10^p - 1$, instead of from $1000\ldots0 = 10^p$.

6. (a,c) $2^{p-1} - 1$, $-(2^{p-1} - 1)$; (b) $2^{p-1} - 1$, -2^{p-1}.

7. A ten's complement representation for a negative number x can be obtained by considering $10^n + x$ (where n is large enough for this to be positive) and extending it on the left with infinitely many nines. The nines' complement representation can be obtained in the usual manner. (These two representations are equal for nonterminating decimals, otherwise the nines' complement representation has the form $\ldots(a)99999\ldots$ while the ten's complement representation has the form $\ldots(a+1)0000\ldots$.) The representations may be considered sensible if we regard the value of the infinite sum $N = 9 + 90 + 900 + 9000 + \cdots$ as -1, since $N - 10N = 9$.

See also exercise 31, which considers p-adic number systems. The latter agree with the p's complement notations considered here, for numbers whose radix-p representation is terminating, but there is no simple relation between the field of p-adic numbers and the field of real numbers.

8. $\sum_j a_j b^j = \sum_j (a_{kj+k-1} b^{k-1} + \cdots + a_{kj}) b^{kj}$.

9. A BAD ADOBE FACADE FADED. [*Note:* Other possible "number sentences" would be DO A DEED A DECADE; A CAD FED A BABE BEEF, COCOA, COFFEE; BOB FACED A DEAD DODO.]

10. $\begin{bmatrix} \ldots, a_3, a_2, a_1, a_0; a_{-1}, a_{-2}, \ldots \\ \ldots, b_3, b_2, b_1, b_0; b_{-1}, b_{-2}, \ldots \end{bmatrix} = \begin{bmatrix} \ldots, A_3, A_2, A_1, A_0; A_{-1}, A_{-2}, \ldots \\ \ldots, B_3, B_2, B_1, B_0; B_{-1}, B_{-2}, \ldots \end{bmatrix}$, if

$$A_j = \begin{bmatrix} a_{k_j+1}-1, a_{k_j+1}-2, \ldots, a_{k_j} \\ b_{k_j+1}-2, \ldots, b_{k_j} \end{bmatrix}, \qquad B_j = b_{k_j+1}-1 \ldots b_{k_j},$$

where $\langle k_n \rangle$ is any infinite sequence of integers with $k_{j+1} > k_j$.

11. (The following algorithm works both for addition or subtraction, depending on whether the plus or minus sign is chosen.)

Start by setting $k \leftarrow a_{n+1} \leftarrow a_{n+2} \leftarrow b_{n+1} \leftarrow b_{n+2} \leftarrow 0$; then for $m = 0, 1, \ldots, n+2$ do the following: Set $c_m \leftarrow a_m \pm b_m + k$; then if $c_m \geq 2$, set $k \leftarrow -1$ and $c_m \leftarrow c_m - 2$; otherwise if $c_m < 0$, set $k \leftarrow 1$ and $c_m \leftarrow c_m + 2$; otherwise (i.e., if $0 \leq c_m \leq 1$), set $k \leftarrow 0$.

12. (a) Subtract $\pm(\ldots a_3 0 a_1 0)_{-2}$ from $\pm(\ldots a_4 0 a_2 0 a_0)_{-2}$ in the negabinary system. (See also exercise 7.1–18 for a trickier solution that uses full-word logical operations.) (b) Subtract $(\ldots b_3 0 b_1 0)_2$ from $(\ldots b_4 0 b_2 0 b_0)_2$ in the binary system.

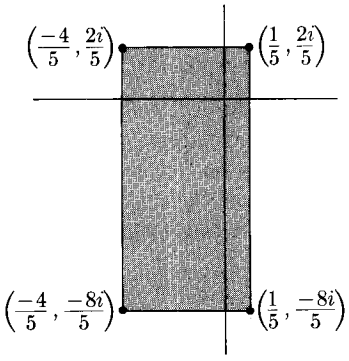


Fig. A–6. Fundamental region for quater-imaginary numbers.

13. $(1.909090\ldots)_{-10} = (0.090909\ldots)_{-10} = \frac{1}{11}$.

14.

$$\begin{array}{rrrrrrrrrrl}
 & & & & 1 & 1 & 3 & 2 & 1 & & [5-4i] \\
 & & & & 1 & 1 & 3 & 2 & 1 & & [5-4i] \\
\hline
 & & & & 1 & 1 & 3 & 2 & 1 & & \\
 & & & 1 & 1 & 2 & 0 & 2 & & & \\
 & & 1 & 2 & 1 & 2 & 3 & & & & \\
 & 1 & 1 & 3 & 2 & 1 & & & & & \\
1 & 1 & 3 & 2 & 1 & & & & & & \\
\hline
0 & 1 & 0 & 3 & 1 & 1 & 2 & 0 & 1 & & [9-40i]
\end{array}$$

15. $[-\frac{10}{11}, \frac{1}{11}]$, and the rectangle shown in Fig. A–6.

16. It is tempting to try to do this in a very simple way, by using the rule $2 = (1100)_{i-1}$ to take care of carries; but that leads to a nonterminating method if, for example, we try to add 1 to $(11101)_{i-1} = -1$.

The following solution does the job by providing four related algorithms (namely for adding or subtracting 1 or i). If α is a string of zeros and ones, let α^P be a string of zeros and ones such that $(\alpha^P)_{i-1} = (\alpha)_{i-1} + 1$; and let α^{-P}, α^Q, α^{-Q} be defined similarly, with -1, $+i$, and $-i$ respectively in place of $+1$. Then

$$\begin{array}{llll}
(\alpha 0)^P = \alpha 1; & (\alpha x 1)^P = \alpha^Q x 0. & (\alpha 0)^Q = \alpha^P 1; & (\alpha 1)^Q = \alpha^{-Q} 0. \\
(\alpha x 0)^{-P} = \alpha^{-Q} x 1; & (\alpha 1)^{-P} = \alpha 0. & (\alpha 0)^{-Q} = \alpha^Q 1; & (\alpha 1)^{-Q} = \alpha^{-P} 0.
\end{array}$$

Here x stands for either 0 or 1, and the strings are extended on the left with zeros if necessary. The processes will clearly always terminate. Hence every number of the form $a + bi$ with a and b integers is representable in the $i - 1$ system.

17. No (in spite of exercise 28); the number -1 cannot be so represented. This can be proved by constructing a set S as in Fig. 1. We do have the representations $-i = (0.1111\ldots)_{1+i}$, $i = (100.1111\ldots)_{1+i}$.

18. Let S_0 be the set of points $(a_7a_6a_5a_4a_3a_2a_1a_0)_{i-1}$, where each a_k is 0 or 1. (Thus, S_0 is given by the 256 interior dots shown in Fig. 1, if that picture is multiplied by 16.) We first show that S is closed: If y_1, y_2, ... is an infinite subset of S, we have $y_n = \sum_{k\geq 1} a_{nk}16^{-k}$, where each a_{nk} is in S_0. Construct a tree whose nodes are $(a_{n1}, \ldots, a_{nr})$, for $1 \leq r \leq n$, and let a node of this tree be an ancestor of another

node if it is an initial subsequence of that node. By the infinity lemma this tree has an infinite path $(a_1, a_2, a_3, \dots)$, and it follows that $\sum_{k\geq 1} a_k 16^{-k}$ is a limit point of $\{y_1, y_2, \dots\}$ in S.

By the answer to exercise 16, all numbers of the form $(a+bi)/16^k$ are representable, when a and b are integers. Therefore if x and y are arbitrary reals and $k \geq 1$, the number $z_k = (\lfloor 16^k x \rfloor + \lfloor 16^k y \rfloor i)/16^k$ is in $S + m + ni$ for some integers m and n. It can be shown that $S + m + ni$ is bounded away from the origin when $(m, n) \neq (0, 0)$. Consequently if $|x|$ and $|y|$ are fixed and k is sufficiently large, we have $z_k \in S$, and $\lim_{k\to\infty} z_k = x + yi$ is in S.

[B. Mandelbrot calls S the "twindragon," since he noticed that it is essentially obtained by joining two "dragon curves" belly-to-belly; see his book *Fractals: Form, Chance, and Dimension* (San Francisco: Freeman, 1977), 313–314. Other properties of the dragon curve are described in C. Davis and D. E. Knuth, *J. Recr. Math.* **3** (1970), 66–81, 133–149.]

19. If $m > u$ or $m < l$, find $a \in D$ such that $m \equiv a$ (modulo b); the desired representation will be a representation of $m' = (m - a)/b$ followed by a. Note that $m > u$ implies $l < m' < m$; $m < l$ implies $m < m' < u$; so the algorithm terminates.

[There are no solutions when $b = 2$. The representation will be unique iff $0 \in D$; nonunique representation occurs for example when $D = \{-3, -1, 7\}$, $b = 3$, since $(\alpha)_3 = (\overline{3}77\overline{3}\alpha)_3$. When $b \geq 3$ it is not difficult to show that there are exactly 2^{b-3} solution sets D in which $|a| < b$ for all $a \in D$. Furthermore the set $D = \{0, 1, 2 - \epsilon_2 b^n, 3 - \epsilon_3 b^n, \dots, b - 2 - \epsilon_{b-2} b^n, b - 1 - b^n\}$ gives unique representations, for all $b \geq 3$ and $n \geq 1$, when each ϵ_j is 0 or 1. Reference: *Proc. IEEE Symp. Comp. Arith.* **4** (1978), 1–9.]

20. (a) $0.\overline{111}\dots = \overline{1}.888\dots = \overline{1}8.^{111}_{777}\dots = \overline{1}8^{1\ 222}_{7\cdot 666}\dots = \cdots = \overline{1}8^{123456\ 777}_{765432\cdot 111}\dots$ has nine representations. (b) A "D-fraction" $.a_1a_2\dots$ always lies between $-1/9$ and $+71/9$. Suppose x has ten or more D-decimal representations. Then for sufficiently large k, $10^k x$ has ten representations that differ to the left of the decimal point: $10^k x = n_1 + f_1 = \cdots = n_{10} + f_{10}$ where each f_j is a D-fraction. By uniqueness of integer representations, the n_j are distinct, say $n_1 < \cdots < n_{10}$, hence $n_{10} - n_1 \geq 9$; but this implies $f_1 - f_{10} \geq 9 > 71/9 - (-1/9)$, a contradiction. (c) Any number of the form $0.a_1a_2\dots$, where each a_j is -1 or 8, equals $\overline{1}.a'_1a'_2\dots$ where $a'_j = a_j + 9$ (and it even has six *more* representations $\overline{1}8.a''_1a''_2\dots$, etc.).

21. We can convert to such a representation by using a method like that suggested in the test for converting to balanced ternary.

In contrast to the systems of exercise 20, zero can be represented in infinitely many ways, all obtained from $\frac{1}{2} + \sum_{k\geq 1}(-4\frac{1}{2}) \cdot 10^{-k}$ (or from the negative of this representation) by multiplying it by a power of ten. The representations of unity are $1\frac{1}{2} - \frac{1}{2}$, $\frac{1}{2} + \frac{1}{2}$, $5 - 3\frac{1}{2} - \frac{1}{2}$, $5 - 4\frac{1}{2} + \frac{1}{2}$, $50 - 45 - 3\frac{1}{2} - \frac{1}{2}$, $50 - 45 - 4\frac{1}{2} + \frac{1}{2}$, etc., where $\pm\frac{1}{2} = (\pm 4\frac{1}{2})(10^{-1} + 10^{-2} + \cdots)$. [*AMM* **57** (1950), 90–93.]

22. Given some approximation $b_n \dots b_1 b_0$ with error $\sum_{0\leq k\leq n} b_k 10^k - x > 10^{-t}$ for $t > 0$, we will show how to reduce the error by approximately 10^{-t}. (The process can be started by finding a suitable $\sum_{0\leq k\leq n} b_k 10^k > x$; then a finite number of reductions of this type will make the error less than ϵ.) Simply choose $m > n$ so large that the decimal representation of $-10^m\alpha$ has a one in position 10^{-t} and no ones in positions $10^{-t+1}, 10^{-t+2}, \dots, 10^n$. Then $10^m\alpha +$ (a suitable sum of powers of 10 between 10^m and 10^n) $+ \sum_{0\leq k\leq n} b_k 10^k \approx \sum_{0\leq k\leq n} b_k 10^k - 10^{-t}$.

23. The set $S = \{\sum_{k\geq 1} a_k b^{-k} \mid a_k \in D\}$ is closed as in exercise 18, hence it is measurable, and in fact it has positive measure. Since $bS = \bigcup_{a\in D}(a+S)$, we have $b\mu(S) = \mu(bS) \leq \sum_{a\in D}\mu(a+S) = \sum_{a\in D}\mu(S) = b\mu(S)$, and we must therefore have $\mu((a+S)\cap(a'+S)) = 0$ when $a \neq a' \in D$. Now T has measure zero since it is a union of countably many sets of the form $10^k(n+((a+S)\cap(a'+S)))$, $a \neq a'$, each of measure zero.

[The set T cannot be empty, since the real numbers cannot be written as a countable union of disjoint, closed, bounded sets; cf. *AMM* **84** (1977), 827–828. If D has less than b elements, the set of numbers representable with radix b and digits from D has measure zero. If D has more than b elements and represents all reals, T has infinite measure.]

24. $\{2a\cdot 10^k + a' \mid 0 \leq a < 5, 0 \leq a' < 2\}$ or $\{5a'\cdot 10^k + a \mid 0 \leq a < 5, 0 \leq a' < 2\}$, for $k \geq 0$. [R. L. Graham has shown that there are no more sets of integer digits with these properties. And Andrew Odlyzko has shown that the restriction to integers is superflous, in the sense that if the smallest two elements of D are 0 and 1, all the digits must be integers. *Proof:* Let $S = \{\sum_{k<0} a_k b^k \mid a_k \in D\}$ be the set of "fractions," and let $X = \{(a_n \ldots a_0)_b \mid a_k \in D\}$ be the set of "whole numbers"; then $[0,\infty) = \bigcup_{x\in X}(x+S)$, and $(x+S)\cap(x'+S)$ has measure zero for $x \neq x' \in X$. We have $(0,1) \subseteq S$, and by induction on m we will prove that $(m, m+1) \subseteq x_m + S$ for some $x_m \in X$. Let $x_m \in X$ be such that $(m, m+\epsilon)\cap(x_m+S)$ has positive measure for all $\epsilon > 0$. Then $x_m \leq m$, and x_m must be an integer lest $x_{\lfloor x_m\rfloor} + S$ overlap $x_m + S$ too much. If $x_m > 0$, the fact that $(m-x_m, m-x_m+1)\cap S$ has positive measure implies by induction that this measure is 1, and $(m, m+1) \subseteq x_m + S$ since S is closed. If $x_m = 0$ and $(m, m+1) \not\subseteq S$, we must have $m < x'_m < m+1$ for some $x'_m \in X$, where $(m, x'_m) \subseteq S$; but then $1+S$ overlaps $x'_m + S$. See *Proc. London Math. Soc.* (3) **18** (1978), 581–595.]

Note: If we drop the restriction $0 \in D$, there *are* many other cases, some of which are quite interesting, especially $\{1,2,3,4,5,6,7,8,9,10\}$, $\{1,2,3,4,5,51,52,53,54,55\}$, and $\{2,3,4,5,6,52,53,54,55,56\}$. Alternatively if we allow negative digits we obtain many other solutions by the method of exercise 19, plus further sets of unusual digits like $\{-1,0,1,2,3,4,5,6,7,18\}$ that don't meet the conditions stated there. It appears hopeless to find a nice characterization of all solutions with negative digits.]

25. A positive number whose base b representation has m consecutive $(b-1)$'s to the right of the decimal point must have the form $c/b^n + (b^m - \theta)/b^{n+m}$, where c and n are nonnegative integers and $0 < \theta \leq 1$. So if u/v has this form, we find that $b^{m+n}u = b^m cv + b^m v - \theta v$. Therefore θv is an integer that is a multiple of b^m. But $0 < \theta v \leq v < b^m$. [There can be arbitrarily long runs of other digits *aaaaa*, if $0 \leq a < b-1$, for example in the representation of $a/(b-1)$.]

26. The proof of "sufficiency" is a straightforward generalization of the usual proof for base b, by successively constructing the desired representation. The proof of "necessity" breaks into two parts: If β_{n+1} is greater than $\sum_{k\leq n} c_k\beta_k$ for some n, then $\beta_{n+1} - \epsilon$ has no representation for small ϵ. If $\beta_{n+1} \leq \sum_{k\leq n} c_k\beta_k$ for all n, but equality does not always hold, we can show that there are two representations for certain x. [See *Transactions of the Royal Society of Canada*, series III, **46** (1952), 45–55.]

27. Proof by induction on $|n|$: If n is even we must take $e_0 > 0$, and the result follows by induction, since $n/2$ has a unique such representation. If n is odd, we must take $e_0 = 0$, and the problem reduces to representing $-(n-1)/2$; if the latter quantity is

either zero or one, there is obviously only one way to proceed, otherwise it has a unique reversing representation by induction.

[It follows that every positive integer has exactly *two* such representations with *decreasing* exponents $e_0 > e_1 > \cdots > e_t$: one with t even and the other with t odd.]

28. A proof like that of exercise 27 may be given. Note that $a + bi$ is a multiple of $1+i$ by a complex integer if and only if $a+b$ is even. This representation is intimately related to the dragon curve discussed in the answer to exercise 18.

29. It suffices to prove that any collection $\{T_0, T_1, T_2, \dots\}$ satisfying Property B may be obtained by collapsing some collection $\{S_0, S_1, S_2, \dots\}$, where $S_0 = \{0, 1, \dots, b-1\}$ and all elements of S_1, S_2, ... are multiples of b.

To prove the latter statement, we may assume that $1 \in T_0$ and that there is a least element $b > 1$ such that $b \notin T_0$. We will prove, by induction on n, that if $nb \notin T_0$, then $nb+1$, $nb+2$, ..., $nb+b-1$ are not in any of the T_j's; but if $nb \in T_0$, then so are $nb+1$, ..., $nb+b-1$. The result then follows with $S_1 = \{\, nb \mid nb \in T_0 \,\}$, $S_2 = T_1$, $S_3 = T_2$, etc.

If $nb \notin T_0$, then $nb = t_0 + t_1 + \cdots$, where t_1, t_2, ... are multiples of b; hence $t_0 < nb$ is a multiple of b. By induction, $(t_0 + k) + t_1 + t_2 + \cdots$ is the representation of $nb+k$, for $0 < k < b$; hence $nb + k \notin T_j$ for any j.

If $nb \in T_0$ and $0 < k < b$, let the representation of $nb+k$ be $t_0 + t_1 + \cdots$. We cannot have $t_j = nb+k$ for $j \geq 1$, lest $nb+b$ have two representations $(b-k) + \cdots + (nb+k) + \cdots = (nb) + \cdots + b + \cdots$. By induction, $t_0 \bmod b = k$; and the representation $nb = (t_0 - k) + t_1 + \cdots$ implies that $t_0 = nb + k$.

[Reference: *Nieuw Archief voor Wiskunde* (3) **4** (1956), 15–17. A finite analog of this result was derived by P. A. MacMahon, *Combinatory Analysis* **1** (1915), 217–223.]

30. (a) Let A_j be the set of numbers n whose representation does not involve b_j; then by the uniqueness property, $n \in A_j$ iff $n + b_j \notin A_j$. Consequently we have $n \in A_j$ iff $n + 2b_j \in A_j$. It follows that, for $j \neq k$, $n \in A_j \cap A_k$ iff $n + 2b_jb_k \in A_j \cap A_k$. Let m be the number of integers $n \in A_j \cap A_k$ such that $0 \leq n < 2b_jb_k$. Then this interval contains exactly m integers that are in A_j but not A_k, exactly m in A_k but not A_j, and exactly m in neither A_j nor A_k; hence $4m = 2b_jb_k$. Therefore b_j and b_k cannot both be odd. But at least one b_j is odd, of course, since odd numbers can be represented.
(b) According to (a) we can renumber the b's so that b_0 is odd and b_1, b_2, ... are even; then $\frac{1}{2}b_1$, $\frac{1}{2}b_2$, ... must also be a binary basis, and the process can be iterated.

(c) If it is a binary basis, we must have positive and negative d_k's for arbitrarily large k, in order to represent $\pm 2^n$ when n is large. Conversely, the following algorithm may be used:

S1. [Initialize.] Set $k \leftarrow 0$.

S2. [Done?] If $n = 0$, terminate.

S3. [Choose.] If n is odd, include $2^k d_k$ in the representation, and set $n \leftarrow (n - d_k)/2$. Otherwise set $n \leftarrow n/2$.

S4. [Advance k.] Increase k by 1 and return to S2. ∎

At each step the choice is forced; furthermore step S3 always decreases $|n|$ unless $n = -d_k$, hence the algorithm must terminate.

(d) Two iterations of steps S2–S4 in the preceding algorithm will change $4m \to m$, $4m+1 \to m+5$, $4m+2 \to m+7$, $4m+3 \to m-1$. Arguing as in exercise 19, we need only show that the algorithm terminates for $-2 \leq n \leq 8$; all other values of n

are moved toward this interval. In this range $3 \to -1 \to -2 \to 6 \to 8 \to 2 \to 7 \to 0$ and $4 \to 1 \to 5 \to 6$. Thus $1 = 7\cdot 2^0 - 13\cdot 2^1 + 7\cdot 2^2 - 13\cdot 2^3 - 13\cdot 2^5 - 13\cdot 2^9 + 7\cdot 2^{10}$.

Note: The choice $d_0, d_1, d_2, \ldots = 5, -3, 3, 5, -3, 3, \ldots$ also yields a binary basis. For further details see *Math. Comp.* **18** (1964), 537–546; A. D. Sands, *Acta Mathematica*, Acad. Sci. Hung., **8** (1957), 65–86.

31. (See also the related exercises 3.2.2–11, 4.3.2–13, 4.6.2–22.)

(a) By multiplying numerator and denominator by suitable powers of 2, we may assume that $u = (\ldots u_2u_1u_0)_2$ and $v = (\ldots v_2v_1v_0)_2$ are 2-adic integers, where $v_0 = 1$. The following computational method now determines w, using the notation $u^{(n)}$ to stand for the integer $(u_{n-1}\ldots u_0)_2 = u \bmod 2^n$ when $n > 0$:

Let $w_0 = u_0$ and $w^{(1)} = w_0$. For $n = 1, 2, \ldots$, assume that we have found an integer $w^{(n)} = (w_{n-1}\ldots w_0)_2$ such that $u^{(n)} \equiv v^{(n)}w^{(n)}$ (modulo 2^n). Then we have $u^{(n+1)} \equiv v^{(n+1)}w^{(n)}$ (modulo 2^n), hence $w_n = 0$ or 1 according as the quantity $(u^{(n+1)} - v^{(n+1)}w^{(n)}) \bmod 2^{n+1}$ is 0 or 2^n.

(b) Find the smallest integer k such that $2^k \equiv 1$ (modulo $2n+1$). Then we have $1/(2n+1) = m/(2^k - 1)$ for some integer m, $1 \le m < 2^{k-1}$. Let α be the k-bit binary representation of m; then $(0.\alpha\alpha\alpha\ldots)_2$ times $2n+1$ is $(0.111\ldots)_2 = 1$ in the binary system, and $(\ldots\alpha\alpha\alpha)_2$ times $2n+1$ is $(\ldots 111)_2 = -1$ in the 2-adic system.

(c) If u is rational, say $u = m/(2^en)$ where n is odd and positive, the 2-adic representation of u is periodic, because the set of numbers with periodic expansions includes $-1/n$ and is closed under the operations of negation, division by 2, and addition. Conversely, if $u_{N+\lambda} = u_N$ for all $N \ge \mu$, the 2-adic number $(2^\lambda - 1)2^{-\mu}u$ is an integer.

(d) The square of any number of the form $(\ldots u_2u_11)_2$ has the form $(\ldots 001)_2$, hence the condition is necessary. To show the sufficiency, we can use the following procedure to compute $v = \sqrt{n}$ when $n \bmod 8 = 1$:

H1. [Initialize.] Set $m \leftarrow (n-1)/8$, $k \leftarrow 2$, $v_0 \leftarrow 1$, $v_1 \leftarrow 0$, $v \leftarrow 1$. (During this algorithm we will have $v = (v_{k-1}\ldots v_1v_0)_2$ and $v^2 = n - 2^{k+1}m$.)

H2. [Transform.] If m is even, set $v_k \leftarrow 0$, $m \leftarrow m/2$. Otherwise set $v_k \leftarrow 1$, $m \leftarrow (m - v - 2^{k-1})/2$, $v \leftarrow v + 2^k$.

H3. [Advance k.] Increase k by 1 and return to H2. ∎

32. A generalization appears in *Math. Comp.* **29** (1975), 84–86.

33. Let K_n be the set of all such n-digit numbers, so that $k_n = ||K_n||$. If S and T are any finite sets of integers, we shall say $S \sim T$ if $S = T + x$ for some integer x, and we shall write $k_n(S) = ||K_n(S)||$, where $K_n(S)$ is the family of all subsets of K_n that are $\sim S$. When $n = 0$, we have $k_n(S) = 0$ unless $||S|| \le 1$, since zero is the only "0-digit" number. When $n \ge 1$ and $S = \{s_1, \ldots, s_r\}$, we have

$$K_n(S) = \bigcup_{0\le j<b} \bigcup_{(a_1,\ldots,a_r)} \Big\{ \{t_1b + a_1, \ldots, t_rb + a_r\} \;\Big|\; \{t_1, \ldots, t_r\} \in K_{n-1}(\{(S_i + j - a_i)/b \mid 1 \le i \le r\}) \Big\},$$

where the inner union is over all sequences of digits $(a_1, \ldots, a_r)$ satisfying the condition $a_i \equiv S_i + j$ (modulo b) for $1 \le i \le r$. In this formula we require $t_i - t_{i'} = (s_i - a_i)/b - (s_{i'} - a_{i'})/b$ for $1 \le i < i' \le r$, so that the naming of subscripts is uniquely determined. By the principle of inclusion and exclusion, therefore, we have $k_n(S) = \sum_{0\le j<b}\sum_{m\ge 1}(-1)^{m-1}f(S, m, j)$, where $f(S, m, j)$ is the number of sets

of integers that can be expressed as $\{t_1 b + a_1, \ldots, t_r b + a_r\}$ in the above manner for m different sequences $(a_1, \ldots, a_r)$, summed over all choices of m different sequences $(a_1, \ldots, a_r)$. Given m different sequences $(a_1^{(l)}, \ldots, a_r^{(l)})$ for $1 \le l \le m$, the number of such sets is $k_{n-1}(\{(s_i + j - a_i^{(l)})/b \mid 1 \le i \le r, 1 \le l \le m\})$. Thus there is a collection of sets $\mathcal{T}(S)$ such that

$$k_n(S) = \sum_{T \in \mathcal{T}(S)} c_T \, k_{n-1}(T),$$

where each c_T is an integer. Furthermore if $T \in \mathcal{T}(S)$, its elements are near those of S; we have $\min T \ge (\min S - \max D)/b$ and $\max T \le (\max S + b - 1 - \min D)/b$. Thus we obtain simultaneous recurrence relations for the sequences $\langle k_n(S) \rangle$, where S runs through the nonempty integer subsets of $[l, u+1]$, in the notation of exercise 19. Since $k_n = k_n(S)$ for any one-element set S, the sequence $\langle k_n \rangle$ appears these recurrences. The coefficients c_T can be computed from the first few values of $k_n(S)$, so we can obtain a system of equations defining the generating functions $k_S(z) = \sum k_n(S) z^n = (\|S\| \le 1) + z \sum_{T \in \mathcal{T}(S)} k_T(z)$.

For example, when $D = \{-1, 0, 3\}$ and $b = 3$ we have $l = -\frac{3}{2}$ and $u = \frac{1}{2}$, so the relevant sets S are $\{0\}$, $\{0, 1\}$, $\{-1, 1\}$, and $\{-1, 0, 1\}$. The corresponding sequences for $n \le 3$ are $\langle 1, 3, 8, 21 \rangle$, $\langle 0, 1, 3, 8 \rangle$, $\langle 0, 0, 1, 4 \rangle$, and $\langle 0, 0, 0, 0 \rangle$; so we obtain

$$k_0(z) = 1 + z(3k_0(z) - k_{01}(z)), \qquad k_{02}(z) = z(k_{01}(z) + k_{02}(z)),$$
$$k_{01}(z) = zk_0(z), \qquad l_{012}(z) = 0,$$

and $k(z) = 1/(1 - 3z + z^2)$. In this case $k_n = F_{2n+2}$ and $k_n(\{0, 2\}) = F_{2n-1} - 1$.

SECTION 4.2.1

1. $N = (62, +.60\ 22\ 52\ 00)$; $\hbar = (37, +.10\ 54\ 50\ 00)$. Note that $10\hbar$ would be $(38, +.01\ 05\ 45\ 00)$.

2. $b^{E-q}(1 - b^{-p})$, b^{-q-p}; $b^{E-q}(1 - b^{-p})$, b^{-q-1}.

3. When e does not have its smallest value, the most significant "one" bit (which appears in all such normalized numbers) need not appear in the computer word.

4. $(51, +.10209877)$; $(50, +.12346000)$; $(53, +.99999999)$. The third answer would be $(54, +.10000000)$ if the first operand had been $(45, -.50000000)$.

5. If $x \sim y$ and m is an integer then $mb + x \sim mb + y$. Furthermore $x \sim y$ implies $x/b \sim y/b$, by considering all possible cases. Another crucial property is that x and y will round to the same integer, whenever $x \sim y$.

Now if $b^{-p-2} F_v \ne f_v$ we must have $(b^{p+2} f_v) \bmod b \ne 0$; hence the transformation leaves f_v unchanged unless $e_u - e_v \ge 2$. Since u was normalized, it is nonzero and $|f_u + f_v| > b^{-1} - b^{-2} \ge b^{-2}$: the leading nonzero digit of $f_u + f_v$ must be at most two places to the right of the radix point, and the rounding operation will convert $b^{p+j}(f_u + f_v)$ to an integer, where $j \le 1$. The proof will be complete if we can show that $b^{p+j+1}(f_u + f_v) \sim b^{p+j+1}(f_u + b^{-p-2} F_v)$. By the previous paragraph, we have $b^{p+2}(f_u + f_v) \sim b^{p+2} f_u + F_v = b^{p+2}(f_u + b^{-p-2} F_v)$, which implies the desired result for all $j \le 1$.

Note that, when $b > 2$ is even, such an integer F_v always exists; but when $b = 2$ we require $p + 3$ bits (let $2F_v$ be an integer). When b is odd, an integer F_v always exists except in the case of division, when a remainder of $\frac{1}{2}b$ is possible.

6. (Consider the case $e_u = e_v$, $f_u = -f_v$ in Program A.) Register A retains its previous sign, as in ADD.

7. Say that a number is normalized iff it is zero or its fraction part lies in the range $\frac{1}{6} < |f| < \frac{1}{2}$. A $(p+1)$-place accumulator suffices for addition and subtraction; rounding (except during division) is equivalent to truncation. A very pleasant system indeed! We might represent numbers with excess-zero exponent, inserted between the first and subsequent digits of the fraction, and complemented if the fraction is negative, so that fixed-point order is preserved.

8. (a) $(06, +.12345679) \oplus (06, -.12345678)$, $(01, +.10345678) \oplus (00, -.94000000)$; (b) $(99, +.87654321) \oplus$ itself, $(99, +.99999999) \oplus (91, +.50000000)$.

9. $a = c = (-50, +.10000000)$, $b = (-41, +.20000000)$, $d = (-41, +.80000000)$, $y = (11, +.10000000)$.

10. $(50, +.99999000) \oplus (55, +.99999000)$.

11. $(50, +.10000001) \otimes (50, +.99999990)$.

12. If $0 < |f_u| < |f_v|$, then $|f_u| \le |f_v| - b^{-p}$; hence $1/b < |f_u/f_v| \le 1 - b^{-p}/|f_v| < 1 - b^{-p}$. If $0 < |f_v| \le |f_u|$, we have $1/b < |f_u/f_v|/b \le ((1-b^{-p})/(1/b))/b = 1 - b^{-p}$.

13. See J. Michael Yohe, *IEEE Transactions* **C–22** (1973), 577–586; cf. also exercise 4.2.2–24.

14.
```
FIX  STJ   9F          Float-to-fix subroutine:
     STA   TEMP
     LD1   TEMP(EXP)   rI1 ← e.
     SLA   1           rA ← ± f f f f 0.
     JAZ   9F          Is input zero?
     DEC1  1
     CMPA  =0=(1:1)    If leading byte is zero,
     JE    *-4           shift left again.
     ENN1  -Q-4,1
     J1N   FIXOVFLO    Is magnitude too large?
     ENTX  0
     SRAX  0,1
     CMPX  =1//2=
     JL    9F
     JG    *+2
     JAO   9F
     STA   *+1(0:0)    Round, if necessary.
     INCA  1           Add ±1 (overflow is impossible).
9H   JMP   *           Exit from subroutine. ∎
```

15.
```
FP   STJ   EXITF        Fractional part subroutine:
     JOV   OFLO         Ensure overflow is off.
     STA   TEMP         TEMP ← u.
     ENTX  0
     SLA   1            rA ← f_u.
```

```
      LD2  TEMP(EXP)        rI2 ← e_u.
      DEC2 Q
      J2NP *+3
      SLA  0,2              Remove integer part of u.
      ENT2 0
      JANN 1F
      ENN2 0,2              Fraction is negative: find
      SRAX 0,2                its complement.
      ENT2 0
      JAZ  *+2
      INCA 1
      ADD  WM1              Add word size minus one.
1H    INC2 Q                Prepare to normalize the answer.
      JMP  NORM             Normalize, round, and exit.
8H    EQU  1(1:1)
WM1   CON  8B-1,8B-1(1:4)   Word size minus one  ∎
```

16. If $|c| \geq |d|$, then set $r \leftarrow d \oslash c$, $s \leftarrow c \oplus (r \otimes d)$; $x \leftarrow (a \oplus (b \otimes r)) \oslash s$, $y \leftarrow (b \ominus (a \otimes r)) \oslash s$. Otherwise set $r \leftarrow c \oslash d$, $s \leftarrow d \oplus (r \otimes c)$; $x \leftarrow ((a \otimes r) \oplus b) \oslash s$, $y \leftarrow ((b \otimes r) \ominus a) \oslash s$. Then $x+iy$ is the desired approximation to $(a+bi)/(c+di)$. [*CACM* **5** (1963), 435. Other algorithms for complex arithmetic and function evaluation are given by P. Wynn, *BIT* **2** (1962), 232–255; see also Paul Friedland, *CACM* **10** (1967), 665.]

17. See Robert Morris, *IEEE Transactions* **C–20** (1971), 1578–1579. Error analysis is more difficult with such systems, so interval arithmetic is correspondingly more desirable.

18. For positive numbers: shift fraction left until $f_1 = 1$, then round, then if the fraction is zero (rounding overflow) shift it right again. For negative numbers: shift fraction left until $f_1 = 0$, then round, then if the fraction is zero (rounding underflow) shift it right again.

19. $(43+(1 \text{ if } e_v < e_u)-(1 \text{ if fraction overflow})-(10 \text{ if result zero})+(4 \text{ if magnitude is rounded up})+(1 \text{ if first rounding digit is } b/2)+(5 \text{ if rounding digits are } b/20\ldots0)+(7 \text{ if rounding overflow}) + 7N + A(-1 + (11 \text{ if } N > 0)))u$, where N is the number of left shifts during normalization and $A = 1$ if rX receives nonzero digits (otherwise $A = 0$). The maximum time of $73u$ occurs for example when

$$u = +50\ 01\ 00\ 00\ 00, \quad v = -46\ 49\ 99\ 99\ 99, \quad b = 100.$$

[The average time, considering the data in Section 4.2.4, will be about $45\frac{1}{2}u$.]

SECTION 4.2.2

1. $u \ominus v = u \oplus -v = -v \oplus u = -(v \oplus -u) = -(v \ominus u)$.

2. $u \oplus x \geq u \oplus 0 = u$, by (8), (2), (6); hence by (8) again, $(u \oplus x) \oplus v \geq u \oplus v$. Similarly, (8) and (6) together with (2) imply that $(u \oplus x) \oplus (v \oplus y) \geq (u \oplus x) \oplus v$.

3. $u = 8.0000001$, $v = 1.2500008$, $w = 8.0000008$; $(u \otimes v) \otimes w = 80.000064$, $u \otimes (v \otimes w) = 80.000057$.

4. Yes; let $1/u \approx v = w$, where v is large.

5. Not always; in decimal arithmetic take $u = v = 9$.

6. (a) Yes. (b) Only for $b+p \le 4$ (try $u = 1-b^{-p}$). [W. M. Kahan observes that the identity *does* hold whenever $b^{-1} \le f_u \le b^{-1/2}$. It follows that $1 \oslash (1 \oslash (1 \oslash u)) = 1 \oslash u$ for all u.]

7. If u and v are consecutive floating binary numbers, $u \oplus v = 2u$ or $2v$. When it is $2v$ we often have $u^{(2)} \oplus v^{(2)} < 2v^{(2)}$. For example, $u = (.10\ldots001)_2$, $v = (.10\ldots010)_2$, $u \oplus v = 2v$, and $u^{(2)} + v^{(2)} = (.10\ldots011)_2$.

8. (a) $\sim$, $\approx$; (b) $\sim$, $\approx$; (c) $\sim$, $\approx$; (d) $\sim$; (e) $\sim$.

9. $|u-w| \le |u-v| + |v-w| \le \epsilon_1 \min(b^{e_u-q}, b^{e_v-q}) + \epsilon_2 \min(b^{e_v-q}, b^{e_w-q}) \le \epsilon_1 b^{e_u-q} + \epsilon_2 b^{e_w-q} \le (\epsilon_1+\epsilon_2)\max(b^{e_u-q}, b^{e_w-q})$. The result cannot be strengthened in general, since for example we might have e_u very small compared to both e_v and e_w, and this means that $u - w$ might be fairly large under the hypotheses.

10. We have $(.a_1\ldots a_{p-1}a_p)_b \otimes (.9\ldots99)_b = (.a_1\ldots a_{p-1}(a_p-1))_b$ if $a_p \ge 1$; here "9" stands for $b-1$. Furthermore, $(.a_1\ldots a_{p-1}a_p)_b \otimes (1.0\ldots0)_b = (.a_1\ldots a_{p-1}0)_b$, so the multiplication is not monotone if $b > 2$ and $a_p \ge 2$. But when $b = 2$, this argument can be extended to show that multiplication *is* monotone; obviously the "certain computer" had $b > 2$.

11. Without loss of generality, let x be an integer, $|x| < b^p$. If $e \le 0$, then $t = 0$. If $0 < e \le p$, then $x - t$ has at most $p+1$ digits, the least significant being zero. If $e > p$, then $x - t = 0$. [The result holds also under the weaker hypothesis $|t| < 2b^e$.]

12. Assume that $e_u = p$, $e_v \le 0$, $u > 0$. Case 1, $u > b^{p-1}$. Case (1a), $w = u+1$, $v \ge \frac{1}{2}$, $e_v = 0$. Then $u' = u$ or $u+1$, $v' = 1$, $u'' = u$, $v'' = 1$ or 0. Case (1b), $w = u$, $|v| \le \frac{1}{2}$. Then $u' = u$, $v' = 0$, $u'' = u$, $v'' = 0$. If $|v| = \frac{1}{2}$ and more general rounding is permitted we might also have $u' = u \pm 1$, $v'' = \mp 1$. Case (1c), $w = u-1$, $v \le -\frac{1}{2}$, $e_v = 0$. Then $u' = u$ or $u-1$, $v' = -1$, $u'' = u$, $v'' = -1$ or 0. Case 2, $u = b^{p-1}$. Case (2a), $w = u+1$, $v \ge \frac{1}{2}$, $e_v = 0$. Like (1a). Case (2b), $w = u$, $|v| \le \frac{1}{2}$, $u' \ge u$. Like (1b). Case (2c), $w = u$, $|v| \le \frac{1}{2}$, $u' < u$. Then $u' = u - j/b$ where $v = j/b + v_1$ and $|v_1| \le \frac{1}{2}b^{-1}$ for some positive integer $j \le \frac{1}{2}b$; we have $v' = 0$, $u'' = u$, $v'' = j/b$. Case (2d), $w < u$. Then $w = u - j/b$ where $v = -j/b + v_1$ and $|v_1| \le \frac{1}{2}b^{-1}$ for some positive integer $j \le b$; we have $(v', u'') = (-j/b, u)$, and $(u', v'') = (u, -j/b)$ or $(u - 1/b, (1-j)/b)$, the latter case only when $v_1 = \frac{1}{2}b^{-1}$. In all cases $u \ominus u' = u - u'$, $v \ominus v' = v - v'$, $u \ominus u'' = u - u''$, $v \ominus v'' = v - v''$, $\text{round}(w - u - v) = w - u - v$.

13. Since $\text{round}(x) = 0$ iff $x = 0$, we want to find a large set of integer pairs (m, n) with the property that $m \oslash n$ is an integer iff m/n is. Assume that $|m|, |n| < b^p$. If m/n is an integer, then $m \oslash n = m/n$ is also. Conversely if m/n is not an integer, but $m \oslash n$ is, we have $1/|n| \le |m \oslash n - m/n| < \frac{1}{2}|m/n|b^{1-p}$, hence $|m| > 2b^{p-1}$. Our answer is therefore to require $|m| \le 2b^{p-1}$ and $0 < |n| < b^p$. (Slightly weaker hypotheses are also possible.)

14. $|(u \otimes v) \otimes w - uvw| \le |(u \otimes v) \otimes w - (u \otimes v)w| + |w|\,|u \otimes v - uv| \le \delta_{(u\otimes v)\otimes w} + b^{e_w-q-l_w}\delta_{u\otimes v} \le (1+b)\delta_{(u\otimes v)\otimes w}$. Now $|e_{(u\otimes v)\otimes w} - e_{u\otimes(v\otimes w)}| \le 2$, so we may take $\epsilon = \frac{1}{2}(1+b)b^{2-p}$.

15. $u \le v$ implies that $(u \oplus u) \oslash 2 \le (u \oplus v) \oslash 2 \le (v \oplus v) \oslash 2$, so the condition holds for all u and v iff it holds whenever $u = v$. For base $b = 2$, the condition is therefore always satisfied (barring overflow); but for $b > 2$ there are numbers $v \neq w$ such that $v \oplus v = w \oplus w$, hence the condition fails. [On the other hand, the formula $u \oplus ((v \ominus u) \oslash 2)$ does give a midpoint in the correct range. *Proof:* It suffices to show that $u + (v \ominus u) \oslash 2 \le v$, i.e., $(v \ominus u) \oslash 2 \le v - u$; and it is easy to verify that $\text{round}(\frac{1}{2}\text{round}(x)) \le x$ for all $x \ge 0$.]

16. (a) Exponent changes occur at $\sum_{10} = 11.111111$, $\sum_{91} = 101.11111$, $\sum_{901} = 1001.1102$, $\sum_{9001} = 10001.020$, $\sum_{90009} = 100000.91$, $\sum_{900819} = 1000000.0$; therefore $\sum_{1000000} = 1109099.1$.

(b) $\sum_{1 \le k \le n} 1.2345679 = 1224782.1$, and (14) tries to take the square root of $-.0053187053$. But (15) and (16) are exact in this case. [If $x_k = 1 + \lfloor (k-1)/2 \rfloor 10^{-7}$, (15) and (16) have errors of order n. See Chan and Lewis, *CACM* **22** (1979), 526–531, for further results on the accuracy of standard deviation calculations.]

(c) We need to show that $u \oplus ((v \ominus u) \oslash k)$ lies between u and v; see exercise 15.

17.
```
FCMP  STJ   9F          Floating point comparison subroutine:
      JOV   OFLO        Ensure overflow is off.
      STA   TEMP
      LDAN  TEMP        v ← −v.
```
(Copy here lines 07–20 of Program 4.2.1A.)
```
      LDX   FV(0:0)     Set rX to zero with sign of f_v.
      DEC1  5
      J1N   *+2
      ENT1  0           Replace large difference in exponents
      SRAX  5,1               by a smaller one.
      ADD   FU          rA ← difference of operands.
      JOV   7F          Fraction overflow: not ∼.
      CMPA  EPSILON(1:5)
      JG    8F          Jump if not ∼.
      JL    6F          Jump if ∼.
      JXZ   9F          Jump if ∼.
      JXP   1F          If |rA| = ε, check sign of rA × rX.
      JAP   9F          Jump if ∼. (rA ≠ 0)
      JMP   8F
7H    ENTX  1
      SRC   1           Make rA nonzero with same sign.
      JMP   8F
1H    JAP   8F          Jump if not ∼. (rA ≠ 0)
6H    ENTA  0
8H    CMPA  =0=         Set comparison indicator.
9H    JMP   *           Exit from subroutine. ∎
```

19. Let $\gamma_k = \delta_k = \eta_k = \sigma_k = 0$ for $k > n$. It suffices to find the coefficient of x_1, since the coefficient of x_k will be just the same except with all subscripts increased by $k-1$. Let (f_k, g_k) denote the coefficient of x_1 in $(s_k - c_k, c_k)$ respectively. Then $f_1 = (1+\eta_1)(1-\gamma_1-\gamma_1\delta_1-\gamma_1\sigma_1-\delta_1\sigma_1-\gamma_1\delta_1\sigma_1)$, $g_1 = (1+\delta_1)(1+\eta_1)(\gamma_1+\sigma_1+\gamma_1\sigma_1)$, and $f_k = (1-\gamma_k\sigma_k-\delta_k\sigma_k-\gamma_k\delta_k\sigma_k)f_{k-1} + (\gamma_k-\eta_k+\gamma_k\delta_k+\gamma_k\eta_k+\gamma_k\delta_k\eta_k+\gamma_k\eta_k\sigma_k+\delta_k\eta_k\sigma_k+\gamma_k\delta_k\eta_k\sigma_k)g_{k-1}$, $g_k = \sigma_k(1+\gamma_k)(1+\delta_k)f_{k-1} - (1+\delta_k)(\gamma_k+$

$\gamma_k\eta_k + \eta_k\sigma_k + \gamma_k\eta_k\sigma_k)g_{k-1}$, for $1 < k \le n$. Thus $f_n = 1 + \eta_1 - \gamma_1 + (4n$ terms of 2nd order) + (higher order terms) $= 1 + \eta_1 - \gamma_1 + O(n\epsilon^2)$ is sufficiently small. [The Kahan summation formula was first published in *CACM* **8** (1965), 40; cf. *Proc. IFIP Congress* (1971), **2**, 1232. For another approach to accurate summation, see R. J. Hanson, *CACM* **18** (1975), 57–58. See also G. Bohlender, *IEEE Trans.* **C–26** (1977), 621–632, for algorithms that compute round$(x_1+\cdots+x_n)$ and round$(x_1 \ldots x_n)$ *exactly*, given $\{x_1, \ldots, x_n\}$.]

20. By the proof of Theorem C, (47) fails for $e_w = p$ only if $|v| + \frac{1}{2} \ge |w - u| \ge b^{p-1} + b^{-1}$; hence $|f_u| \ge |f_v| \ge 1 - (\frac{1}{2}b - 1)b^{-p}$. This rather rare case, in which $|f_w|$ before normalization takes its maximum value 2, is necessary and sufficient for failure.

21. (Solution by G. W. Veltkamp.) Let $c = 2^{\lceil p/2 \rceil} + 1$; we may assume that $p \ge 2$, so c is representable. First compute $u' = u \otimes c$, $u_1 = (u \ominus u') \oplus u'$, $u_2 = u \ominus u_1$; similarly, $v' = v \otimes c$, $v_1 = (v \ominus v') \oplus v'$, $v_2 = v \ominus v_1$. Then set $w \leftarrow u \otimes v$, $w' \leftarrow (((u_1 \otimes v_1 \ominus w) \oplus (u_1 \otimes v_2)) \oplus (u_2 \otimes v_1)) \oplus (u_2 \otimes v_2)$.

It suffices to prove this when $u, v > 0$ and $e_u = e_u = p$, so that u and v are integers $\in [2^{p-1}, 2^p)$. Then $u = u_1 + u_2$ where $2^{p-1} \le u_1 \le 2^p$, $u_1 \bmod 2^{\lceil p/2 \rceil} = 0$, and $|u_2| \le 2^{\lceil p/2 \rceil - 1}$; similarly $v = v_1 + v_2$. The operations during the calculation of w' are exact, because $w - u_1v_1$ is a multiple of 2^{p-1} such that $|w - u_1v_1| \le |w - uv| + |u_2v_1 + u_1v_2 + u_2v_2| \le 2^{p-1} + 2^{p+\lceil p/2 \rceil} + 2^{p-1}$; and similarly $|w - u_1v_1 - u_1v_2| \le |w - uv| + |u_2v| < 2^{p-1} + 2^{\lceil p/2 \rceil - 1 + p}$, where $w - u_1v_1 - u_1v_2$ is a multiple of $2^{\lceil p/2 \rceil}$.

22. We may assume that $b^{p-1} \le u, v < b^p$. If $uv \le b^{2p-1}$, then $x_1 = uv - r$ where $|r| \le \frac{1}{2}b^{p-1}$, hence $x_2 = \text{round}(u - r/v) = x_0$ (since $|r/v| \le \frac{1}{2}b^{p-1}/b^{p-1} \le \frac{1}{2}$, and equality implies $v = b^{p-1}$ hence $r = 0$). If $uv > b^{2p-1}$, then $x_1 = uv - r$ where $|r| \le \frac{1}{2}b^p$, hence $x_1/v \le u - r/v < b^p + \frac{1}{2}b$ and $x_2 \le b^p$. If $x_2 = b^p$, then $x_3 = x_1$ (for otherwise $(b^p - \frac{1}{2})v \le x_1 \le b^p(v - 1)$). If $x_2 < b^p$ and $x_1 > b^{2p-1}$, then let $x_2 = x_1/v + q$ where $|q| \le \frac{1}{2}$; we have $x_3 = \text{round}(x_1 + qv) = x_1$. Finally if $x_2 < b^p$ and $x_1 = b^{2p-1}$ and $x_3 < b^{2p-1}$, then $x_4 = x_2$ by the first case above. This situation arises, for example, when $b = 10$, $p = 2$, $u = 19$, $v = 55$, $x_1 = 1000$, $x_2 = 18$, $x_3 = 990$.

23. Let $\lfloor u \rfloor = n$, so that $u \bmod 1 = u \ominus n = u - n + r$ where $|r| \le \frac{1}{2}b^{-p}$; we wish to show that round$(n - r) = n$. The result is clear if $|n| > 1$; and $r = 0$ when $n = 0$ or 1, so the only subtle case is when $n = -1$, $r = -\frac{1}{2}b^{-p}$. The identity fails iff b is a multiple of 4 and $-b^{-1} < u < -b^{-2}$ and $u \bmod 2b^{-p} = \frac{3}{2}b^{-p}$ (e.g., $p = 3$, $b = 8$, $u = -(.0124)_8$).

24. Let $u = [u_l, u_r]$, $v = [v_l, v_r]$. Then $u \oplus v = [u_l \mathbin{\triangledown} v_l, u_r \mathbin{\triangle} v_r]$, where $x \mathbin{\triangle} y = y \mathbin{\triangle} x$, $x \mathbin{\triangle} +0 = x$ for all x, $x \mathbin{\triangle} -0 = x$ for all $x \ne +0$, $x \mathbin{\triangle} +\infty = +\infty$ for all $x \ne -\infty$, and $x \mathbin{\triangle} -\infty$ needn't be defined; $x \mathbin{\triangledown} y = -((-x) \mathbin{\triangle} (-y))$. If $x \oplus y$ would overflow in normal floating point arithmetic because $x + y$ is too large, then $x \mathbin{\triangle} y$ is $+\infty$ and $x \mathbin{\triangledown} y$ is the largest representable number.

For subtraction, let $u \ominus v = u \oplus (-v)$, where $-v = [-v_r, -v_l]$.

Multiplication is somewhat more complicated. The correct procedure is to let

$$u \otimes v = [\min(u_l \mathbin{\triangledown} v_l, u_l \mathbin{\triangledown} v_r, u_r \mathbin{\triangledown} v_l, u_r \mathbin{\triangledown} v_r),\ \max(u_l \mathbin{\triangle} v_l, u_l \mathbin{\triangle} v_r, u_r \mathbin{\triangle} v_l, u_r \mathbin{\triangle} v_r)],$$

where $x \mathbin{\triangle} y = y \mathbin{\triangle} x$, $x \mathbin{\triangle} (-y) = -(x \mathbin{\triangledown} y) = (-x) \mathbin{\triangle} y$; $x \mathbin{\triangle} +0 = (+0$ for $x > 0$, -0 for $x < 0)$; $x \mathbin{\triangle} -0 = -(x \mathbin{\triangle} +0)$; $x \mathbin{\triangle} +\infty = (+\infty$ for $x > +0$, $-\infty$ for $x < -0)$. (It is possible to determine the min and max simply by looking at the signs of u_l, u_r, v_l, and v_r, thereby computing only two of the eight products, except when

$u_l < 0 < u_r$ and $v_l < 0 < v_r$; in the latter case we compute four products, and the answer is $[\min(u_l \mathbin{\triangledown} v_r, u_r \mathbin{\triangledown} v_l), \max(u_l \mathbin{\triangle} v_l, u_r \mathbin{\triangle} v_r)]$.)

Finally, $u \oslash v$ is undefined if $v_l < 0 < v_r$; otherwise we use the formulas for multiplication with v_l and v_r replaced respectively by v_r^{-1} and v_l^{-1}, where $x \mathbin{\triangle} y^{-1} = x \mathbin{\triangle} y$, $x \mathbin{\triangledown} y^{-1} = x \mathbin{\triangledown} y$, $(\pm 0)^{-1} = \pm\infty$, $(\pm\infty)^{-1} = \pm 0$.

[Cf. E. R. Hansen, *Math. Comp.* **22** (1968), 374–384. An alternative scheme, in which division by 0 gives no error messages and intervals may be neighborhoods of ∞, has been proposed by W. M. Kahan. In Kahan's scheme, for example, the reciprocal of $[-1, +1]$ is $[+1, -1]$, and an attempt to multiply an interval containing 0 by an interval containing ∞ yields $[-\infty, +\infty]$, the set of all numbers. See *Numerical Analysis*, Univ. Michigan Engineering Summer Conf. Notes No. 6818 (1968).]

25. Cancellation reveals *previous* errors in the computation of u and v. For example, if ϵ is small, we often get poor accuracy when computing $f(x + \epsilon) \ominus f(x)$, because the rounded calculation of $f(x + \epsilon)$ destroys much of the information about ϵ. It is desirable to rewrite such formulas as $\epsilon \otimes g(x, \epsilon)$, where $g(x, \epsilon) = (f(x + \epsilon) - f(x))/\epsilon$ is first computed symbolically. Thus, if $f(x) = x^2$ then $g(x, \epsilon) = 2x + \epsilon$; if $f(x) = \sqrt{x}$ then $g(x, \epsilon) = 1/(\sqrt{x + \epsilon} + \sqrt{x})$.

26. See *Math. Comp.* **32** (1978), 227–232.

SECTION 4.2.3

1. First, $(w_m, w_l) = (.573, .248)$; then $w_m v_l / v_m = .290$; so the answer is $(.572, .958)$. This in fact is the correct result to six decimals.

2. The answer is not affected, since the normalization routine truncates to eight places and can never look at this particular byte position. (Scaling to the left occurs at most once during normalization, since the inputs are normalized.)

3. Overflow obviously cannot occur at line 09, since we are adding two-byte quantities, or at line 22, since we are adding four-byte quantities. In line 30 we are computing the sum of three four-byte quantities, so this cannot overflow. Finally, in line 32, overflow is impossible because the product $f_u f_v$ must be less than unity.

4. Insert "`JOV OFLO; ENT1 0`" between lines 03 and 04. Replace lines 21–22 by "`ADD TEMP(ABS); JNOV *+2; INC1 1`", and change lines 28–31 to "`SLAX 5; ADD TEMP; JNOV *+2; INC1 1; ENTX 0,1; SRC 5`". This adds five lines of code and only 1, 2, or 3 units of execution time.

5. Insert "`JOV OFLO`" after line 06. Change lines 22, 31, 39 respectively to "`SRAX 0,1`", "`SLAX 5`", "`ADD ACC`". Between lines 40 and 41, insert "`DEC2 1; JNOV DNORM; INC2 1; INCX 1; SRC 1`". (It's tempting to remove the "`DEC2 1`" in favor of "`STZ EXPO`", but then "`INC2 1`" might overflow rI2!) This adds six lines of code; the running time *decreases* by $3u$, unless there is fraction overflow, when it increases by $7u$.

6.
```
DOUBLE STJ  EXITDF      Convert to double precision:
       ENTX 0           Clear rX.
       STA  TEMP
       LD2  TEMP(EXP)   rI2 ← e.
       INC2 QQ-Q        Correct for difference in excess.
```

```
        STZ   EXPO          EXPO ← 0.
        SLAX  1             Remove exponent.
        JMP   DNORM         Normalize and exit.
SINGLE  STJ   EXITF         Convert to single precision:
        JOV   OFLO          Ensure overflow is off.
        STA   TEMP
        LD2   TEMP(EXPD)    rI2 ← e.
        DEC2  QQ-Q          Correct for difference in excess.
        SLAX  2             Remove exponent.
        JMP   NORM          Normalize, round, and exit. ∎
```

7. All three routines give zero as the answer if and only if the exact result would be zero, so we need not worry about zero denominators in the expressions for relative error. The worst case of the addition routine is pretty bad: Visualized in decimal notation, if the inputs are 1.0000000 and .99999999, the answer is b^{-7} instead of b^{-8}; thus the maximum relative error δ_1 is $b-1$, where b is the byte size.

For multiplication and division, we may assume that both operands are positive and have the same exponent QQ. The maximum error in multiplication is readily bounded by considering Fig. 4: When $uv \geq 1/b$, we have $0 \leq uv - u \otimes v < 3b^{-9} + (b-1)b^{-9}$, so the relative error is bounded by $(b+2)b^{-8}$. When $1/b^2 \leq uv < 1/b$, we have $0 \leq uv - u \otimes v < 3b^{-9}$, so the relative error in this case is bounded by $3b^{-9}/uv \leq 3b^{-7}$. We take δ_2 to be the larger of the two estimates, namely $3b^{-7}$.

Division requires a more careful analysis of Program D. The quantity actually computed by the subroutine is $\alpha - \delta - b\epsilon((\alpha-\delta'')(\beta-\delta') - \delta''') - \delta_n$ where $\alpha = (u_m + \epsilon u_l)/bv_m$, $\beta = v_l/bv_m$, and the nonnegative truncation errors $(\delta, \delta', \delta'', \delta''')$ are respectively less than $(b^{-10}, b^{-5}, b^{-5}, b^{-6})$; finally δ_n (the truncation during normalization) is nonnegative and less than either b^{-9} or b^{-8}, depending on whether scaling occurs or not. The actual value of the quotient is $\alpha/(1+b\epsilon\beta) = \alpha - b\epsilon\alpha\beta + b^2\alpha\beta^2\delta''''$, where δ'''' is the nonnegative error due to truncation of the infinite series (2); here $\delta'''' < \epsilon^2 = b^{-10}$, since it is an alternating series. The relative error is therefore the absolute value of $(b\epsilon\delta' + b\epsilon\delta''\beta/\alpha + b\epsilon\delta'''/\alpha) - (\delta/\alpha + b\epsilon\delta'\delta''/\alpha + b^2\beta^2\delta'''' + \delta_n/\alpha)$, times $(1+b\epsilon\beta)$. The positive terms in this expression are bounded by $b^{-9}+b^{-8}+b^{-8}$, and the negative terms are bounded by $b^{-8}+b^{-12}+b^{-8}$ plus the contribution by the normalizing phase, which can be about b^{-7} in magnitude. It is therefore clear that the potentially greatest part of the relative error comes during the normalization phase, and that $\delta_3 = (b+2)b^{-8}$ is a safe upper bound for the relative error.

8. Addition: If $e_u \leq e_v + 1$, the entire relative error occurs during the normalization phase, so it is bounded above by b^{-7}. If $e_u \geq e_v + 2$, and if the signs are the same, again the entire error may be ascribed to normalization; if the signs are opposite, the error due to shifting digits out of the register is in the opposite direction from the subsequent error introduced during normalization. Both of these errors are bounded by b^{-7}, hence $\delta_1 = b^{-7}$. (This is substantially better then the result in exercise 7.)

Multiplication: An analysis as in exercise 7 gives $\delta_2 = (b+2)b^{-8}$.

SECTION 4.2.4

1. Since fraction overflow can occur only when the operands have the same sign, this is the probability that fraction overflow occurs divided by the probability that the operands have the same sign, namely, $7\%/(\frac{1}{2}(91\%)) \approx 15\%$.

3. $\log_{10} 2.4 - \log_{10} 2.3 \approx 1.84834\%$.

4. The pages would be uniformly gray (same as "random point on a slide rule").

5. The probability that $10f_U \le r$ is $(r-1)/10 + (r-1)/100 + \cdots = (r-1)/9$. So in this case the leading digits are *uniformly* distributed; e.g., leading digit 1 occurs with probability $\frac{1}{9}$.

6. The probability that there are three leading zero bits is $\log_{16} 2 = \frac{1}{4}$; the probability that there are two leading zero bits is $\log_{16} 4 - \log_{16} 2 = \frac{1}{4}$; and similarly for the other two cases. The "average" number of leading zero bits is $1\frac{1}{2}$, so the "average" number of "significant bits" is $p + \frac{1}{2}$. The worst case, $p-1$ bits, occurs however with rather high probability. In practice, it is usually necessary to base error estimates on the worst case, since a chain of calculations is only as strong as its weakest link. In the error analysis of Section 4.2.2, the upper bound on relative rounding error for floating hex is 2^{1-p}. In the binary case we can have $p+1$ significant bits at all times (cf. exercise 4.2.1–3), with relative rounding errors bounded by 2^{-1-p}. Extensive computational experience confirms that floating binary (even with p-bit precision instead of $p+1$) produces significantly more accurate results than $(p+2)$-bit floating hex.

Tables 1 and 2 show that hexadecimal arithmetic can be done a little faster, since fewer cycles are needed when scaling to the right or normalizing to the left. But this fact is insignificant compared to the substantial advantages of $b = 2$ over other radices (cf. also Theorem 4.2.2C and exercises 4.2.2–13, 15, 21), especially since floating binary can be made as fast as floating hex with only a tiny increase in total processor cost.

7. For example, suppose that $\sum_m (F(10^{km} \cdot 5^k) - F(10^{km})) = \log 5^k / \log 10^k$ and also that $\sum_m (F(10^{km} \cdot 4^k) - F(10^{km})) = \log 4^k / \log 10^k$; then

$$\sum_m (F(10^{km} \cdot 5^k) - F(10^{km} \cdot 4^k)) = \log_{10} \frac{5}{4}$$

for all k. But now let ϵ be a small positive number, and choose $\delta > 0$ so that $F(x) < \epsilon$ for $0 < x < \delta$, and choose $M > 0$ so that $F(x) > 1 - \epsilon$ for $x > M$. We can take k so large that $10^{-k} \cdot 5^k < \delta$ and $4^k > M$; hence by the monotonicity of F,

$$\begin{aligned}\sum_m (F(10^{km} \cdot 5^k) - F(10^{km} \cdot 4^k)) &\le \sum_{m \le 0} (F(10^{km} \cdot 5^k) - F(10^{k(m-1)} \cdot 5^k)) \\ &\quad + \sum_{m \ge 0} (F(10^{k(m+1)} \cdot 4^k) - F(10^{km} \cdot 4^k)) \\ &= F(10^{-k}5^k) + 1 - F(10^k 4^k) < 2\epsilon.\end{aligned}$$

8. When $s > r$, $P_0(10^n s)$ is 1 for small n, and 0 when $\lfloor 10^n s \rfloor > \lfloor 10^n r \rfloor$. The least n for which this happens may be arbitrarily large, so no uniform bound can be given for $N_0(\epsilon)$ independent of s. (In general, calculus textbooks prove that such a uniform bound would imply that the limit function $S_0(s)$ would be continuous, and it isn't.)

9. Let $q_1, q_2, \ldots$ be such that $P_0(n) = q_1\binom{n-1}{0} + q_2\binom{n-1}{1} + \cdots$ for all n. It follows that $P_m(n) = 1^{-m} q_1\binom{n-1}{0} + 2^{-m} q_2\binom{n-1}{1} + \cdots$ for all m and n.

10. When $1 < r < 10$ the generating function $C(z)$ has simple poles at the points $1 + w_n$, where $w_n = 2\pi ni/\ln 10$, hence

$$C(z) = \frac{\log_{10} r - 1}{1 - z} + \sum_{n\neq 0} \frac{1 + w_n}{w_n} \frac{e^{-w_n \ln r} - 1}{(\ln 10)(z - 1 - w_n)} + E(z)$$

where $E(z)$ is analytic in the entire plane. Thus if $\theta = \arctan(2\pi/\ln 10)$,

$$\begin{aligned} c_m &= \log_{10} r - 1 - \frac{2}{\ln 10} \sum_{n>0} \Re\left(\frac{e^{-w_n \ln r} - 1}{w_n(1 + w_n)^m}\right) + e_m \\ &= \log_{10} r - 1 + \frac{\sin(m\theta + 2\pi \log_{10} r) - \sin(m\theta)}{\pi(1 + 4\pi^2/(\ln 10)^2)^{m/2}} + O\left(\frac{1}{(1 + 16\pi^2/(\ln 10)^2)^{m/2}}\right). \end{aligned}$$

11. When $(\log_b U) \bmod 1$ is uniformly distributed in $[0, 1)$, so is $(\log_b 1/U) \bmod 1 = (1 - \log_b U) \bmod 1$.

12. We have $h(z) = \int_{1/b}^{z} f(x)\,dx\, g(z/bx)/bx + \int_{z}^{1} f(x)\,dx\, g(z/x)/x$, and the same holds for $l(z) = \int_{1/b}^{z} f(x)\,dx\, l(z/bx)/bx + \int_{z}^{1} f(x)\,dx\, l(z/x)/x$, hence

$$\frac{h(z) - l(z)}{l(z)} = \int_{1/b}^{z} f(x)\,dx \frac{g(z/bx) - l(z/bx)}{l(z/bx)} + \int_{z}^{1} f(x)\,dx \frac{g(z/x) - l(z/x)}{l(z/x)}.$$

Since $f(x) \geq 0$, $|(h(z) - l(z))/l(z)| \leq \int_{1/b}^{z} f(x)\,dx\, A(g) + \int_{z}^{1} f(x)\,dx\, A(g)$ for all z, hence $A(h) \leq A(g)$. By symmetry, $A(h) \leq A(f)$. [*Bell System Tech. J.* **49** (1970), 1609–1625.]

13. Let $X = (\log_b U) \bmod 1$ and $Y = (\log_b V) \bmod 1$, so that X and Y are independently and uniformly distributed in $[0, 1)$. No left shift is needed if and only if $X + Y \geq 1$, and this occurs with probability $\frac{1}{2}$.

(Similarly, the probability is $\frac{1}{2}$ that floating point division by Algorithm 4.2.1M needs no normalization shifts; this needs only the weaker assumption that both of the operands independently have the *same* distribution.)

14. For convenience, the calculations are shown here for $b = 10$. If $k = 0$, the probability of a carry is

$$\left(\frac{1}{\ln 10}\right)^2 \int_{\substack{1\leq x,y\leq 10 \\ x+y\geq 10}} \frac{dx}{x} \frac{dy}{y}.$$

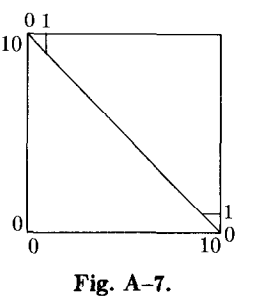


Fig. A–7.

(See Fig. A–7.) The value of the integral is

$$\int_0^{10} \frac{dy}{y} \int_{10-y}^{10} \frac{dx}{x} - 2\int_0^{1} \frac{dy}{y} \int_{10-y}^{10} \frac{dx}{x},$$

and

$$\int_0^t \frac{dy}{y} \ln\left(\frac{1}{1 - y/10}\right) = \int_0^t \left(\frac{1}{10} + \frac{y}{200} + \frac{y^2}{3000} + \cdots\right) dy = \frac{t}{10} + \frac{t^2}{400} + \frac{t^3}{9000} + \cdots.$$

(The latter integral is essentially a "dilogarithm.") Hence the probability of a carry when $k = 0$ is $(1/\ln 10)^2(\pi^2/6 - 2\sum_{n\geq 1} 1/n^2 10^n) = .27154$. [*Note:* When $b = 2$ and $k = 0$, fraction overflow *always* occurs, so this derivation proves that $\sum_{n\geq 1} 1/n^2 2^n = \pi^2/12 - \frac{1}{2}(\ln 2)^2$.]

When $k > 0$, the probability is

$$\Big(\frac{1}{\ln 10}\Big)^2 \int_{10^{1-k}}^{10^{1-k}} \frac{dy}{y} \int_{10-y}^{10} \frac{dx}{x} = \Big(\frac{1}{\ln 10}\Big)^2 \Big(\sum_{n\geq 1} \frac{1}{n^2 10^{nk}} - \sum_{n\geq 1} \frac{1}{n^2 10^{n(k+1)}}\Big).$$

Thus when $b = 10$, fraction overflow should occur with probability $.272p_0 + .017p_1 + .002p_2 + \cdots$. When $b = 2$ the corresponding figures are $p_0 + .655p_1 + .288p_2 + .137p_3 + .067p_4 + \cdots$.

Now if we use the probabilities from Table 1, dividing by .91 to eliminate zero operands and assuming that the probabilities are independent of the operand signs, we predict a probability of about 14 percent when $b = 10$, instead of the 15 percent in exercise 1. For $b = 2$, we predict about 48 percent, while the table yields 44 percent. These results are certainly in agreement within the limits of experimental error.

15. When $k = 0$, the leading digit is 1 if and only if there is a carry. (It is possible for fraction overflow and subsequent rounding to yield a leading digit of 2, when $b \geq 4$, but we are ignoring rounding in this exercise.) The probability of fraction overflow is $.272 < \log_{10} 2$, as shown in the previous exercise.

When $k > 0$, the leading digit is 1 with probability

$$\Big(\frac{1}{\ln 10}\Big)^2 \Big(\int_{10^{1-k}}^{10^{1-k}} \frac{dy}{y} \int_{\substack{1\leq x<2-y \\ \text{or } 10-y\leq x<10}} \frac{dx}{x}\Big) < \Big(\frac{1}{\ln 10}\Big)^2 \Big(\int_{10^{1-k}}^{10^{1-k}} \frac{dy}{y} \int_{1\leq x\leq 2} \frac{dx}{x}\Big) = \log_{10} 2.$$

16. To prove the hint [which is due to Landau, *Prace matematyczno-fizyczne* **21** (1910), 103–113], assume first that $\limsup a_n = \lambda > 0$. Let $\epsilon = \lambda/(\lambda + 4M)$ and choose N so that $|a_1 + \cdots + a_n| < \frac{1}{10}\epsilon\lambda n$ for all $n > N$. Let $n > N/(1-\epsilon)$, $n > 5/\epsilon$ be such that $a_n > \frac{1}{2}\lambda$. Then, by induction, $a_{n-k} \geq a_n - kM/(n-\epsilon n) > \frac{1}{4}\lambda$ for $0 \leq k < \epsilon n$, and $\sum_{n-\epsilon n<k\leq n} a_k \geq \frac{1}{4}\lambda(\epsilon n - 1) > \frac{1}{5}\lambda\epsilon n$. But

$$\Big|\sum_{n-\epsilon n<k\leq n} a_k\Big| = \Big|\sum_{1\leq k\leq n} a_k - \sum_{1\leq k\leq n-\epsilon n} a_k\Big| \leq \tfrac{1}{5}\epsilon\lambda n$$

since $n - \epsilon n > N$. A similar contradiction applies if $\liminf a_n < 0$.

Assuming that $P_{m+1}(n) \to \lambda$ as $n \to \infty$, let $a_k = P_m(k) - \lambda$. If $m > 0$, the a_k satisfy the hypotheses of the hint (cf. Eq. 4.2.2–15), since $0 \leq P_m(k) \leq 1$; hence $P_m(n) \to \lambda$.

17. See *Fibonacci Quarterly* **7** (1969), 474–475. (Persi Diaconis [Ph.D. thesis, Harvard University, 1974] has shown among other things that the definition of probability by repeated averaging is weaker than harmonic probability, in the following precise sense: If $\lim_{m\to\infty} \liminf_{n\to\infty} P_m(n) = \lim_{m\to\infty} \limsup_{n\to\infty} P_m(n) = \lambda$ then the harmonic probability is λ. On the other hand the statement "$10^{k^2} \leq n < 10^{k^2+k}$ for some integer $k > 0$" has logarithmic probability $\frac{1}{2}$, while repeated averaging never settles down to give it any particular probability.)

18. Let $p(a) = P(L_a)$ and $p(a,b) = \sum_{a \le k < b} p(k)$ for $1 \le a < b$. Since $L_a = L_{10a} \cup L_{10a+1} \cup \cdots \cup L_{10a+9}$ for all a, we have $p(a) = p(10a, 10(a+1))$ by (i). Furthermore since $P(S) = P(2S) + P(2S+1)$ by (i), (ii), (iii), we have $p(a) = p(2a, 2(a+1))$. It follows that $p(a,b) = p(2^m 10^n a, 2^m 10^n b)$ for all $m, n \ge 0$.

If $1 < b/a < b'/a'$, then $p(a,b) \le p(a',b')$. The reason is that there exist integers m, n, m', n' such that $2^{m'}10^{n'}a' \le 2^m 10^n a < 2^m 10^n b \le 2^{m'}10^{n'}b'$ as a consequence of the fact that $\log 2/\log 10$ is irrational, hence we can apply (v). (Cf. exercise 3.5–22 with $k = 1$ and $U_n = n \log 2/\log 10$.) In particular, $p(a) \ge p(a+1)$, and it follows that $p(a,b)/p(a,b+1) \ge (b-a)/(b+1-a)$. (Cf. Eq. 4.2.2–15.)

Now we can prove that $p(a,b) = p(a',b')$ whenever $b/a = b'/a'$; for $p(a,b) = p(10^n a, 10^n b) \le c_n p(10^n a, 10^n b - 1) \le c_n p(a',b')$, for arbitrarily large values of n, where $c_n = 10^n(b-a)/(10^n(b-a)-1) = 1 + O(10^{-n})$.

For any positive integer n we have $p(a^n, b^n) = p(a^n, ba^{n-1}) + p(ba^{n-1}, b^2a^{n-2}) + \cdots + p(b^{n-1}a, b^n) = np(a,b)$. If $10^m \le a^n \le 10^{m+1}$ and $10^{m'} \le b^n \le 10^{m'+1}$, then $p(10^{m+1}, 10^{m'}) \le p(a^n, b^n) \le p(10^m, 10^{m'+1})$ by (v). But $p(1,10) = 1$ by (iv), hence $p(10^m, 10^{m'}) = m' - m$ for all $m' \ge m$. We conclude that $\lfloor \log_{10} b^n \rfloor - \lfloor \log_{10} a^n \rfloor - 1 \le np(a,b) \le \lfloor \log_{10} b^n \rfloor + \lfloor \log_{10} a^n \rfloor + 1$ for all n, and $p(a,b) = \log_{10}(b/a)$.

[This exercise was inspired by D. I. A. Cohen, who proved a slightly weaker result in *J. Combinatorial Theory* (A) **20** (1976), 367–370.]

SECTION 4.3.1

2. If the ith number to be added is $u_i = (u_{i1}u_{i2}\ldots u_{in})_b$, use Algorithm A with step A2 changed to the following:

A2′. [Add digits.] Set

$$w_j \leftarrow (u_{1j} + \cdots + u_{mj} + k) \bmod b, \quad \text{and} \quad k \leftarrow \lfloor (u_{1j} + \cdots + u_{mj} + k)/b \rfloor.$$

(The maximum value of k is $m-1$, so step A3 would have to be altered if $m > b$.)

3.

```
    ENT1 N          1
    JOV  OFLO       1       Ensure overflow is off.
    ENTA 0          1       k ← 0.
2H  ENT2 0          N       (rI2 ≡ next value of k)
    ENT3 M*N-N,1    N       (LOC(u_ij) ≡ U + n(i-1) + j)
3H  ADD  U,3        MN      rA ← rA + u_ij.
    JNOV *+2        MN
    INC2 1          K       Carry one.
    DEC3 N          MN      Repeat for m ≥ i ≥ 0.
    J3P  3B         MN      (rI3 ≡ n(i-1) + j)
    STA  W,1        N       w_j ← rA.
    ENTA 0,2        N       k ← rI2.
    DEC1 1          N
    J1P  2B         N       Repeat for n ≥ j ≥ 0.
    STA  W          1       Store final carry in w_0.  ∎
```

Running time, assuming that $K = \frac{1}{2}MN$, is $5.5MN + 7N + 4$ cycles.

4. We may make the following assertion before A1: "$n \geq 1$; and $0 \leq u_i, v_i < b$ for $1 \leq i \leq n$." Before A2, we assert: "$1 \leq j \leq n$; $0 \leq u_i, v_i < b$ for $1 \leq i \leq n$; $0 \leq w_i < b$ for $j < i \leq n$; $0 \leq k \leq 1$; and

$$(u_{j+1} \ldots u_n)_b + (v_{j+1} \ldots v_n)_b = (kw_{j+1} \ldots w_n)_b.$$"

The latter statement means more precisely that

$$\sum_{j<t\leq n} u_t b^{n-t} + \sum_{j<t\leq n} v_t b^{n-t} = kb^{n-j} + \sum_{j<t\leq n} w_t b^{n-t}.$$

Before A3, we assert: "$1 \leq j \leq n$; $0 \leq u_i, v_i < b$ for $1 \leq i \leq n$; $0 \leq w_i < b$ for $j \leq i \leq n$; $0 \leq k \leq 1$; and $(u_j \ldots u_n)_b + (v_j \ldots v_n)_b = (kw_j \ldots w_n)_b$." After A3, we assert that $0 \leq w_i < b$ for $1 \leq i \leq n$; $0 \leq w_0 \leq 1$; and $(u_1 \ldots u_n)_b + (v_1 \ldots v_n)_b = (w_0 \ldots w_n)_b$.

It is a simple matter to complete the proof by verifying the necessary implications between the assertions and by showing that the algorithm always terminates.

5. **B1.** Set $j \leftarrow 1$, $w_0 \leftarrow 0$.

B2. Set $t \leftarrow u_j + v_j$, $w_j \leftarrow t \bmod b$, $i \leftarrow j$.

B3. If $t \geq b$, set $i \leftarrow i - 1$, $t \leftarrow w_i + 1$, $w_i \leftarrow t \bmod b$, and repeat this step until $t < b$.

B4. Increase j by one, and if $j \leq n$ go back to B2. ∎

6. **C1.** Set $j \leftarrow 1$, $i \leftarrow 0$, $r \leftarrow 0$.

C2. Set $t \leftarrow u_j + v_j$. If $t \geq b$, set $w_i \leftarrow r + 1$, $w_k \leftarrow 0$ for $i < k < j$; then set $i \leftarrow j$ and $r \leftarrow t \bmod b$. Otherwise if $t < b - 1$, set $w_i \leftarrow r$, $w_k \leftarrow b - 1$ for $i < k < j$; then set $i \leftarrow j$ and $r \leftarrow t$.

C3. Increase j by one. If $j \leq n$, go back to C2; otherwise set $w_i \leftarrow r$, and $w_k \leftarrow b - 1$ for $i < k \leq n$. ∎

7. When $j = 3$, for example, we have $k = 0$ with probability $(b+1)/2b$; $k = 1$ with probability $((b-1)/2b)(1 - 1/b)$, namely the probability that a carry occurs and that the preceding digit wasn't $b - 1$; $k = 2$ with probability $((b-1)/2b)(1/b)(1 - 1/b)$; and $k = 3$ with probability $((b-1)/2b)(1/b)(1/b)(1)$. For fixed k we may add the probabilities as j varies from 1 to n; this gives the mean number of times the carry propagates back k places,

$$m_k = \frac{b-1}{2b^k}\Big((n+1-k)\Big(1-\frac{1}{b}\Big)+\frac{1}{b}\Big).$$

As a check, we find that the average number of carries is

$$m_1 + 2m_2 + \cdots + nm_n = \frac{1}{2}\left(n - \frac{1}{b-1}\left(1 - \left(\frac{1}{b}\right)^n\right)\right),$$

in agreement with (6).

```
8.     ENN1 N        1          2H LDA  W+N+1,2 K
       JOV  OFLO     1             INCA 1       K
       STZ  W        1             STA  W+N+1,2 K
     1H LDA  U+N+1,1 N             DEC2 1       K
       ADD  V+N+1,1 N              JOV  2B      K
       STA  W+N+1,1 N           3H INC2 1       N
       JNOV 3F       N             J2N  1B      N
       ENT2 -1,1     L                                  ∎
```

The running time depends on L, the number of positions in which $u_j + v_j \geq b$; and on K, the total number of carries. It is not difficult to see that K is the same quantity that appears in Program A. The analysis in the text shows that L has the average value $N((b-1)/2b)$, and K has the average value $\frac{1}{2}(N - b^{-1} - b^{-2} - \cdots - b^{-n})$. So if we ignore terms of order $1/b$, the running time is $9N + L + 7K + 3 \approx 13N + 3$ cycles.

Note: Since a carry occurs almost half of the time, it would be more efficient to delay storing the result by one step. This leads to a somewhat longer program whose running time is approximately $12N + 5$ cycles, based on the somewhat more detailed information calculated in exercise 7.

9. Replace "b" by "b_{n-j}" everywhere in step A2.

10. If lines 06 and 07 were interchanged, we would almost always have overflow, but register A might have a negative value at line 08, so this would not work. If the instructions on lines 05 and 06 were interchanged, the sequence of overflows occurring in the program would be slightly different in some cases, but the program would still be right.

11. (a) Set $j \leftarrow 1$; (b) if $u_j < v_j$, terminate $[u < v]$; if $u_j = v_j$ and $j = n$, terminate $[u = v]$; if $u_j = v_j$ and $j < n$, set $j \leftarrow j + 1$ and repeat (b); if $u_j > v_j$, terminate $[u > v]$. This algorithm tends to be quite fast, since there is usually low probability that j will have to get very high before we encounter a case with $u_j \neq v_j$.

12. Use Algorithm S with $u_j = 0$ and $v_j = w_j$. Another "borrow" will occur at the end of the algorithm; this time it should be ignored.

```
13.    ENTX N      1          ADD  CARRY N
       JOV  OFLO   1          JNOV *+2   N
       ENTX 0      1          INCX 1     K
     2H STX  CARRY N          STA  W,1   N
       LDA  U,1    N          DEC1 1     N
       MUL  V      N          J1P  2B    N
       SLC  5      N          STX  W     1   ∎
```

The running time is $23N + K + 5$ cycles, and K is roughly $\frac{1}{2}N$.

14. The key inductive assertion is the one that should be valid at the beginning of step M4; all others are readily filled in from this one, which is as follows: "$1 \leq i \leq n$; $1 \leq j \leq m$; $0 \leq u_r < b$ for $1 \leq r \leq n$; $0 \leq v_r < b$ for $1 \leq r \leq m$; $0 \leq w_r < b$ for $j < r \leq m + n$; $0 \leq k < b$; and

$$(w_{j+1} \ldots w_{m+n})_b + kb^{m+n-i-j} = u \times (v_{j+1} \ldots v_m)_b + (u_{i+1} \ldots u_n)_b \times v_j b^{m-j}.$$"

(For the precise meaning of this notation, see the answer to exercise 4.)

15. The error is nonnegative and less than $(n-2)b^{-n-1}$. [Similarly, if we ignore the products with $i+j > n+3$, the error is bounded by $(n-3)b^{-n-2}$, etc.; but, in some cases, we must compute all of the products if we want to get the true rounded result.]

16. **S1.** Set $r \leftarrow 0$, $j \leftarrow 1$.

S2. Set $w_j \leftarrow \lfloor (rb+u_j)/v \rfloor$, $r \leftarrow (rb+u_j) \bmod v$.

S3. Increase j by 1, and return to S2 if $j \le n$. ∎

17. $u/v > u_0 b^n/(v_1+1)b^{n-1} = b(1-1/(v_1+1)) > b(1-1/(b/2)) = b-2$.

18. $(u_0 b + u_1)/(v_1+1) \le u/(v_1+1)b^{n-1} < u/v$.

19. $u - \hat{q}v \le u - \hat{q}v_1 b^{n-1} - \hat{q}v_2 b^{n-2} = u_2 b^{n-2} + \cdots + u_n + \hat{r}b^{n-1} - \hat{q}v_2 b^{n-2} < b^{n-2}(u_2 + 1 + \hat{r}b - \hat{q}v_2) \le 0$. Since $u - \hat{q}v < 0$, $q < \hat{q}$.

20. If $q \le \hat{q} - 2$, then $u < (\hat{q}-1)v < \hat{q}(v_1 b^{n-1} + (v_2+1)b^{n-2}) - v < \hat{q}v_1 b^{n-1} + \hat{q}v_2 b^{n-2} + b^{n-1} - v \le \hat{q}v_1 b^{n-1} + (b\hat{r} + u_2)b^{n-2} + b^{n-1} - v = u_0 b^n + u_1 b^{n-1} + u_2 b^{n-2} + b^{n-1} - v \le u_0 b^n + u_1 b^{n-1} + u_2 b^{n-2} \le u$. In other words, $u < u$, and this is a contradiction.

21. (Solution by G. K. Goyal.) The inequality $v_2\hat{q} \le b\hat{r} + u_2$ implies that we have $\hat{q} \le (u_0 b^2 + u_1 b + u_2)/(v_1 b + v_2) \le u/((v_1 b + v_2)b^{n-2})$. Now $u \bmod v = u - qv = v(1-\alpha)$ where $0 \le \alpha = q - u/v \le \hat{q} - u/v \le u(1/((v_1 b + v_2)b^{n-2}) - 1/v) = u(v_3 b^{n-3} + \cdots)/((v_1 b + v_2)b^{n-2}v) < u/(v_1 b v) \le \hat{q}/(v_1 b) \le (b-1)/(v_1 b)$, and this is at most $2/b$ since $v_1 \ge \frac{1}{2}(b-1)$.

22. Let $u = 4100$, $v = 588$. We first try $\hat{q} = \lfloor \frac{41}{5} \rfloor = 8$, but $8 \cdot 8 > 10(41-40) + 0$. Then we set $\hat{q} = 7$, and now we find $7 \cdot 8 < 10(41-35) + 0$. But 7 times 588 equals 4116, so the true quotient is $q = 6$. (Incidentally, this example shows that Theorem B cannot be improved under the given hypotheses, when $b = 10$.)

23. Obviously $v\lfloor b/(v+1) \rfloor < (v+1)\lfloor b/(v+1) \rfloor \le (v+1)b/(v+1) = b$; also if $v \ge \lfloor b/2 \rfloor$ we obviously have $v\lfloor b/(v+1) \rfloor \ge v \ge \lfloor b/2 \rfloor$. Finally, assume that we have $1 \le v < \lfloor b/2 \rfloor$. Then $v\lfloor b/(v+1) \rfloor > v(b/(v+1) - 1) \ge b/2 - 1 \ge \lfloor b/2 \rfloor - 1$, because $v(b/(v+1) - 1) - (b/2 - 1) = (b/2 - v - 1)(v-1)/(v+1) \ge 0$. Since $v\lfloor b/(v+1) \rfloor > \lfloor b/2 \rfloor - 1$, we must have $v\lfloor b/(v+1) \rfloor \ge \lfloor b/2 \rfloor$.

24. The approximate probability is only $\log_b 2$, not $\frac{1}{2}$. (For example, if $b = 2^{35}$, the probability is approximately $\frac{1}{35}$; this is still high enough to warrant the special test for $d = 1$ in steps D1 and D8.)

25.

002		ENTA 1	1	
003		ADD V+1	1	
004		STA TEMP	1	
005		ENTA 1	1	
006		JOV 1F	1	Jump if $v_1 = b-1$.
007		ENTX 0	1	
008		DIV TEMP	1	Otherwise compute $b/(v_1+1)$.
009		JOV DIVBYZERO	1	Jump if $v_1 = 0$.
010	1H	STA D	1	
011		DECA 1	1	
012		JANZ *+3	1	Jump if $d \neq 1$.
013		STZ U	$1-A$	
014		JMP D2	$1-A$	

```
015      ENT1 N              A        Multiply v by d.
016      ENTX 0              A
017   2H STX  CARRY          AN
018      LDA  V,1            AN
019      MUL  D              AN
...                                   (as in exercise 13)
026      J1P  2B             AN
027      ENT1 M+N            A        (Now rX = 0.)
028   2H STX  CARRY       A(M+N)      Multiply u by d.
029      LDA  U,1         A(M+N)
...                                   (as in exercise 13)
037      J1P  2B          A(M+N)
038      STX  U              A        ∎
```

26. (See the algorithm of exercise 16.)

```
101   D8 LDA  D              1    (Remainder will be left in
102      DECA 1              1         locations U+M+1 through U+M+N)
103      JAZ  DONE           1    Terminate if d = 1.
104      ENN1 N              A    rI1 ≡ j − n − 1; j ← 1.
105      ENTA 0              A    r ← 0.
106   1H LDX  U+M+N+1,1      AN   rAX ← rb + u_{m+j}.
107      DIV  D              AN
108      STA  U+M+N+1,1      AN
109      SLAX 5              AN   r ← (rb + u_{m+j}) mod d.
110      INC2 1              AN   j ← j + 1.
111      J2N  1B             AN   Repeat for 1 ≤ j ≤ n.  ∎
```

At this point, the division routine is complete; and by the next exercise, register AX is zero.

27. It is $du \bmod dv = d(u \bmod v)$.

28. For convenience, let us assume that v has a decimal point at the left, i.e., $v = (v_0.v_1v_2\ldots)_b$. After step N1 we have $\frac{1}{2} \le v < 1 + 1/b$: for

$$v\left\lfloor\frac{b+1}{v_1+1}\right\rfloor \le \frac{v(b+1)}{v_1+1} = \frac{v(1+1/b)}{(1/b)(v_1+1)} < 1+\frac{1}{b},$$

and

$$v\left\lfloor\frac{b+1}{v_1+1}\right\rfloor \ge \frac{v(b+1-v_1)}{v_1+1} \ge \frac{1}{b}\frac{v_1(b+1-v_1)}{v_1+1}.$$

The latter quantity takes its smallest value when $v_1 = 1$, since it is a convex function and the other extreme value is greater.

The formula in step N2 may be written

$$v \leftarrow \left\lfloor\frac{b(b+1)}{v_1+1}\right\rfloor\frac{v}{b},$$

so we see as above that v will never become $\ge 1 + 1/b$.

The minimum value of v after one iteration of step N2 is $\geq$

$$\left(\frac{b(b+1)-v_1}{v_1+1}\right)\frac{v}{b} \geq \left(\frac{b(b+1)-v_1}{v_1+1}\right)\frac{v_1}{b^2} = \left(\frac{b(b+1)+1-t}{t}\right)\left(\frac{t-1}{b^2}\right)$$
$$= 1+\frac{1}{b}+\frac{2}{b^2}-\frac{1}{b^2}\left(t+\frac{b(b+1)+1}{t}\right),$$

if $t = v_1 + 1$. The minimum of this quantity occurs for $t = b/2 + 1$; a lower bound is $1 - 3/2b$. Hence $v_1 \geq b - 2$, after one iteration of step N2. Finally, we have $(1-3/2b)(1+1/b)^2 > 1$, when $b \geq 5$, so at most two more iterations are needed. The assertion is easily verified when $b < 5$.

29. True, since $(u_j \ldots u_{j+n})_b < v$.

30. In Algorithms A and S, such overlap is possible if the algorithms are rewritten slightly; e.g., in Algorithm A, we could rewrite step A2 thus: "Set $t \leftarrow u_j + v_j + k$, $w_j \leftarrow t \bmod b$, $k \leftarrow \lfloor t/b \rfloor$."

In Algorithm M, v_j may be in the same location as w_j. In Algorithm D, it is most convenient (as in Program D, exercise 26) to let $r_1 \ldots r_n$ be the same as $u_{m+1} \ldots u_{m+n}$; and we can also have $q_0 \ldots q_m$ the same as $u_0 \ldots u_m$, provided that no alteration of u_j is made in step D6. (Line 098 of Program D can safely be changed to "J1P 2B", since u_j isn't used in the subsequent calculation.)

31. Consider the situation of Fig. 6 with $u = (u_j u_{j+1} \ldots u_{j+n})_3$ as in Algorithm D. If the leading nonzero digits of u and v have the same sign, set $r \leftarrow u - v$, $q \leftarrow 1$; otherwise set $r \leftarrow u + v$, $q \leftarrow -1$. Now if $|r| > |u|$, or if $|r| = |u|$ and the first nonzero digit of $u_{j+n+1} \ldots u_{m+n}$ has the same sign as the first nonzero digit of r, set $q \leftarrow 0$; otherwise set $u_j \ldots u_{j+n}$ equal to the digits of r.

36. Values to 1000 decimal and 1100 octal places have been computed by R. P. Brent, Comp. Centre Tech. Rep. 47 (Canberra: Australian Nat. Univ., 1975).

37. Let $d = 2^e$ so that $b > dv_1 \geq b/2$. Instead of normalizing u and v in step D1, simply compute the two leading digits $v'_1 v'_2$ of $2^e(v_1 v_2 v_3)_b$ by shifting left e bits. In step D3, use (v'_1, v'_2) instead of (v_1, v_2) and $(u'_j, u'_{j+1}, u'_{j+2})$ instead of (u_j, u_{j+1}, u_{j+2}), where the digits $u'_j u'_{j+1} u'_{j+2}$ are obtained from $(u_j u_{j+1} u_{j+2} u_{j+3})_b$ by shifting left e bits. Omit division by d in step D8. (In essence, u and v are being "virtually" shifted. This method saves computation when m is small compared to n.)

SECTION 4.3.2

1. The solution is unique since $7 \cdot 11 \cdot 13 = 1001$. The "constructive" proof of Theorem C tells us that the answer is $((11 \cdot 13)^6 + 6 \cdot (7 \cdot 13)^{10} + 5 \cdot (7 \cdot 11)^{12}) \bmod 1001$. But this answer is perhaps not explicit enough! By (23) we have $v_1 = 1$, $v_2 = (6-1) \cdot 8 \bmod 11 = 7$, $v_3 = ((5-1) \cdot 2 - 7) \cdot 6 \bmod 13 = 6$, so $u = 6 \cdot 7 \cdot 11 + 7 \cdot 7 + 1 = 512$.

2. No. There is at most one such u; the additional condition $u_1 \equiv \cdots \equiv u_r$ (modulo 1) is necessary and sufficient, and it follows that such a generalization is not very interesting.

3. $u \equiv u_i$ (modulo m_i) implies that $u \equiv u_i$ (modulo $\gcd(m_i, m_j)$), so the condition $u_i \equiv u_j$ (modulo $\gcd(m_i, m_j)$) must surely hold if there is a solution. Furthermore if $u \equiv v$ (modulo m_j) for all j, then $u - v$ is a multiple of $\operatorname{lcm}(m_1, \ldots, m_r) = m$; hence there is at most one solution.

The proof can now be completed in a nonconstructive manner by counting the number of different r-tuples $(u_1, \ldots, u_r)$ satisfying the conditions $0 \le u_j < m_j$ and $u_i \equiv u_j$ (modulo $\gcd(m_i, m_j)$). If this number is m, there must be a solution since $(u \bmod m_1, \ldots, u \bmod m_r)$ takes on m distinct values as u goes from a to $a + m$. Assume that $u_1, \ldots, u_{r-1}$ have been chosen satisfying the given conditions; we must now pick $u_r \equiv u_j$ (modulo $\gcd(m_j, m_r)$) for $1 \le j < r$, and by the generalized Chinese remainder theorem for $r - 1$ elements there are

$$\begin{aligned} m_r/\operatorname{lcm}(\gcd(m_1, m_r), \ldots, \gcd(m_{r-1}, m_r)) &= m_r/\gcd(\operatorname{lcm}(m_1, \ldots, m_{r-1}), m_r) \\ &= \operatorname{lcm}(m_1, \ldots, m_r)/\operatorname{lcm}(m_1, \ldots, m_{r-1}) \end{aligned}$$

ways to do this. [This proof is based on identities (10), (11), (12), and (14) of Section 4.5.2.]

A constructive proof [A. S. Fraenkel, *Proc. Amer. Math. Soc.* **14** (1963), 790–791] generalizing (24) can be given as follows. Let $M_j = \operatorname{lcm}(m_1, \ldots, m_j)$; we wish to find $u = v_r M_{r-1} + \cdots + v_2 M_1 + v_1$, where $0 \le v_j < M_j/M_{j-1}$. Assume that $v_1, \ldots, v_{j-1}$ have already been determined; then we must solve the congruence

$$v_j M_{j-1} + v_{j-1} M_{j-2} + \cdots + v_1 \equiv u_j \text{ (modulo } m_j).$$

Here $v_{j-1} M_{j-2} + \cdots + v_1 \equiv u_i \equiv u_j$ (modulo $\gcd(m_i, m_j)$) for $i < j$ by hypothesis, so $c = u_j - (v_{j-1} M_{j-2} + \cdots + v_1)$ is a multiple of

$$\operatorname{lcm}(\gcd(m_1, m_j), \ldots, \gcd(m_{j-1}, m_j)) = \gcd(M_{j-1}, m_j) = d_j.$$

We therefore must solve $v_j M_{j-1} \equiv c$ (modulo m_j). By Euclid's algorithm there is a number c_j such that $c_j M_{j-1} \equiv d_j$ (modulo m_j); hence we may take

$$v_j = (c_j\, c)/d_j \bmod (m_j/d_j).$$

Note that, as in the nonconstructive proof, we have $m_j/d_j = M_j/M_{j-1}$.

4. (After $m_4 = 91 = 7 \cdot 13$, we have used up all products of two or more odd primes that can be less than 100, so $m_5, \ldots$ must all be prime.)

$$\begin{array}{lllll} m_7 = 79, & m_8 = 73, & m_9 = 71, & m_{10} = 67, & m_{11} = 61, \\ m_{12} = 59, & m_{13} = 53, & m_{14} = 47, & m_{15} = 43, & m_{16} = 41, \\ m_{17} = 37, & m_{18} = 31, & m_{19} = 29, & m_{20} = 23, & m_{21} = 17, \end{array}$$

and then we are stuck ($m_{22} = 1$ does no good).

5. No. The obvious upper bound,

$$3^4 5^2 7^2 11^1 \cdots = \prod_{\substack{p \text{ odd} \\ p \text{ prime}}} p^{\lfloor \log_p 100 \rfloor},$$

is attained if we choose $m_1 = 3^4$, $m_2 = 5^2$, etc. (It is more difficult, however, to maximize $m_1 \ldots m_r$ when r is fixed, or to maximize $m_1 + \cdots + m_r$ as we would attempt to do when using moduli $2^{m_j} - 1$.)

6. (a) If $e = f + kg$, then $2^e = 2^f(2^g)^k \equiv 2^f \cdot 1^k$ (modulo $2^g - 1$). So if $2^e \equiv 2^f$ (modulo $2^g - 1$), we have $2^{e \bmod g} \equiv 2^{f \bmod g}$ (modulo $2^g - 1$); and since the latter quantities lie between zero and $2^g - 1$ we must have $e \bmod g = f \bmod g$. (b) By part (a), $(1 + 2^d + \cdots + 2^{(c-1)d}) \cdot (2^e - 1) \equiv (1 + 2^d + \cdots + 2^{(c-1)d}) \cdot (2^d - 1) = 2^{cd} - 1 \equiv 2^{ce} - 1 \equiv 2^1 - 1 = 1$ (modulo $2^f - 1$).

7.
$$\begin{aligned}(u_j - (v_1 + m_1(v_2 + m_2(v_3 + \cdots + m_{j-2}v_{j-1})\ldots)))\,c_{1j}\ldots c_{(j-1)j} &= (u_j - v_1)\,c_{1j}\ldots c_{(j-1)j} - m_1v_2c_{1j}\ldots c_{(j-1)j} - \cdots \\ &\qquad - m_1\ldots m_{j-2}v_{j-1}c_{1j}\ldots c_{(j-1)j} \\ &\equiv (u_j - v_1)\,c_{1j}\ldots c_{(j-1)j} - v_2c_{2j}\ldots c_{(j-1)j} - \cdots - v_{j-1}c_{(j-1)j} \\ &= (\ldots((u_j - v_1)\,c_{1j} - v_2)\,c_{2j} - \cdots - v_{j-1})\,c_{(j-1)j} \text{ (modulo } m_j).\end{aligned}$$

This method of rewriting the formulas uses the same number of arithmetic operations and fewer constants; but the number of constants is fewer only if we order the moduli so that $m_1 < m_2 < \cdots < m_r$, otherwise we would need a table of $m_i \bmod m_j$. This ordering of the moduli might seem to require more computation than if we made m_1 the largest, m_2 the next largest, etc., since there are many more operations to be done modulo m_r than modulo m_1; but since v_j can be as large as $m_j - 1$, we are better off with $m_1 < m_2 < \cdots < m_r$ in (23) also. So this idea appears to be preferable to the formulas in the text, although the formulas in the text are advantageous when the moduli have the form (14), as shown in Section 4.3.3.

8. $m_{j-1}\ldots m_1v_j \equiv m_{j-1}\ldots m_1\,(\ldots((u_j - v_1)\,c_{1j} - v_2)\,c_{2j} - \cdots - v_{j-1})\,c_{(j-1)j} \equiv m_{j-2}\ldots m_1\,(\ldots(u_j - v_1)\,c_{1j} - \cdots - v_{j-2})\,c_{(j-2)j} - v_{j-1}m_{j-2}\ldots m_1 \equiv \cdots \equiv u_j - v_1 - v_2m_1 - \cdots - v_{j-1}m_{j-2}\ldots m_1$ (modulo m_j).

9. $u_r \leftarrow ((\ldots(v_rm_{r-1} + v_{r-1})\,m_{r-2} + \cdots)\,m_1 + v_1) \bmod m_r,\ \ldots,$
$u_2 \leftarrow (v_2m_1 + v_1) \bmod m_2,\ u_1 \leftarrow v_1 \bmod m_1.$

(The computation should be done in this order, if we want to let u_j and v_j share the same memory locations, as they can in (23).)

10. If we redefine the "mod" operator so that it produces residues in the symmetrical range, the basic formulas (2), (3), (4) for arithmetic and (23), (24) for conversion remain the same, and the number u in (24) lies in the desired range (10). (Here (24) is a *balanced mixed-radix* notation, generalizing "balanced ternary" notation.) The comparison of two numbers may still be done from left to right, in the simple manner described in the text. Furthermore, it is possible to retain the value u_j in a single computer word, if we have signed magnitude representation within the computer, even if m_j is almost twice the word size. But the arithmetic operations analogous to (11) and (12) are more difficult, so it appears that on most computers this idea would result in slightly slower operation.

11. Multiply by

$$\frac{m+1}{2} = \left(\frac{m_1+1}{2}, \ldots, \frac{m_r+1}{2}\right).$$

Note that $2t \cdot \frac{m+1}{2} \equiv t$ (modulo m). In general if v is relatively prime to m, then we can find (by Euclid's algorithm) a number $v' = (v'_1, \ldots, v'_r)$ such that $vv' \equiv 1$ (modulo m); and then if u is known to be a multiple of v we have $u/v = uv'$, where the latter is computed with modular multiplication. When v is not relatively prime to m, division is much more difficult.

12. Obvious from (11), if we replace m_j by m. [Another way to test for overflow, if m is odd, is to maintain extra bits $u_0 = u \bmod 2$ and $v_0 = v \bmod 2$. Then overflow has occurred iff $u_0 + v_0 \not\equiv w_1 + \cdots + w_r$ (modulo 2), where $(w_1, \ldots, w_r)$ are the mixed-radix digits corresponding to $u + v$.]

13. (a) $x^2 - x = (x-1)x \equiv 0$ (modulo 10^n) is equivalent to $(x-1)x \equiv 0$ (modulo p^n) for $p = 2$ and 5. Either x or $x-1$ must be a multiple of p, and then the other is relatively prime to p^n; so either x or $x-1$ must be a multiple of p^n. If $x \bmod 2^n = x \bmod 5^n = 0$ or 1, we must have $x \bmod 10^n = 0$ or 1; hence automorphs have $x \bmod 2^n \neq x \bmod 5^n$. (b) If $x = qp^n + r$, where $r = 0$ or 1, then $r \equiv r^2 \equiv r^3$, so $3x^2 - 2x^3 \equiv (6qp^n r + 3r) - (6qp^n r + 2r) \equiv r$ (modulo p^{2n}). (c) Let c' be the magic constant $(3(cx)^2 - 2(cx)^3)/x^2 = 3c^2 - 2c^3 x$.

Note: Since the last k digits of an n-digit automorph form a k-digit automorph, it makes sense to speak of the two ∞-digit automorphs, x and $1 - x$, which are 10-adic numbers (cf. exercise 4.1–31). The set of 10-adic numbers is equivalent under modular arithmetic to the set of ordered pairs (u_1, u_2), where u_1 is a 2-adic number and u_2 is a 5-adic number.

SECTION 4.3.3

1.

27×47:	18×42:	09×05:	2718×4742:
08	04	00	1269
08	04	00	1269
−15	+14	−45	−0045
49	16	45	0756
49	16	45	0756
1269	0756	0045	12888756

2. $\sqrt{Q + \lfloor\sqrt{Q}\rfloor} \leq \sqrt{Q + \sqrt{Q}} < \sqrt{Q + 2\sqrt{Q} + 1} = \sqrt{Q} + 1$, so $\lfloor\sqrt{Q+R}\rfloor \leq \lfloor\sqrt{Q}\rfloor + 1$.

3. When $k \leq 2$, the result is true, so assume that $k > 2$. Let $q_k = 2^{Q_k}$, $r_k = 2^{R_k}$, so that $R_k = \lfloor\sqrt{Q_k}\rfloor$ and $Q_k = Q_{k-1} + R_{k-1}$. We must show that $1 + (R_k + 1)2^{R_k} \leq 2^{Q_k - 1}$; this inequality isn't close at all, one way is to observe that $1 + (R_k + 1)2^{R_k} \leq 1 + 2^{2R_k}$ and $2R_k < Q_{k-1}$ when $k > 2$. (The fact that $2R_k < Q_{k-1}$ is readily proved by induction since $R_{k+1} - R_k \leq 1$ and $Q_k - Q_{k-1} \geq 2$.)

4. For $j = 1, \ldots, r$, calculate $U_e(j^2)$, $jU_o(j^2)$, $V_e(j^2)$, $jV_o(j^2)$; and by recursively calling the multiplication algorithm, calculate

$$W(j) = (U_e(j^2) + jU_o(j^2))(V_e(j^2) + jV_o(j^2)),$$
$$W(-j) = (U_e(j^2) - jU_o(j^2))(V_e(j^2) - jV_o(j^2)).$$

Then we have $W_e(j^2) = \frac{1}{2}(W(j) + W(-j))$, $W_o(j^2) = \frac{1}{2}(W(j) - W(-j))$. Also calculate $W_e(0) = U(0)V(0)$. Now construct difference tables for W_e and W_o, which are polynomials whose respective degrees are r and $r - 1$.

This method reduces the size of the numbers being handled, and reduces the number of additions and multiplications. Its only disadvantage is a longer program (since the control is somewhat more complex, and some of the calculations must be done with signed numbers).

Another possibility would perhaps be to evaluate W_e and W_o at 1^2, 2^2, 4^2, ..., $(2^r)^2$; although the numbers involved are larger, the calculations are faster, since all multiplications are replaced by shifting and all divisions are by binary numbers of the form $2^j(2^k - 1)$. (Simple procedures are available for dividing by such numbers.)

5. Start the q and r sequences out with q_0 and q_1 large enough so that the inequality in exercise 3 is valid. Then we will find in the formulas like those preceding Theorem C that we have $\eta_1 \to 0$ and $\eta_2 = (1+1/(2r_k))2^{1+\sqrt{2Q_k}-\sqrt{2Q_{k+1}}}(Q_k/Q_{k+1})$. The factor $Q_k/Q_{k+1} \to 1$ as $k \to \infty$, so we can ignore it if we want to show that $\eta_2 < 1-\epsilon$ for all large k. Now $\sqrt{2Q_{k+1}} = \sqrt{2Q_k + 2\lceil\sqrt{2Q_k}\rceil + 2} \geq \sqrt{(2Q_k + 2\sqrt{2Q_k} + 1) + 1} \geq \sqrt{2Q_k} + 1 + 1/(3R_k)$. Hence $\eta_2 \leq (1 + 1/(2r_k))2^{-1/(3R_k)}$, and $\lg \eta_2 < 0$ for large enough k.

Note: Algorithm C can also be modified to define a sequence $q_0, q_1, \ldots$ of a similar type that is based on n, so that $n \approx q_k + q_{k+1}$ after step C1. This modification leads to the estimate (19).

6. Any common divisor of $6q + d_1$ and $6q + d_2$ must also divide their difference $d_2 - d_1$. The $\binom{6}{2}$ differences are 2, 3, 4, 6, 8, 1, 2, 4, 6, 1, 3, 5, 2, 4, 2, so we must only show that at most one of the given numbers is divisible by each of the primes 2, 3, 5. Clearly only $6q + 2$ is even, and only $6q + 3$ is a multiple of 3; and there is at most one multiple of 5, since $q_k \not\equiv 3$ (modulo 5).

7. Let $p_{k-1} < N \leq p_k$. We have $t_k \leq 6t_{k-1} + ck3^k$ for some constant c; so $t_k/6^k \leq t_{k-1}/6^{k-1} + ck/2^k \leq t_0 + c\sum_{j\geq 1}(j/2^j) = M$. Thus $t_k \leq M \cdot 6^k = O(p_k^{\log_3 6})$.

8. Let 2^k be the smallest power of 2 that exceeds $2K$. Set $a_t \leftarrow \omega^{-t^2/2}u_t$ and $b_t \leftarrow \omega^{(2K-2-t)^2/2}$, where $u_t = 0$ for $t \geq K$. We want to calculate the convolutions $c_r = \sum_{0\leq j\leq r} a_j b_{r-j}$ for $r = 2K - 2 - s$, when $0 \leq s < K$. The convolutions can be found by using three fast Fourier transformations of order 2^k, as in the text's multiplication procedure. [Note that this device works for any complex number ω, not necessarily a root of unity. See L. I. Bluestein, *Northeast Electronics Res. and Eng. Meeting Record* **10** (1968), 218–219.]

9. $\tilde{u}_s = \hat{u}_{(qs) \bmod K}$. In particular, if $q = -1$ we get $\hat{u}_{(-r) \bmod K}$, which avoids shuffling when computing inverse transforms.

10. $A^{[j]}(s_{k-1}, \ldots, s_{k-j}, t_{k-j-1}, \ldots, t_0)$ can be written

$$\sum_{0\leq t_{k-1},\ldots,t_{k-j}\leq 1} \omega^{(s_0\ldots s_{k-1})_2\cdot(t_{k-1}\ldots t_{k-j}0\ldots0)_2} \Big(\sum_{0\leq p<K} \omega^{tp}u_p\Big)\Big(\sum_{0\leq q<K} \omega^{tq}v_q\Big),$$

and this is $\sum_{p,q} u_p v_q S(p,q)$, where $S(p,q) = 0$ or 2^j. We have $S(p,q) = 2^j$ for exactly $2^{2k}/2^j$ values of p and q.

11. An automaton cannot have $z_2 = 1$ until it has $c \geq 2$, and this occurs first for M_j at time $3j - 1$. It follows that M_j cannot have $z_2 z_1 z_0 \neq 000$ until time $3(j-1)$. Furthermore, if M_j has $z_0 \neq 0$ at time t, we cannot change this to $z_0 = 0$ without affecting the output; but the output cannot be affected by this value of z_0 until at least time $t + j - 1$, so we must have $t + j - 1 \leq 2n$. Since the first argument we gave proves that $3(j-1) \leq t$, we must have $4(j-1) \leq 2n$, that is, $j - 1 \leq n/2$, i.e., $j \leq \lfloor n/2 \rfloor + 1$. This is the best possible bound, since the inputs $u = v = 2^n - 1$ require the use of M_j for all $j \leq \lfloor n/2 \rfloor + 1$. (For example, note from Table 1 that M_2 is needed to multiply two-bit numbers, at time 3.)

12. We can "sweep through" K lists of MIX-like instructions, executing the first instruction on each list, in $O(K + (N \log N)^2)$ steps as follows: (1) A radix list sort (Section 5.2.5) will group together all identical instructions, in time $O(K + N)$. (2) Each set of j identical instructions can be performed in $O(\log N)^2 + O(j)$ steps, and there are $O(N^2)$ sets. A bounded number of sweeps will finish all the lists. The remaining details are straightforward; for example, arithmetic operations can be simulated by converting p and q to binary. [*SIAM J. Computing*, to appear.]

13. If it takes $T(n)$ steps to multiply n-bit numbers, we can accomplish m-bit times n-bit multiplication by breaking the n-bit number into $\lceil n/m \rceil$ m-bit groups, using $\lceil n/m \rceil T(m) + O(n + m)$ operations. The results of this section therefore give an estimated running time of $O(n \log m \log\log m)$ on Turing machines, or $O(n \log m)$ on machines with random access to words of bounded size, or $O(n)$ on pointer machines.

SECTION 4.4

1. We compute $(\ldots(a_m b_{m-1} + a_{m-2})b_{m-2} + \cdots + a_1)b_1 + a_0$ by adding and multiplying in the B_j system.

	T.	= 20(cwt.	= 8(st.	= 14(lb.	= 16 oz.)))
Start with zero	0	0	0	0	0
Add 3	0	0	0	0	3
Multiply by 24	0	0	0	4	8
Add 9	0	0	0	5	1
Multiply by 60	0	2	5	9	12
Add 12	0	2	5	10	8
Multiply by 60	8	3	1	0	0
Add 37	8	3	1	2	5

(Addition and multiplication by a constant in a mixed-radix system are readily done using a simple generalization of the usual carry rule; cf. exercise 4.3.1–9.)

2. We compute $\lfloor u/B_0 \rfloor$, $\lfloor \lfloor u/B_0 \rfloor / B_1 \rfloor$, etc., and the remainders are A_0, A_1, etc. The division is done in the b_j system.

	d.	= 24(h.	= 60(m.	= 60 s.))	
Start with u	3	9	12	37	
Divide by 16	0	5	4	32	Remainder = 5
Divide by 14	0	0	21	45	Remainder = 2
Divide by 8	0	0	2	43	Remainder = 1
Divide by 20	0	0	0	8	Remainder = 3
Divide by ∞	0	0	0	0	Remainder = 8

Answer: 8 T. 3 cwt. 1 st. 2 lb. 5 oz.

3. The following procedure due to G. L. Steele Jr. and Jon L. White generalizes Taranto's algorithm for $B = 2$ originally published in *CACM* **2**, 7 (July 1959), 27.

A1. [Initialize.] Set $M \leftarrow 0$, $U_0 \leftarrow 0$.

A2. [Done?] If $u < \epsilon$ or $u > 1 - \epsilon$, go to step A4. (Otherwise no M-place fraction will satisfy the given conditions.)

A3. [Transform.] Set $M \leftarrow M+1$, $U_{-M} \leftarrow \lfloor Bu \rfloor$, $u \leftarrow Bu \bmod 1$, $\epsilon \leftarrow B\epsilon$, and return to A2. (This transformation returns us to essentially the same state we were in before; the remaining problem is to convert u to U with fewest radix-B places so that $|U-u| < \epsilon$. Note, however, that ϵ may now be ≥ 1; in this case we could go immediately to step A4 instead of storing the new value of ϵ.)

A4. [Round.] If $u \geq \frac{1}{2}$, increase U_{-M} by 1. (If $u = \frac{1}{2}$ exactly, another rounding rule such as "increase U_{-M} by 1 only when it is odd" might be preferred.) ∎

Step A4 will never increase U_{-M} from $B-1$ to B; for if $U_{-M} = B-1$ we must have $M > 0$, but no $(M-1)$-place fraction was sufficiently accurate. Steele and White go on to consider floating-point conversions in their paper [to appear].

4. (a) $1/2^k = 5^k/10^k$. (b) Every prime divisor of b divides B.

5. Iff $10^n - 1 \leq c < w$, cf. (3).

7. $\alpha u \leq ux \leq \alpha u + u/w \leq \alpha u + 1$, hence $\lfloor \alpha u \rfloor \leq \lfloor ux \rfloor \leq \lfloor \alpha u + 1 \rfloor$. Furthermore, in the special case cited we have $ux < \alpha u + \alpha$ and $\lfloor \alpha u \rfloor = \lfloor \alpha u + \alpha - \epsilon \rfloor$.

```
8.    ENT1 0
      LDA  U
   1H MUL  =1//10=
   3H STA  TEMP
      MUL  =-10=
      SLAX 5
      ADD  U
      JANN 2F
      LDA  TEMP       (Can occur only on
      DECA 1               the first iteration,
      JMP  3B              by exercise 7.)
   2H STA  ANSWER,1 (May be minus zero.)
      LDA  TEMP
      INC1 1
      JAP  1B          ∎
```

9. If x' is an integer, $x - \epsilon \leq x' \leq x$, then $(1+1/n)x - ((1+1/n)\epsilon + 1 - 1/n) \leq x' + \lfloor x'/n \rfloor \leq (1+1/n)x$. Hence if α is the binary fraction satisfying

$$1/10 - 2^{-35} < \alpha = (.00011001100110011001100110011001)_2 < 1/10,$$

we find that $\alpha u - \epsilon \leq v \leq \alpha u$ at the end of the computation, where

$$\epsilon = \tfrac{7}{8} + (.10001000101010001100100010101001)_2 < \tfrac{3}{2}.$$

Hence $u/10 - 2 < u/10 - (\epsilon + (1/10 - \alpha)u) \leq v \leq \alpha u < u/10$. Since v is an integer, the proof is complete.

10. (a) Shift right one; (b) Extract left bit of each group; (c) Shift result of (b) right two; (d) Shift result of (c) right one, and add to result of (c); (e) Subtract result of (d) from result of (a).

11.
```
  5.7 7 2 1
- 1 0
  4 7.7 2 1
-   9 4
  3 8 3.2 1
-   7 6 6
  3 0 6 6.1
-   6 1 3 2
  2 4 5 2 9
```
Answer: $(24529)_{10}$.

12. First convert the ternary number to nonary (radix 9) notation, then proceed as in octal-to-decimal conversion but without doubling. Decimal to nonary is similar. In the given example, we have

```
  1.7 6 4 7 2 3
- 1
  1 6.6 4 7 2 3
- 1 6
  1 5 0.4 7 2 3
- 1 5 0
  1 3 5 4.7 2 3
- 1 3 5 4
  1 2 1 9 3.2 3
- 1 2 1 9 3
  1 0 9 7 3 9.3
- 1 0 9 7 3 9
    9 8 7 6 5 4
```
Answer: $(987654)_{10}$.

```
    9.8 7 6 5 4
+     9
    1 1 8.7 6 5 4
+   1 1 8
    1 3 1 6.6 5 4
+   1 3 1 6
    1 4 4 8 3.5 4
+   1 4 4 8 3
    1 6 0 4 2 8.4
+   1 6 0 4 2 8
    1 7 6 4 7 2 3
```
Answer: $(1764723)_9$.

13.
```
BUF   ALF  .␣␣␣␣          (Radix point on first line)
      ORIG *+39
START JOV  OFLO           Ensure overflow is off.
      ENT2 -40            Set buffer pointer.
8H    ENT3 10             Set loop counter.
1H    ENT1 m              Begin multiplication routine.
      ENTX 0
2H    STX  CARRY
      ...                 (See exercise 4.3.1–13, with
      J1P  2B                 v = 10^9 and W = U.)
      SLAX 5              rA ← next nine digits.
      CHAR
      STA  BUF+40,2(2:5)  Store next nine digits.
      STX  BUF+41,2
      INC2 2              Increase buffer pointer.
      DEC3 1
      J3P  1B             Repeat ten times.
      OUT  BUF+20,2(PRINTER)
      J2N  8B             Repeat until both lines printed. ∎
```

14. Let $K(n)$ be the number of steps required to convert an n-digit decimal number to binary and at the same time to compute the binary representation of 10^n. Then we have $K(2n) \le 2K(n) + O(M(n))$. *Proof:* Given the number $U = (u_{2n-1} \dots u_0)_{10}$, compute $U_1 = (u_{2n-1} \dots u_n)_{10}$ and $U_0 = (u_{n-1} \dots u_0)_{10}$ and 10^n, in $2K(n)$ steps, then compute $U = 10^n U_1 + U_0$ and $10^{2n} = 10^n \cdot 10^n$ in $O(M(n))$ steps. It follows that $K(2^n) = O(M(2^n) + 2M(2^{n-1}) + 4M(2^{n-2}) + \cdots) = O(nM(2^n))$.

[Similarly, Schönhage has observed that we can convert a $(2^n \lg 10)$-bit number U from binary to decimal, in $O(nM(2^n))$ steps. First form $V = 10^{2^{n-1}}$ in $O(M(2^{n-1}) + M(2^{n-2}) + \cdots) = O(M(2^n))$ steps, then compute $U_0 = (U \bmod V)$ and $U_1 = \lfloor U/V \rfloor$ in $O(M(2^n))$ further steps, then convert U_0 and U_1.]

18. Let $U = \text{round}_B(u, P)$ and $v = \text{round}_b(U, p)$. We may assume that $u \neq 0$, so that $U \neq 0$ and $v \neq 0$. *Case 1, $v < u$:* Determine e and E such that $b^{e-1} < u \le b^e$, $B^{E-1} \le U < B^E$. Then $u \le U + \frac{1}{2}B^{E-P}$ and $U \le u - \frac{1}{2}b^{e-p}$; hence $B^{P-1} \le B^{P-E}U < B^{P-E}u \le b^{p-e}u \le b^p$. *Case 2, $v > u$:* Determine e and E such that $b^{e-1} \le u < b^e$, $B^{E-1} < U \le B^E$. Then $u \ge U - \frac{1}{2}B^{E-P}$ and $U \ge u + \frac{1}{2}b^{e-p}$; hence $B^{P-1} \le B^{P-E}(U - B^{E-P}) < B^{P-E}u \le b^{p-e}u < b^p$. Thus we have proved that $B^{P-1} < b^p$ whenever $v \neq u$.

Conversely, if $B^{P-1} < b^p$, the above proof suggests that the most likely example for which $u \neq v$ will occur when u is a power of b and at the same time it is close to a power of B. We have $B^{P-1}b^p < B^{P-1}b^p + \frac{1}{2}b^p - \frac{1}{2}B^{P-1} - \frac{1}{4} = (B^{P-1} + \frac{1}{2})(b^p - \frac{1}{2})$; hence $1 < \alpha = 1/(1 - \frac{1}{2}b^{-p}) < 1 + \frac{1}{2}B^{1-P} = \beta$. There are integers e and E such that $\log_B \alpha < e \log_B b - E < \log_B \beta$, since Weyl's theorem (exercise 3.5–22) implies that there is an integer e with $0 < \log_B \alpha < (e \log_B b) \bmod 1 < \log_B \beta < 1$ when $\log_B b$ is irrational. Hence $\alpha < b^e/B^E < \beta$, for some e and E. (Such e and E may also be found by applying the theory of continued fractions, see Section 4.5.3.) Now we have $\text{round}_B(b^e, P) = B^E$, and $\text{round}_b(B^E, p) < b^e$. [*CACM* **11** (1968), 47–50; *Proc. Amer. Math. Soc.* **19** (1968), 716–723.]

19. $m_1 = (\texttt{F0F0F0F0})_{16}$, $c_1 = 1 - 10/16$ makes $U = ((u_7u_6)_{10} \dots (u_1u_0)_{10})_{256}$; then $m_2 = (\texttt{FF00FF00})_{16}$, $c_2 = 1 - 10^2/16^2$ makes $U = ((u_7u_6u_5u_4)_{10}(u_3u_2u_1u_0)_{10})_{65536}$; and $m_3 = (\texttt{FFFF0000})_{16}$, $c_3 = 1 - 10^4/16^4$ finishes the job. (Cf. exercise 14. This technique is due to Roy A. Keir, circa 1958.)

SECTION 4.5.1

1. Test whether or not $uv' < u'v$, since the denominators are positive.

2. If $c > 1$ divides both u/d and v/d, then cd divides both u and v.

3. Let p be prime. If p^e is a divisor of uv and $u'v'$ for $e \ge 1$, then either $p^e \setminus u$ and $p^e \setminus v'$ or $p^e \setminus u'$ and $p^e \setminus v$; hence $p^e \setminus \gcd(u, v')\gcd(u', v)$. The converse follows by reversing the argument.

4. Let $d_1 = gcd(u, v)$, $d_2 = \gcd(u', v')$; the answer is $w = (u/d_1)(v'/d_2)\text{sign}(v)$, $w' = |(u'/d_2)(v/d_1)|$, with a "divide by zero" error message if $v = 0$.

5. $d_1 = 10$, $t = 17 \cdot 7 - 27 \cdot 12 = -205$, $d_2 = 5$, $w = -41$, $w' = 168$.

6. Let $u'' = u'/d_1$, $v'' = v'/d_1$; our goal is to show that $\gcd(uv'' + u''v, d_1) = \gcd(uv'' + u''v, d_1u''v'')$. If p is a prime that divides u'', then p does not divide u or v'', so p does not divide $uv'' + u''v$. A similar argument holds for prime divisors of v'', so no prime divisors of $u''v''$ affect the given gcd.

7. $(N-1)^2+(N-2)^2 = 2N^2-(6N-5)$. If the inputs are n-bit binary numbers, $2n+1$ bits may be necessary to represent t.

8. For multiplication and division these quantities obey the rules $x/0 = \text{sign}(x)\infty$, $(\pm\infty)\times x = x\times(\pm\infty) = (\pm\infty)/x = \pm\text{sign}(x)\infty$, $x/(\pm\infty) = 0$, provided that x is finite and nonzero, without change to the algorithms described. Furthermore, the algorithms can readily be modified so that $0/0 = 0\times(\pm\infty) = (\pm\infty)\times 0 =$ "(0/0)", where the latter is a representation of "undefined"; and so that if either operand is "undefined" the result will be "undefined" also.

Since the multiplication and division subroutines can yield these fairly natural rules of "extended arithmetic," it is sometimes worthwhile to modify the addition and subtraction operations so that they satisfy the rules $x\pm\infty = \pm\infty$, $x\pm(-\infty) = \mp\infty$, for x finite; $(\pm\infty)+(\pm\infty) = \pm\infty-(\mp\infty) = \pm\infty$; furthermore $(\pm\infty)+(\mp\infty) = (\pm\infty)-(\pm\infty) = (0/0)$; and if either or both operands are (0/0), the result should also be (0/0). Equality tests and comparisons may be treated in a similar manner.

The above remarks are independent of "overflow" indications. If ∞ is being used to suggest overflow, it is incorrect to let $1/\infty$ be equal to zero, lest inaccurate results be regarded as true answers. It is far better to represent overflow by (0/0), and to adhere to the convention that the result of any operation is undefined if at least one of the inputs is undefined. This type of overflow indication has the advantage that final results of an extended calculation reveal exactly which answers are defined and which are not.

9. If $u/u' \neq v/v'$, then

$$1 \le |uv'-u'v| = u'v'|(u/u')-(v/v')| < |2^{2n}(u/u')-2^{2n}(v/v')|;$$

two quantities differing by more than unity cannot have the same "floor." (In other words, the first $2n$ bits to the right of the binary point are enough to characterize the value of the fraction, when there are n-bit denominators. We cannot improve this to $2n-1$ bits, for if $n=4$ we have $\frac{1}{13} = (.00010011\ldots)_2$, $\frac{1}{14} = (.00010010\ldots)_2$.)

11. To divide by $(v+v'\sqrt{5})/v''$, when v and v' are not both zero, multiply by the reciprocal, $(v-v'\sqrt{5})v''/(v^2-5v'^2)$, and reduce to lowest terms.

12. One idea is to limit numerator and denominator to a total of 27 bits, where we need only store 26 of these bits (since the leading bit of the denominator can be assumed 1). This leaves room for a sign and five bits to indicate the denominator size. Another idea is to use 28 bits for numerator and denominator, which are to have a total of at most seven hexadecimal digits, together with a sign and a 3-bit field to indicate the number of hexadecimal digits in the denominator.

[Using the formulas in the next exercise, the first alternative leads to exactly 2140040119 finite representable numbers, while the second leads to 1830986459. The first alternative is preferable because it represents more values, and because it is 'cleaner' and makes smoother transitions between ranges.]

13. The number of multiples of n in the interval $(a,b]$ is $\lfloor n/a\rfloor - \lfloor a/n\rfloor$. Hence, by inclusion and exclusion, the answer to this problem is $S_0 - S_1 + S_2 - \cdots$, where S_k is $\sum\lfloor M/P\rfloor(\lfloor N_2/P\rfloor - \lfloor N_1/P\rfloor)$, summed over all products P of k distinct primes.

SECTION 4.5.2

1. Substitute min, max, $+$ consistently for gcd, lcm, $\times$, respectively.

2. For prime p, let $u_p, v_{1p}, \ldots, v_{np}$ be the exponents of p in the canonical factorizations of $u, v_1, \ldots, v_n$. By hypothesis, $u_p \le v_{1p} + \cdots + v_{np}$. We must show that $u_p \le \min(u_p, v_{1p}) + \cdots + \min(u_p, v_{np})$, and this is certainly true if u_p is greater than or equal to each v_{jp}, or if u_p is less than some v_{jp}.

3. *Solution 1:* A one-to-one correspondence is obtained if we set $u = \gcd(d, n)$ and $v = n^2/\operatorname{lcm}(d, n)$ for each divisor d of n^2. *Solution 2:* If $n = p_1^{e_1} \ldots p_r^{e_r}$, the number in each case is $(2e_1 + 1) \ldots (2e_r + 1)$.

4. See exercise 3.2.1.2–15(a).

5. Shift u and v right until neither is a multiple of 3, remembering the proper power of 3 that will appear in the gcd. Each subsequent iteration sets $t \leftarrow u + v$ or $t \leftarrow u - v$ (whichever is a multiple of 3), shifts t right until it is not a multiple of 3, then replaces $\max(u, v)$ by the result.

u	v	t
13634	24140	10506, 3502;
13634	3502	17136, 5712, 1904;
1904	3502	5406, 1802;
1904	1802	102, 34;
34	1802	1836, 612, 204, 68;
34	68	102, 34;
34	34	0.

The evidence that $\gcd(40902, 24140) = 34$ is now overwhelming.

6. The probability that both u and v are even is $\frac{1}{4}$; the probability that both are multiples of four is $\frac{1}{16}$; etc. Thus A has the distribution given by the generating function

$$\frac{3}{4} + \frac{3}{16}z + \frac{3}{64}z^2 + \cdots = \frac{3/4}{1 - z/4}.$$

The mean is $\frac{1}{3}$, and the standard deviation is $\sqrt{\frac{2}{9} + \frac{1}{3} - \frac{1}{9}} = \frac{2}{3}$. If u and v are independently and uniformly distributed with $1 \le u, v < 2^N$, then some small correction terms are needed; the mean is then actually

$$(2^N - 1)^{-2} \sum_{1 \le k \le N} (2^{N-k} - 1)^2 = \tfrac{1}{3} - \tfrac{4}{3}(2^N - 1)^{-1} + N(2^N - 1)^{-2}.$$

7. When u, v are not both even, each of the cases (even, odd), (odd, even), (odd, odd) is equally probable, and $B = 1, 0, 0$ in these cases. Hence $B = \frac{1}{3}$ on the average. Actually, as in exercise 6, a small correction could be given to be strictly accurate when $1 \le u, v < 2^N$; the probability that $B = 1$ is actually

$$(2^N - 1)^{-2} \sum_{1 \le k \le N} (2^{N-k} - 1)2^{N-k} = \tfrac{1}{3} - \tfrac{1}{3}(2^N - 1)^{-1}.$$

8. The quantity E is the number of subtraction cycles in which $u > v$, plus one if u is odd after step B1. If we change the inputs from (u, v) to (v, u), the value of C stays unchanged, while E becomes $C - E$ or $C - E - 1$; the latter case occurs iff u and v are both odd after step B1, and this has probability $\frac{1}{3} + \frac{2}{3}/(2^N - 1)$. Hence

$$E_{\text{ave}} = C_{\text{ave}} - E_{\text{ave}} - \tfrac{1}{3} - \tfrac{2}{3}/(2^N - 1).$$

9. The binary algorithm first gets to B6 with $u = 1963$, $v = 1359$; then $t \leftarrow 604, 302, 151$, etc. The gcd is 302. Using Algorithm X we find that $2 \cdot 31408 - 23 \cdot 2718 = 302$.

10. (a) Two integers are relatively prime iff they are not both divisible by any prime number. (b) Rearrangement of the sum in (a), in terms of the denominators $k = p_1 \ldots p_r$. (Note that each of the sums in (a) and (b) is actually finite.) (c) Since $(n/k)^2 - \lfloor n/k \rfloor^2 = O(n/k)$, we have $q_n - \sum_{1 \le k \le n} \mu(k)(n/k)^2 = \sum_{1 \le k \le n} O(n/k) = O(nH_n)$. Furthermore $\sum_{k>n} (n/k)^2 = O(n)$. (d) $\sum_{d \backslash n} \mu(d) = \delta_{1n}$. [In fact, we have the more general result

$$\sum_{d \backslash n} \mu(d) \Big(\frac{n}{d}\Big)^s = n^s - \sum \Big(\frac{n}{p}\Big)^s + \sum \Big(\frac{n}{pq}\Big)^s - \cdots,$$

as in part (b), where the sums on the right are over the prime divisors of n, and this is equal to $n^s(1 - 1/p_1^s) \ldots (1 - 1/p_r^s)$ if $n = p_1^{e_1} \ldots p_r^{e_r}$.]

Notes: Similarly, we find that a set of k integers is relatively prime with probability $1/(\sum_{n \ge 1} 1/n^k)$. This proof of Theorem D is due to F. Mertens, *J. für die reine und angew. Math.* **77** (1874), 289–291. The technique actually gives a much stronger result, namely that $6\pi^{-2}mn + O(n \log m)$ pairs of integers $u \in [f(m), f(m) + m)$, $v \in [g(n), g(n) + n)$ are relatively prime, for arbitrary f and g, when $m \le n$.

11. (a) $6/\pi^2$ times $1 + \frac{1}{4} + \frac{1}{9}$, namely $49/(6\pi^2) \approx .82746$. (b) $6/\pi^2$ times $1/1 + 2/4 + 3/9 + \cdots$, namely ∞. (This is true in spite of the result of exercise 12, and in spite of the fact that the average value of $\ln \gcd(u, v)$ is a small, finite number.)

12. Let $\sigma(n)$ be the number of positive divisors of n. The answer is

$$\sum_{k \ge 1} \sigma(k) \cdot \frac{6}{\pi^2 k^2} = \frac{6}{\pi^2} \Big(\sum_{k \ge 1} \frac{1}{k^2} \Big)^2 = \frac{\pi^2}{6}.$$

[Thus, the average is *less* than 2, although there are always at least two common divisors when u and v are not relatively prime.]

13. $1 + \frac{1}{9} + \frac{1}{25} + \cdots = 1 + \frac{1}{4} + \frac{1}{9} + \cdots - \frac{1}{4}(1 + \frac{1}{4} + \frac{1}{9} + \cdots)$.

14. $v_1 = \pm v/u_3$, $v_2 = \mp u/u_3$ (the sign depends on whether the number of iterations is even or odd). This follows from the fact that v_1 and v_2 are relatively prime to each other (throughout the algorithm), and that $v_1 u = -v_2 v$. [Hence $v_1 u = \operatorname{lcm}(u, v)$ at the close of the algorithm, but this is not an especially efficient way to compute the least common multiple. For a generalization, see exercise 4.6.1–18.]

G. E. Collins has observed that $|u_1| \le \frac{1}{2} v/u_3$, $|u_2| \le \frac{1}{2} u/u_3$, at the termination of Algorithm X, except in certain trivial cases, since the final value of q is usually ≥ 2. This bounds the size of $|u_1|$ and $|u_2|$ throughout the execution of the algorithm.

15. Apply Algorithm X to v and m, thus obtaining a value x such that $xv \equiv 1$ (modulo m). (This can be done by simplifying Algorithm X so that u_2, v_2, and t_2 are not computed, since they are never used in the answer.) Then set $w \leftarrow ux \bmod m$. [It follows, as in exercise 30, that this process requires $O(n^2)$ units of time, when it is applied to large n-bit numbers. An alternative to Algorithm X appears in exercise 35.]

16. (a) Set $t_1 = x + 2y + 3z$; then $3t_1 + y + 2z = 1$, $5t_1 - 3y - 20z = 3$. Eliminate y, then $14t_1 - 14z = 6$: No solution. (b) This time $14t_1 - 14z = 0$. Divide by 14, eliminate t_1; the general solution is $x = 8z - 2$, $y = 1 - 5z$, z arbitrary.

17. Let u_1, u_2, u_3, v_1, v_2, v_3 be multiprecision variables, in addition to u and v. The extended algorithm will act the same on u_3 and v_3 as Algorithm L does on u and v. New multiprecision operations are to set $t \leftarrow Au_j$, $t \leftarrow t + Bv_j$, $w \leftarrow Cu_j$, $w \leftarrow w + Dv_j$, $u_j \leftarrow t$, $v_j \leftarrow w$ for all j, in step L4; also if $B = 0$ in that step to set $t \leftarrow u_j - qv_j$, $u_j \leftarrow v_j$, $v_j \leftarrow t$ for all j and for $q = \lfloor u_3/v_3 \rfloor$. A similar modification is made to step L1 if v_3 is small. The inner loop (steps L2 and L3) is unchanged.

18. If $mn = 0$, the probabilities of the lattice-point model in the test are exact, so we may assume that $m \geq n > 0$. *Valida vi*, the following values have been obtained:

Case 1, $m = n$. From (n, n) we go to $(n-t, n)$ with probability $t/2^t - 5/2^{t+1} + 3/2^{2t}$, for $2 \leq t < n$. (These values are $\frac{1}{16}$, $\frac{7}{64}$, $\frac{27}{256}$, ...) To $(0, n)$ the probability is $n/2^{n-1} - 1/2^n + 1/2^{2n-2}$. To (n, k) the probability is the same as to (k, n). The algorithm terminates with probability $1/2^{n-1}$.

Case 2, $m = n+1$. From $(n+1, n)$ we get to (n, n) with probability $\frac{1}{8}$ when $n > 1$, or 0 when $n = 1$; to $(n-t, n)$ with probability $11/2^{t+3} - 3/2^{2t+1}$, for $1 \leq t < n-1$. (These values are $\frac{5}{16}$, $\frac{1}{4}$, ...) We get to $(1, n)$ with probability $5/2^{n+1} - 3/2^{2n-1}$, for $n > 1$; to $(0, n)$ with probability $3/2^n - 1/2^{2n-1}$.

Case 3, $m \geq n + 2$. The probabilities are given by the following table:

$(m-1, n)$:	$1/2 - 3/2^{m-n+2} - \delta_{n1}/2^{m+1}$;
$(m-t, n)$:	$1/2^t + 3/2^{m-n+t+1}$, $\quad 1 < t < n$;
$(m-n, n)$:	$1/2^n + 1/2^m$, $\quad n > 1$;
$(m-n-1, n)$:	$1/2^{n+1} + 1/2^{m-1}$;
$(m-n-t, n)$:	$1/2^{n+t}$, $\quad 1 < t < m-n$;
$(0, n)$:	$1/2^{m-1}$.

Note: Although these exact probabilities will certainly improve on the lattice-point model considered in the text, they lead to recurrence relations of much greater complexity; and they will not provide the true behavior of Algorithm B, since for example the probability that $\gcd(u, v) = 5$ is different from the probability that $\gcd(u, v) = 7$.

19. $A_{n+1} = a + \sum_{1 \leq k \leq n} 2^{-k} A_{(n+1)(n-k)} + 2^{-n} b = a + \sum_{1 \leq k \leq n} 2^{-k} A_{n(n-k)} + \frac{1}{2}c(1 - 2^{-n}) + 2^{-n} b = a + \frac{1}{2}A_{n(n-1)} + \frac{1}{2}(A_n - a) + \frac{1}{2}c(1 - 2^{-n})$; now substitute for $A_{n(n-1)}$ from (36).

20. The paths described in the hint have the same probability, but the subsequent termination of the algorithm has a different probability; thus $\lambda = k+1$ with probability 2^{-k} times the probability that $\lambda = 1$. Let the latter probability be p. We know from the text that $\lambda = 0$ with probability $\approx \frac{3}{5}$; hence $\frac{2}{5} \approx p(1 + \frac{1}{2} + \frac{1}{4} + \cdots) = 2p$. The average is $p(1 + \frac{2}{2} + \frac{3}{4} + \frac{4}{8} + \cdots) = p(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots)^2 = 4p$. [The exact probability that $\lambda = 1$ is $\frac{1}{5} - \frac{6}{5}(-\frac{1}{4})^n$ if $m > n \geq 1$, $\frac{1}{5} - \frac{16}{5}(-\frac{1}{4})^n$ if $m = n \geq 2$.]

21. Show that for fixed v and for $2^m < u < 2^{m+1}$, when m is large, each subtraction-shift cycle of the algorithm reduces $\lfloor \lg u \rfloor$ by two, on the average.

22. Exactly $(N-m)2^{m-1+\delta_{m0}}$ integers u in the range $1 \leq u < 2^N$ have $\lfloor \lg u \rfloor = m$, after u has been shifted right until it is odd.

23. The first sum is $2^{2N-2}\sum_{0\le m<n<N} mn2^{-m-n}((\alpha+\beta)N+\gamma-\alpha m-\beta n)$. Since

$$\sum_{0\le m<n} m2^{-m} = 2-(n+1)2^{1-n}$$

and

$$\sum_{0\le m<n} m(m-1)2^{-m} = 4-(n^2+n+2)2^{1-n},$$

the sum on m is

$$2^{2N-2}\sum_{0\le n<N} n2^{-n}((\gamma-\alpha+(\alpha+\beta)N)(2-(n+1)2^{1-n}) \\ -\alpha(4-(n^2+n+2)2^{1-n})-\beta n)$$

$$= 2^{2N-2}((\alpha+\beta)N\sum_{0\le n<N} n2^{-n}(2-(n+1)2^{1-n})+O(1)).$$

Thus the coefficient of $(\alpha+\beta)N$ in the answer is found to be $2^{-2}(4-(\frac{4}{3})^3)=\frac{11}{27}$. A similar argument applies to the other sum.

Note: The exact value of the sums may be obtained after some tedious calculation by means of the general formula

$$\sum_{0\le k<n} k^{\underline{m}} z^k = \frac{m!\,z^m}{(1-z)^{m+1}} - \sum_{0\le k\le m} \frac{m^{\underline{k}} n^{\underline{m-k}} z^{n+k}}{(1-z)^{k+1}},$$

which follows from summation by parts.

24. Solving a recurrence similar to (34), we find that the number of times is A_{mn}, where $A_{00}=1$, $A_{0n}=(n+3)/2$, $A_{nn}=\frac{8}{5}-(3n+13)/(9\cdot 2^n)+\frac{128}{45}(-\frac{1}{4})^n$ if $n\ge 1$, $A_{mn}=\frac{8}{5}-2/(3\cdot 2^n)+\frac{16}{15}(-\frac{1}{4})^n$ if $m>n\ge 1$. Since the condition $u=1$ or $v=1$ is therefore satisfied only about 1.6 times in an average run, it is not worth making the suggested test each time step B5 is performed. (Of course the lattice model is not completely accurate, but it seems reasonable to believe that it is not too inaccurate for this application.)

25. (a) $F_{n+1}(x)=\sum_{d\ge 1}2^{-d}$ probability that ($X_n<1$ and $2^d/(X_n^{-1}-1)<x$ or $X_n>1$ and $(X_n-1)/2^d<x$) $=\sum_{d\ge 1}2^{-d}(F_n(1/(1+2^dx^{-1}))+F_n(1+2^dx)-F_n(1))$. (b) $G_{n+1}(x)=1-\sum_{d\ge 1}2^{-d}(G_n(1/(1+2^dx))-G_n(1/(1+2^dx^{-1})))$. (c) $H_n(x)=\sum_{d\ge 1}2^{-d}$ probability that ($Y_n\le x$ and $(1-Y_n)/2^d\le x$) can be transformed into $\sum_{d\ge 1}2^{-d}\max(0,G_n(x)-G_n(1-2^dx))$.

Starting with $G_0(x)=x$ we get rapid convergence to a limiting distribution where

$$(G(.1),\ldots,G(.9))=(.2750,.4346,.5544,.6507,.7310,.7995,.8590,.9114,.9581).$$

The expected value of $\ln(\max(u_n,v_n)/\max(u_{n+1},v_{n+1}))$ is $\int_0^1 H_n(t)\,dt/t$, and Brent has shown that this can be written

$$\int_{1/3}^1 \frac{G_n(t)}{t}\,dt-\int_0^{1/3}\frac{G_n(t)}{1-t}\,dt+\sum_{k\ge 1}2^{-k}\int_{1/(1+2^{k+1})}^{1/(1+2^k)}\frac{G_n(t)}{t(1-t)}\,dt.$$

26. By induction, the length is $m+\lfloor n/2\rfloor$ when $m \geq n$, except that when $m = n = 1$ there is *no* path to $(0,0)$.

27. Let $a_n = (2^n - (-1)^n)/3$; then $a_0, a_1, a_2, \ldots = 0, 1, 1, 3, 5, 11, 21, \ldots$. (This sequence of numbers has an interesting pattern of zeros and ones in its binary representation. Note that $a_n = a_{n-1} + 2a_{n-2}$, and $a_n + a_{n+1} = 2^n$.) For $m > n$, let $u = 2^{m+1} - a_{n+2}$, $v = a_{n+2}$. For $m = n > 0$, let $u = a_{n+2}$ and $v = 2a_{n+1}$, or $u = 2a_{n+1}$ and $v = a_{n+2}$ (depending on which is larger). Another example for the case $m = n > 0$ is $u = 2^{n+1} - 1$, $v = 2^{n+1} - 2$; this choice takes more shifts, and gives $C = n + 1$, $D = 2n$, $E = 1$.

28. This is a problem where it appears to be necessary to prove *more* than was asked just to prove what was asked. Let us prove the following: *If u and v are positive integers, Algorithm B does $\leq 1 + \lfloor \lg \max(u,v) \rfloor$ subtraction steps; and if equality holds, then $\lfloor \lg(u+v) \rfloor > \lfloor \lg \max(u,v) \rfloor$.*

For convenience, let us assume that $u \geq v$; let $m = \lfloor \lg u \rfloor$, $n = \lfloor \lg v \rfloor$; and let us use the "lattice-point" terminology, saying that we are "at point (m,n)." The proof is by induction on $m + n$.

Case 1, $m = n$. Clearly, $\lfloor \lg(u+v) \rfloor > \lfloor \lg u \rfloor$ in this case. If $u = v$ the result is trivial; otherwise the next subtraction-shift cycle takes us to a point $(m-k, m)$. By induction, at most $m+1$ further subtraction steps will be required; but if $m+1$ more *are* needed, we have $\lfloor \lg((u-v)2^{-r} + v) \rfloor > \lfloor \lg v \rfloor$, where $r \geq 1$ is the number of right shifts that were made. This is impossible, since $(u-v)2^{-r} + v < (u-v) + v = u$. So at most m further steps are needed.

Case 2, $m > n$. The next subtraction step takes us to $(m-k, n)$, and at most $1 + \max(m-k, n) \leq m$ further steps will be required. Now if m further steps *are* required, then u has been replaced by $u' = (u-v)2^{-r}$ for some $r \geq 1$. By induction, $\lfloor \lg(u'+v) \rfloor \geq m$; hence

$$\lfloor \lg(u+v) \rfloor = \lfloor \lg 2((u-v)/2 + v) \rfloor \geq \lfloor \lg 2(u'+v) \rfloor \geq m + 1 > \lfloor \lg u \rfloor.$$

29. Subtract the kth column from the $2k$th, $3k$th, $4k$th, etc., for $k = 1, 2, 3, \ldots$. The result is a triangular matrix with x_k on the diagonal in column k, where $m = \sum_{d \backslash m} x_d$. It follows that $x_m = \varphi(m)$, so the determinant is $\varphi(1)\varphi(2)\ldots\varphi(n)$.

[In general, "Smith's determinant," in which the (i,j) element is $f(\gcd(i,j))$ for an arbitrary function f, is equal to $\prod_{1 \leq m \leq n} \sum_{d \backslash m} \mu(m/d) f(d)$, by the same argument. See L. E. Dickson, *History of the Theory of Numbers* **1** (New York: Chelsea, 1952), 122–123.]

30. To determine A and r such that $u = Av + r$, $0 \leq r < v$, using ordinary long division, takes $O((1 + \log A)(\log u))$ units of time. If the quotients during the algorithm are $A_1, A_2, \ldots, A_m$, then $A_1 A_2 \ldots A_m \leq u$, so $\log A_1 + \cdots + \log A_m \leq \log u$. Also $m = O(\log u)$.

31. In general, since $(a^u - 1) \bmod (a^v - 1) = a^{u \bmod v} - 1$ (cf. Eq. 4.3.2–19), we find that $\gcd(a^m - 1, a^n - 1) = a^{\gcd(m,n)} - 1$ for all positive integers a.

32. Yes, to $O(n(\log n)^2(\log\log n))$, even if we also need to compute the sequence of partial quotients that would be computed by Euclid's algorithm; see A. Schönhage, *Acta Informatica* **1** (1971), 139–144. [But Algorithm L is better in practice unless n is extremely large.]

34. Keep track of the most significant and least significant words of the operands (the most significant is used to guess the sign of t and the least significant is to determine the amount of right shift), while building a 2×2 matrix A of single-precision integers such that $A\binom{u}{v} = \binom{u'w}{v'w}$, where w is the computer word size and where u' and v' are smaller than u and v. (Instead of dividing the simulated odd operand by 2, multiply the other one by 2, until obtaining multiples of w after exactly $\lg w$ shifts.) Experiments show this algorithm running four times as fast as Algorithm L, on at least one computer.

35. (Solution by Michael Penk.)

Y1. [Find power of 2.] Same as step B1.

Y2. [Initialize.] Set $(u_1, u_2, u_3) \leftarrow (1, 0, u)$ and $(v_1, v_2, v_3) \leftarrow (v, 1 - u, v)$. If u is odd, set $(t_1, t_2, t_3) \leftarrow (0, -1, -v)$ and go to Y4. Otherwise set $(t_1, t_2, t_3) \leftarrow (1, 0, u)$.

Y3. [Halve t_3.] If t_1 and t_2 are both even, set $(t_1, t_2, t_3) \leftarrow (t_1, t_2, t_3)/2$; otherwise set $(t_1, t_2, t_3) \leftarrow (t_1 + v, t_2 - u, t_3)/2$. (In the latter case, $t_1 + v$ and $t_2 - u$ will both be even.)

Y4. [Is t_3 even?] If t_3 is even, go back to Y3.

Y5. [Reset $\max(u_3, v_3)$.] If t_3 is positive, set $(u_1, u_2, u_3) \leftarrow (t_1, t_2, t_3)$; otherwise set $(v_1, v_2, v_3) \leftarrow (v - t_1, -u - t_2, -t_3)$.

Y6. [Subtract.] Set $(t_1, t_2, t_3) \leftarrow (u_1, u_2, u_3) - (v_1, v_2, v_3)$. Then if $t_1 < 0$, set $(t_1, t_2) \leftarrow (t_1 + v, t_2 - u)$. If $t_3 \neq 0$, go back to B3. Otherwise the algorithm terminates with $(u_1, u_2, u_3 \cdot 2^k)$ as the output. ∎

It is clear that the relations in (16) are preserved, and that $0 \le u_1, v_1, t_1 \le v$, $0 \ge u_2, v_2, t_2 \ge -u$ after each of steps Y2–Y6. If u is odd after step Y2, then step Y3 can be simplified, since t_1 and t_2 are both even iff t_2 is even; similarly, if v is odd, then t_1 and t_2 are both even iff t_1 is even. Thus, as in Algorithm X, it is possible to suppress all calculations involving u_2, v_2, and t_2, provided that v is odd after step Y2. This condition is often known in advance (e.g., when v is prime and we are trying to compute u^{-1} modulo v).

SECTION 4.5.3

1. The running time is about $19.02T + 6$, just a trifle slower than Program 4.5.2A.

2. $\begin{pmatrix} Q_n(x_1, x_2, \ldots, x_{n-1}, x_n) & Q_{n-1}(x_1, x_2, \ldots, x_{n-1}) \\ Q_{n-1}(x_2, \ldots, x_{n-1}, x_n) & Q_{n-2}(x_2, \ldots, x_{n-1}) \end{pmatrix}$.

3. $Q_n(x_1, \ldots, x_n)$.

4. By induction, or by taking the determinant of the matrix product in exercise 2.

5. When the x's are positive, the q's of (9) are positive, and $q_{n+1} > q_{n-1}$; hence (9) is an alternating series of decreasing terms, and it converges iff $q_n q_{n+1} \to \infty$. By induction, if the x's are greater than ϵ, we have $q_n \ge c(1 + \epsilon/2)^n$, where c is chosen small enough to make this inequality valid for $n = 1$ and 2. But if $x_n = 1/2^n$, we have $q_n \le 2 - 1/2^n$.

6. It suffices to prove that $A_1 = B_1$; and from the fact that $0 \le /x_1, \ldots, x_n/ < 1$ whenever $x_1, \ldots, x_n$ are positive integers, we have $B_1 = \lfloor 1/X \rfloor = A_1$.

7. Only $1\,2\ldots n$ and $n\ldots 2\,1$. (The variable x_k appears in exactly $F_k F_{n-k}$ terms; hence x_1 and x_n can only be permuted into x_1 and x_n. If x_1 and x_n are fixed by the permutation, it follows by induction that $x_2, \ldots, x_{n-1}$ are also fixed.)

8. This is equivalent to

$$\frac{Q_{n-2}(A_{n-1},\ldots,A_2) - XQ_{n-1}(A_{n-1},\ldots,A_1)}{Q_{n-1}(A_n,\ldots,A_2) - XQ_n(A_n,\ldots,A_1)} = -\frac{1}{X_n},$$

and by (6) this is equivalent to

$$X = \frac{Q_{n-1}(A_2,\ldots,A_n) + X_n Q_{n-2}(A_2,\ldots,A_{n-1})}{Q_n(A_1,\ldots,A_n) + X_n Q_{n-1}(A_1,\ldots,A_{n-1})}.$$

9. (a) By definition. (b), (d) Prove this when $n = 1$, then apply (a) to get the result for general n. (c) Prove when $n = k + 1$, then apply (a).

10. If $A_0 > 0$, then $B_0 = 0$, $B_1 = A_0$, $B_2 = A_1$, $B_3 = A_2$, $B_4 = A_3$, $B_5 = A_4$, $m = 5$. If $A_0 = 0$, then $B_0 = A_1$, $B_1 = A_2$, $B_2 = A_3$, $B_3 = A_4$, $m = 3$. If $A_0 = -1$ and $A_1 = 1$, then $B_0 = -(A_2 + 2)$, $B_1 = 1$, $B_2 = A_3 - 1$, $B_3 = A_4$, $m = 3$. If $A_0 = -1$ and $A_1 > 1$, then $B_0 = -2$, $B_2 = A_1 - 2$, $B_3 = A_2$, $B_4 = A_3$, $B_5 = A_4$, $m = 5$. If $A_0 < -1$, then $B_0 = -1$, $B_1 = 1$, $B_2 = -A_0 - 2$, $B_3 = 1$, $B_4 = A_1 - 1$, $B_5 = A_2$, $B_6 = A_3$, $B_7 = A_4$. [Actually, the last three cases involve eight subcases; if any of the B's is set to zero, the values should be "collapsed together" by using the rule of exercise 9(c). For example, if $A_0 = -1$, $A_1 = A_3 = 1$, we actually have $B_0 = -(A_2 + 2)$, $B_1 = A_4 + 1$, $m = 1$. Double collapsing occurs when $A_0 = -2$, $A_1 = 1$.]

11. Let $q_n = Q_n(A_1,\ldots,A_n)$, $q'_n = Q_n(B_1,\ldots,B_n)$, $p_n = Q_{n+1}(A_0,\ldots,A_n)$, $p'_n = Q_{n+1}(B_0,\ldots,B_n)$. By (5) and (11) we have $X = (p_m + p_{m-1}X_m)/(q_m + q_{m-1}X_m)$, $Y = (p'_n + p'_{n-1}Y_n)/(q'_n + q'_{n-1}Y_n)$; therefore if $X_m = Y_n$, the stated relation between X and Y holds by (8). Conversely, if $X = (qY + r)/(sY + t)$, $|qt - rs| = 1$, we may assume that $s \geq 0$, and we can show that the partial quotients of X and Y eventually agree, by induction on s. The result is clear when $s = 0$, by exercise 9(d). If $s > 0$, let $q = as + s'$, where $0 \leq s' < s$. Then $X = a + 1/((sY + t)/(s'Y + r - at))$; since $s(r - at) - ts' = sr - tq$, and $s' < s$, we know by induction and exercise 10 that the partial quotients of X and Y eventually agree. [*Note:* The fact that m is always odd in exercise 10 shows, by a close inspection of this proof, that $X_m = Y_n$ if and only if $X = (qY + r)/(sY + t)$, where $qt - rs = (-1)^{m-n}$.]

12. (a) Since $V_n V_{n+1} = D - U_n^2$, we know that $D - U_{n+1}^2$ is a multiple of V_{n+1}; hence by induction $X_n = (\sqrt{D} - U_n)/V_n$, where U_n and V_n are integers. [Note that the identity $V_{n+1} = A_n(U_{n-1} - U_n) + V_{n-1}$ makes it unnecessary to divide when V_{n+1} is being determined.]

(b) Let $Y = (-\sqrt{D} - U)/V$, $Y_n = (-\sqrt{D} - U_n)/V_n$. The stated identity obviously holds by replacing $\sqrt{D}$ by $-\sqrt{D}$ in the proof of (a). We have

$$Y = (p_n/Y_n + p_{n-1})/(q_n/Y_n + q_{n-1}),$$

where p_n and q_n are defined in part (c) of this exercise; hence

$$Y_n = (-q_n/q_{n-1})(Y - p_n/q_n)/(Y - p_{n-1}/q_{n-1}).$$

But by (12), p_{n-1}/q_{n-1} and p_n/q_n are extremely close to X; since $X \neq Y$, $Y - p_n/q_n$ and $Y - p_{n-1}/q_{n-1}$ will have the same sign as $Y - X$ for all large n. This proves that $Y_n < 0$ for all large n; hence $0 < X_n < X_n - Y_n = 2\sqrt{D}/V_n$; V_n must be positive. Also $U_n < \sqrt{D}$, since $X_n > 0$. Hence $V_n < 2\sqrt{D}$, since $V_n \le A_n V_n < \sqrt{D} + U_{n-1}$.

Finally, we want to show that $U_n > 0$. Since $X_n < 1$, we have $U_n > \sqrt{D} - V_n$, so we need only consider the case $V_n > \sqrt{D}$; then $U_n = A_n V_n - U_{n-1} \ge V_n - U_{n-1} > \sqrt{D} - U_{n-1}$, and this is positive as we have already observed.

Note: In the repeating cycle, $\sqrt{D} + U_n = A_n V_n + (\sqrt{D} - U_{n-1}) > V_n$; hence $\lfloor(\sqrt{D} + U_{n+1})/V_{n+1}\rfloor = \lfloor A_{n+1} + V_n/(\sqrt{D} + U_n)\rfloor = A_{n+1} = \lfloor(\sqrt{D} + U_n)/V_{n+1}\rfloor$. In other words A_{n+1} is determined by U_{n+1} and V_{n+1}; we can determine (U_n, V_n) from its successor (U_{n+1}, V_{n+1}) in the period. In fact, when $0 < V_n < \sqrt{D} + U_n$ and $0 < U_n < \sqrt{D}$, the arguments above prove that $0 < V_{n+1} < \sqrt{D} + U_{n+1}$ and $0 < U_{n+1} < \sqrt{D}$; moreover, if the pair (U_{n+1}, V_{n+1}) follows (U', V') with $0 < V' < \sqrt{D} + U'$ and $0 < U' < \sqrt{D}$, then $U' = U_n$ and $V' = V_n$. Hence (U_n, V_n) *is part of the cycle if and only if* $0 < V_n < \sqrt{D} + U_n$ *and* $0 < U_n < \sqrt{D}$.

(c) $$\frac{-V_{n+1}}{V_n} = X_n Y_n = \frac{(q_n X - p_n)(q_n Y - p_n)}{(q_{n-1} X - p_{n-1})(q_{n-1} Y - p_{n-1})}.$$

There is also a companion identity, namely

$$V p_n p_{n-1} + U(p_n q_{n-1} + p_{n-1} q_n) + ((U^2 - D)/V) q_n q_{n-1} = (-1)^n U_n.$$

(d) If $X_n = X_m$ for some $n \neq m$, then X is an irrational number that satisfies the quadratic equation $(q_n X - p_n)/(q_{n-1} X - p_{n-1}) = (q_m X - p_m)/(q_{m-1} X - p_{m-1})$.

14. As in exercise 9, we need only verify the stated identities when c is the last partial quotient, and this verification is trivial. Now Hurwitz's rule gives $2/e = /1, 2, 1, 2, 0, 1, 1, 1, 1, 1, 0, 2, 3, 2, 0, 1, 1, 3, 1, 1, 0, 2, 5, \ldots/$. Taking the reciprocal, collapsing out the zeros as in exercise 9, and taking note of the pattern that appears, we find (cf. exercise 16) that $e/2 = 1 + /2, \overline{2m+1, 3, 1, 2m+1, 1, 3}/$, $m \ge 0$. [*Schriften der phys.-ökon. Gesellschaft zu Königsberg* **32** (1891), 59–62.]

15. (This procedure maintains four integers (A, B, C, D) with the invariant meaning that "our remaining job is to output the continued fraction for $(Ay + B)/(Cy + D)$, where y is the input yet to come.") Initially set $j \leftarrow k \leftarrow 0$, $(A, B, C, D) \leftarrow (a, b, c, d)$; then input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$, one or more times until $C + D$ has the same sign as C. (When $j \ge 1$ and the input has not terminated, we know that $1 < y < \infty$; and when $C + D$ has the same sign as C we know therefore that $(Ay + B)/(Cy + D)$ lies between $(A + B)/(C + D)$ and A/C.) Now comes the general step: If no integer lies strictly between $(A + B)/(C + D)$ and A/C, output $X_k \leftarrow \lfloor A/C\rfloor$, and set $(A, B, C, D) \leftarrow (C, D, A - X_k C, B - X_k D)$, $k \leftarrow k + 1$; otherwise input x_j and set $(A, B, C, D) \leftarrow (Ax_j + B, A, Cx_j + D, C)$, $j \leftarrow j + 1$. The general step is repeated ad infinitum. However, if at any time the *final* x_j is input, the algorithm immediately switches gears: It outputs the continued fraction for $(Ax_j + B)/(Cx_j + D)$, using Euclid's algorithm, and terminates.

The following tableau shows the working for the requested example, where the matrix $\left(\begin{smallmatrix} B & A \\ D & C \end{smallmatrix}\right)$ begins at the upper left corner, then shifts right one on input, down one

on output:

	x_j	-1	5	1	1	1	2	1	2	∞
X_k	39	97	-58	-193						
-2	-25	-62	37	123						
2			16	53						
3			5	17	22					
7			1	2	3	5				
1				3	1	4	5	14		
1					2	1	3	7		
1							2	7	9	25
12							1	0	1	2
2										1
∞										0

M. Mendès France has shown that the number of quotients output per quotient input is asymptotically bounded between $1/r$ and r, where $r = 2\lfloor K(|ad-bc|)/2\rfloor + 1$ and K is the function defined in exercise 38; this bound is best possible. [*Topics in Number Theory*, ed. by P. Turán, *Colloquia Math. Soc. János Bolyai* **13** (1976), 183–194.]

Gosper has also shown that the above algorithm can be generalized to compute the continued fraction for $(axy+bx+cy+d)/(Axy+Bx+Cy+D)$ from those of x and y (in particular, to compute sums and products). [MIT AI Laboratory Memo 239 (Feb. 29, 1972), Hack 101.]

16. It is not difficult to prove by induction that $f_n(z) = z/(2n+1) + O(z^3)$ is an odd function with a convergent power series in a neighborhood of the origin, and that it satisfies the given differential equation. Hence

$$f_0(z) = \mathop{/}z^{-1} + f_1(z)\mathop{/} = \cdots = \mathop{/}z^{-1}, 3z^{-1}, \ldots, (2n+1)z^{-1} + f_{n+1}(z)\mathop{/}.$$

It remains to prove that $\lim_{n\to\infty} \mathop{/}z^{-1}, 3z^{-1}, \ldots, (2n+1)z^{-1}\mathop{/} = f_0(z)$. [Actually Euler, age 24, obtained continued fraction expansions for the considerably more general differential equation $f_n'(z) = az^m + bf_n(z)z^{m-1} + cf_n(z)^2$; but he did not bother to prove convergence, since formal manipulation and intuition were good enough in the eighteenth century.]

There are several ways to prove the desired limiting equation. First, letting $f_n(z) = \sum_k a_{nk} z^k$, we can argue from the equation

$$(2n+1)a_{n1} + (2n+3)a_{n3}z^2 + (2n+5)a_{n5}z^4 + \cdots = 1 - (a_{n1}z + a_{n3}z^3 + a_{n5}z^5 + \cdots)^2$$

that $(-1)^k a_{n(2k+1)}$ is a sum of terms of the form $c_k/(2n+1)^{k+1}(2n+b_{k1})\ldots(2n+b_{kk})$, where the c_k and b_{km} are positive integers independent of n. For example, we have $-a_{n7} = 4/(2n+1)^4(2n+3)(2n+5)(2n+7) + 1/(2n+1)^4(2n+3)^2(2n+7)$. Thus $|a_{(n+1)k}| \le |a_{nk}|$, and $|f_n(z)| \le \tan|z|$ for $|z| < \pi/2$. This uniform bound on $f_n(z)$ makes the convergence proof very simple. Careful study of this argument reveals that the power series for $f_n(z)$ actually converges for $|z| < \pi\sqrt{2n+1}/2$; this is interesting, since it shows that the singularities of $f_n(z)$ get farther and farther away from the

origin as n grows, so the continued fraction actually represents $\tanh z$ *throughout* the complex plane.

Another proof gives further information of a different kind: If we let

$$A_n(z) = n! \sum_{0\le k\le n} \binom{2n-k}{n} z^k/k! = \sum_{k\ge 0} \frac{(n+k)!\, z^{n-k}}{k!\,(n-k)!},$$

then

$$\begin{aligned} A_{n+1}(z) &= \sum_{k\ge 0} \frac{(n+k-1)!\,\big((4n+2)k + (n+1-k)(n-k)\big)}{k!\,(n+1-k)!}\, z^{n+1-k} \\ &= (4n+2)A_n(z) + z^2 A_{n-1}(z). \end{aligned}$$

It follows, by induction, that

$$\begin{aligned} Q_n\Big(\frac{1}{z}, \frac{3}{z}, \ldots, \frac{2n-1}{z}\Big) &= \frac{A_n(2z) + A_n(-2z)}{2^{n+1} z^n}, \\ Q_{n-1}\Big(\frac{3}{z}, \ldots, \frac{2n-1}{z}\Big) &= \frac{A_n(2z) - A_n(-2z)}{2^{n+1} z^n}. \end{aligned}$$

Hence

$$\big/ z^{-1}, 3z^{-1}, \ldots, (2n-1)z^{-1} \big/ = \frac{A_n(2z) - A_n(-2z)}{A_n(2z) + A_n(-2z)},$$

and we want to show that this ratio approaches $\tanh z$. By Eqs. 1.2.9–11 and 1.2.6–24,

$$e^z A_n(-z) = n! \sum_{m\ge 0} z^m \left(\sum_{0\le k\le n} \binom{m}{k}\binom{2n-k}{n}(-1)^k \right) = \sum_{m\ge 0} \binom{2n-m}{n} z^m \frac{n!}{m!}.$$

Hence

$$e^z A_n(-z) - A_n(z) = R_n(z) = (-1)^n x^{2n+1} \sum_{k\ge 0} \frac{(n+k)!\, x^k}{(2n+k+1)!\, k!}.$$

We now have $(e^{2z}-1)(A_n(2z)+A_n(-2z)) - (e^{2z}+1)(A_n(2z)-A_n(-2z)) = 2R_n(2z)$; hence

$$\tanh z - \big/ z^{-1}, 3z^{-1}, \ldots, (2n-1)z^{-1} \big/ = \frac{2R_n(2z)}{(A_n(2z) + A_n(-2z))(e^{2z}+1)}.$$

Thus we have an exact formula for the difference. When $|z| \le 1$, the factor $e^{2z}+1$ is bounded away from zero, $|R_n(2z)| \le e\, n!/(2n+1)!$, and

$$\begin{aligned} \tfrac{1}{2}|A_n(2z) + A_n(-2z)| &\ge n! \left(\binom{2n}{n} - \binom{2n-2}{n} - \binom{2n-4}{n} - \cdots \right) \\ &\ge \frac{(2n)!}{n!}(1 - \tfrac{1}{4} - \tfrac{1}{16} - \cdots) = \frac{2}{3}\,\frac{(2n)!}{n!}. \end{aligned}$$

Thus convergence is very rapid, even for complex values of z.

To go from this continued fraction to the continued fraction for e^z, we have $\tanh z = 1 - 2/(e^{2z}+1)$; hence we get the continued-fraction representation for $(e^{2z}+1)/2$ by simple manipulations. Hurwitz's rule gives the expansion of $e^{2z}+1$, from which we may subtract unity. For n odd,

$$e^{-2/n} = \big/ \overline{1, 3mn + \lfloor n/2 \rfloor, (12m+6)n, (3m+2)n + \lfloor n/2 \rfloor, 1} \big/, \qquad m \ge 0.$$

Another derivation has been given by C. S. Davis, *J. London Math. Soc.* **20** (1945), 194–198.

17. (b) $/x_1 - 1, 1, x_2 - 2, 1, x_3 - 2, 1, \ldots, 1, x_{2n-1} - 2, 1, x_{2n} - 1/$. [*Note:* One can remove negative parameters from continuants by using the identity

$$Q_{m+n+1}(x_1, \ldots, x_n, -x, y_m, \ldots, y_1) = (-1)^{m-1} Q_{m+n+2}(x_1, \ldots, x_{n-1}, x_n - 1, 1, x - 1, -y_m, \ldots, -y_1),$$

from which we obtain

$$Q_{m+n+1}(x_1, \ldots, x_n, -x, y_m, \ldots, y_1) = -Q_{m+n+3}(x_1, \ldots, x_{n-1}, x_n - 1, 1, x - 2, 1, y_m - 1, y_{m-1}, \ldots, y_1)$$

after a second application.]

(c) $1 + /1, 1, 3, 1, 5, 1, \ldots/ = 1 + /\overline{2m+1, 1}/, \quad m \geq 0.$

19. The sum for $1 \leq k \leq N$ is $\log_b((1+x)(N+1)/(N+1+x))$.

20. Let $H = SG$, $g(x) = (1+x)G'(x)$, $h(x) = (1+x)H'(x)$. Then (35) implies that $h(x+1)/(x+2) - h(x)/(x+1) = -(1+x)^{-2} g(1/(1+x))/(1+1/(1+x))$.

21. $\varphi(x) = c/(cx+1)^2 + (2-c)/((c-1)x+1)^2$, $U\varphi(x) = 1/(x+c)^2$. When $c \leq 1$, the minimum of $\varphi(x)/U\varphi(x)$ occurs at $x = 0$ and is $2c^2 \leq 2$. When $c \geq \phi = \frac{1}{2}(\sqrt{5}+1)$, the minimum occurs at $x = 1$ and is $\leq \phi^2$. When $c \approx 1.31266$ the values at $x = 0$ and $x = 1$ are nearly equal and the minimum is > 3.2; the bounds $(0.29)^n \varphi \leq U^n \varphi \leq (0.31)^n \varphi$ are obtained. Still better bounds come from well-chosen linear combinations $Tg(x) = \sum a_j/(x + c_j)$.

23. By the interpolation formula of exercise 4.6.4–15 with $x_0 = 0$, $x_1 = x$, $x_2 = x + \epsilon$, letting $\epsilon \to 0$, we have the general identity $R_n'(x) = (R_n(x) - R_n(0))/x + \frac{1}{2} x R_n''(\theta(x))$ for some $\theta(x)$ between 0 and x, whenever R_n is a function with continuous second derivative. Hence in this case $R_n'(x) = O(2^{-n})$.

24. ∞. [A. Khinchin, in *Compos. Math.* **1** (1935), 361–382, proved that the sum $A_1 + \cdots + A_n$ of the first n partial quotients of a real number X will be asymptotically $n \lg n$, for almost all X. Exercise 35 shows that the behavior is different for rational X.]

25. Any union of intervals can be written as a union of disjoint intervals, since we have $\bigcup_{k \geq 1} I_k = \bigcup_{k \geq 1} (I_k \setminus \bigcup_{1 \leq j < k} I_j)$, and this is a disjoint union in which $I_k \setminus \bigcup_{1 \leq j < k} I_j$ can be expressed as a finite union of disjoint intervals. Therefore we may take $\mathcal{I} = \bigcup I_k$, where I_k is an interval of length $\epsilon/2^k$ containing the kth rational number in $[0, 1]$, using some enumeration of the rationals. In this case $\mu(\mathcal{I}) \leq \epsilon$, but $\|\mathcal{I} \cap P_n\| = n$ for all n.

26. The continued fractions $/A_1, \ldots, A_t/$ that appear are precisely those for which $A_1 > 1$, $A_t > 1$, and $Q_t(A_1, A_2, \ldots, A_t)$ is a divisor of n. Therefore (6) completes the proof. [*Note:* If $m_1/n = /A_1, \ldots, A_t/$ and $m_2/n = /A_t, \ldots, A_1/$, where m_1 and m_2 are relatively prime to n, then $m_1 m_2 \equiv \pm 1$ (modulo n); this rule defines the correspondence. When $A_1 = 1$ an analogous symmetry is valid, according to (44).]

27. First prove the result for $n = p^e$, then for $n = rs$, where r and s are relatively prime. Alternatively, use the formulas in the next exercise.

28. (a) The left-hand side is multiplicative (see exercise 1.2.4–31), and it is easily evaluated when n is a power of a prime. (c) From (a), we have *Möbius's inversion formula*: If $f(n) = \sum_{d \backslash n} g(d)$, then $g(n) = \sum_{d \backslash n} \mu(n/d) f(d)$.

29. The sum is approximately $(((12\ln 2)/\pi^2)\ln N!)/N - \sum_{d\geq 1}\Lambda(d)/d^2 + 1.47$; here $\sum_{d\geq 1}\Lambda(d)/d^2$ converges to the constant value $-\zeta'(2)/\zeta(2)$, and we know that $\ln N! = N\ln N - N + O(\log N)$ by Stirling's approximation.

30. The modified algorithm affects the calculation if and only if the following division step in the unmodified algorithm would have the quotient 1, and in this case it avoids the following division step. The probability that a given division step is avoided is the probability that $A_k = 1$ and that this quotient is preceded by an even number of quotients equal to 1. By the symmetry condition, this is the probability that $A_k = 1$ and is *followed* by an even number of quotients equal to 1. The latter happens if and only if $X_{k-1} > \phi - 1 = 0.618\ldots$, where ϕ is the golden ratio: For $A_k = 1$ and $A_{k+1} > 1$ iff $\frac{2}{3} \leq X_{k-1} < 1$; $A_k = A_{k+1} = A_{k+2} = 1$ and $A_{k+3} > 1$ iff $\frac{5}{8} \leq X_{k-1} < \frac{2}{3}$; etc. Thus we save approximately $F'_{k-1}(1) - F_{k-1}(\phi - 1) \approx 1 - \lg\phi \approx 0.306$ of the division steps. The average number of steps is approximately $((12\ln\phi)/\pi^2)\ln n$, when $v = n$ and u is relatively prime to n. Kronecker [*Vorlesungen über Zahlentheorie* **1** (Leipzig: Teubner, 1901), 118] observed that this choice of least remainder in absolute value always gives the shortest possible number of iterations, over all algorithms that replace u by $(\pm u) \bmod v$ at each iteration. For further results see N. G. de Bruijn and W. M. Zaring, *Nieuw Archief voor Wiskunde* (3) **1** (1953), 105–112; G. J. Rieger, *Math. Nachr.* **82** (1978), 157–180.

On many computers, the modified algorithm makes each division step longer; the idea of exercise 1, which saves *all* division steps when the quotient is unity, would be preferable in such cases.

31. Let $a_0 = 0$, $a_1 = 1$, $a_{n+1} = 2a_n + a_{n-1}$; then $a_n = ((1+\sqrt{2})^n - (1-\sqrt{2})^n)/2\sqrt{2}$, and the worst case (in the sense of Theorem F) occurs when $u = a_n + a_{n-1}$, $v = a_n$, $n \geq 2$.

This result is due to A. Dupré [*J. de Math.* **11** (1846), 41–64], who also investigated more general "look-ahead" procedures suggested by J. Binet. See P. Bachmann, *Niedere Zahlentheorie* **1** (Leipzig: Teubner, 1902), 99–118, for a discussion of early analyses of Euclid's algorithm.

32. (b) $Q_{m-1}(x_1,\ldots,x_{m-1})Q_{n-1}(x_{m+2},\ldots,x_{m+n})$ corresponds to Morse code sequences of length $(m+n)$ in which a dash occupies positions m and $(m+1)$; the other term corresponds to the opposite case. (Alternatively, use exercise 2. The more general identity

$$\begin{aligned} Q_{m+n}(x_1,\ldots,x_{m+n})Q_k(x_{m+1},\ldots,x_{m+k}) = \\ Q_{m+k}(x_1,\ldots,x_{m+k})Q_n(x_{m+1},\ldots,x_{m+n}) \\ + (-1)^k Q_{m-1}(x_1,\ldots,x_{m-1})Q_{n-k-1}(x_{m+k+2},\ldots,x_{m+n}) \end{aligned}$$

also appeared in Euler's paper.)

33. (a) The new representations are $x = m/d$, $y = (n-m)/d$, $x' = y' = d = \gcd(m, n-m)$, for $\frac{1}{2}n < m < n$. (b) The relation $(n/x') - y \leq x < n/x'$ defines x. (c) Count the x' satisfying (b). (d) A pair of integers $x > y > 0$ with $\gcd(x,y) = 1$ can be uniquely written in the form $x = Q_m(x_1,\ldots,x_m)$, $y = Q_{m-1}(x_1,\ldots,x_{m-1})$, where $x_1 \geq 2$ and $m \geq 1$; here $y/x = /x_m,\ldots,x_1/$. (e) It suffices to show that $\sum_{1\leq k\leq n/2} T(k,n) = 2\lfloor n/2\rfloor + h(n)$. For $1 \leq k \leq n/2$ the continued fractions $k/n = /x_1,\ldots,x_m/$ run through all sequences $(x_1,\ldots,x_m)$ such that $m \geq 1$, $x_1 \geq 2$, $x_m \geq 2$, $Q_m(x_1,\ldots,x_m) \setminus n$; and $T(k,n) = 2 + (m-1)$.

34. (a) Dividing x and y by $\gcd(x,y)$ yields $g(n) = \sum_{d\backslash n} h(n/d)$; apply exercise 28(c), and use the symmetry between primed and unprimed variables. (b) For fixed y and t, the representations with $xd \geq x'$ have $x' < \sqrt{nd}$; hence there are $O(\sqrt{nd}/y)$ such representations. Now sum for $0 < t \leq y < \sqrt{n/d}$. (c) If $s(y)$ is the given sum, then $\sum_{d\backslash y} s(d) = y(H_{2y} - H_y) = k(y)$, say; hence $s(y) = \sum_{d\backslash y} k(y/d)$. Now $k(y) = y\ln 2 - \frac{1}{4} + O(1/y)$. (d) $\sum_{1\leq y\leq n} \varphi(y)/y^2 = \sum_{1\leq y\leq n,\, d\backslash y} \mu(d)/yd = \sum_{cd\leq n} \mu(d)/cd^2$. (Similarly, $\sum_{1\leq y\leq n} \sigma_{-1}(y)/y^2 = O(1)$.) (e) $\sum_{1\leq k\leq n} \mu(k)/k^2 = 6/\pi^2 + O(1/n)$ (see exercise 4.5.2–10(d)); and $\sum_{1\leq k\leq n} \mu(k)\log k/k^2 = O(1)$. Hence we have $h_d(n) = n((3\ln 2)/\pi^2)\ln(n/d) + O(n)$ for $d \geq 1$. Finally $h(n) = 2\sum_{cd\backslash n} \mu(d)h_c(n/cd) = ((6\ln 2)/\pi^2)n(\ln n - \sum - \sum') + O(n\sigma_{-1}(n)^2)$, where the remaining sums are $\sum = \sum_{cd\backslash n} \mu(d)\ln(cd)/cd = 0$ and $\sum' = \sum_{cd\backslash n} \mu(d)\ln c/cd = \sum_{d\backslash n} \Lambda(d)/d$. [It is well known that $\sigma_{-1}(n) = O(\log\log n)$; cf. Hardy and Wright, *Theory of Numbers*, §22.9.]

35. See *Proc. Nat. Acad. Sci.* **72** (1975), 4720–4722.

36. Working the algorithm backwards, we want to choose $k_1, \ldots, k_{n-1}$ so that $u_k \equiv F_{k_1}\ldots F_{k_i-1}F_{k_i} - 1$ (modulo $\gcd(u_{i+1},\ldots,u_n)$) for $1 \leq i < n$, with $u_n = F_{k_1}\ldots F_{k_{n-1}}$ a minimum, where the k's are positive, $k_1 \geq 3$, and $k_1 + \cdots + k_{n-1} = N + n - 1$. The solution is $k_2 = \cdots = k_{n-1} = 2$, $u_n = F_{N-n+3}$. [See *CACM* **13** (1970), 433–436, 447–448.]

37. See *Proc. Amer. Math. Soc.* **7** (1956), 1014–1021; cf. also exercise 6.1–18.

38. Let $m = \lceil n/\phi \rceil$, so that $m/n = \phi^{-1} + \epsilon = /x_1, x_2, \ldots/$ where $0 < \epsilon < 1/n$. Let k be minimal such that $x_k \geq 2$; then $(\phi^{1-k} + (-1)^k F_{k-1}\epsilon)/(\phi^{-k} - (-1)^k F_k\epsilon) \geq 2$, hence k is even and $\phi^{-2} = 2 - \phi \leq \phi^k F_{k+2}\epsilon = (\phi^{2k+2} - \phi^{-2})\epsilon/\sqrt{5}$. [*Ann. Polon. Math.* **1** (1954), 203–206.]

39. At least 287 at bats; $/2,1,95/ = 96/287 = .33449477\ldots$, and no fraction with denominator < 287 lies in the interval $[.3335, .3345] = [/2,1,666/, /2,1,94,1,1,3/]$.

To solve the general question of the fraction in $[a,b]$ with smallest denominator, where $0 < a < b < 1$, note that in terms of regular continued-fraction representations we have $/x_1,x_2,\ldots/ < /y_1,y_2,\ldots/$ iff $(-1)^j x_j < (-1)^j y_j$ for the smallest j with $x_j \neq y_j$, where we place "∞" after the last partial quotient of a rational number. Thus if $a = /x_1,x_2,\ldots/$ and $b = /y_1,y_2,\ldots/$, and if j is minimal with $x_j \neq y_j$, the fractions in $[a,b]$ have the form $c = /x_1,\ldots,x_{j-1},z_j,\ldots,z_m/$ where $/z_j,\ldots,z_m/$ lies between $/x_j,x_{j+1},\ldots/$ and $/y_j,y_{j+1},\ldots/$ inclusive. Let $Q_{-1} = 0$. The denominator

$$Q_{j-1}(x_1,\ldots,x_{j-1})Q_{m-j+1}(z_j,\ldots,z_m) + Q_{j-2}(x_1,\ldots,x_{j-2})Q_{m-j}(z_{j+1},\ldots,z_m)$$

of c is minimized when $m = j$ and $z_j = (j \text{ odd} \Rightarrow y_j + 1 - \delta_{y_{j+1}\infty};\ x_j + 1 - \delta_{x_{j+1}\infty})$. [Another way to derive this method comes from the theory in the following exercise.]

40. One can prove by induction that $p_r q_l - p_l q_r = 1$ at each node, hence p_l and q_l are relatively prime. Since $p/q < p'/q'$ implies that $p/q < (p+p')/(q+q') < p'/q'$, it is also clear that the labels on all left descendants of p/q are less than p/q, while the labels on all its right descendants are greater. Therefore each rational number occurs at most once as a label.

It remains to show that each rational does appear. If $p/q = /a_1,\ldots,a_r,1/$, where each a_i is a positive integer, one can show by induction that the node labeled p/q is found by going left a_1 times, then right a_2 times, then left a_3 times, etc.

[The sequence of labels on successive levels of this tree was first studied by M. A. Stern, *J. für die reine und Angew. Math.* **55** (1858), 193–220, although the relation to binary trees is not explicit in his paper. Peirce independently communicated this construction in a letter dated July 17, 1903, but he never published it; and during the next few years he occasionally amused himself by making rather cryptic remarks about it without revealing the underlying mechanism. See C. S. Peirce, *The New Elements of Mathematics* **3** (The Hague: Mouton, 1976), 781–784, 826–829; also **1**, 207–211; and his *Collected Papers* **4** (1933), 276–280. See also D. H. Lehmer, *AMM* **36** (1929), 59–67.]

41. In fact, the regular continued fractions for numbers of the general form

$$\frac{1}{l_1}+\frac{(-1)^{e_1}}{l_1^2 l_2}+\frac{(-1)^{e_2}}{l_1^4 l_2^2 l_3}+\cdots$$

have an interesting pattern, based on the continuant identity

$$\begin{aligned}Q_{m+n+1}(x_1,\ldots,x_{n-1},x_n-1,1,y_m-1,y_{m-1},\ldots,y_1)=\\ x_nQ_{n-1}(x_1,\ldots,x_{n-1})Q_m(y_m,\ldots,y_1)\\ +(-1)^mQ_{m+n}(x_1,\ldots,x_{n-1},0,-y_m,-y_{m-1},\ldots,-y_1).\end{aligned}$$

This identity is most interesting when $y_m = x_{n-1}$, $y_{m-1} = x_{n-2}$, etc., since

$$Q_{n+1}(z_1,\ldots,z_k,0,z_{k+1},\ldots,z_n)=Q_{n-1}(z_1,\ldots,z_{k-1},z_k+z_{k+1},z_{k+2},\ldots,z_n).$$

In particular we find that if $p_n/q_n = Q_{n-1}(x_2,\ldots,x_n)/Q_n(x_1,\ldots,x_n) = /x_1,\ldots,x_n/$, then $p_n/q_n+(-1)^n/q_n^2 r = /x_1,\ldots,x_n,r-1,1,x_n-1,x_{n-1},\ldots,x_1/$. By changing $/x_1,\ldots,x_n/$ to $/x_1,\ldots,x_{n-1},x_n-1,1/$, we can control the sign $(-1)^n$ as desired.

For example, the partial sums of the first series have the following continued fractions of even length: $/1,1/$; $/1,1,1,1,0,1/ = /1,1,1,2/$; $/1,1,1,2,1,1,1,1,1,1/$; $/1,1,1,2,1,1,1,1,1,1,1,1,0,1,1,1,1,1,2,1,1,1/ = /1,1,1,2,1,1,1,1,1,1,1,2,1,1,1,1,2,1,1,1/$; and from this point on the sequence settles down and obeys a simple reflecting pattern. We find that the nth partial quotient a_n can be computed rapidly as follows, if $n-1 = 20q+r$ where $0\le r<20$:

$$a_n=\begin{cases}1, & \text{if } r=0,2,4,5,6,7,9,10,12,13,14,15,17 \text{ or } 19;\\ 2, & \text{if } r=3 \text{ or } 16;\\ 1+(q+r)\bmod 2, & \text{if } r=8 \text{ or } 11;\\ 2-d_q, & \text{if } r=1;\\ 1+d_{q+1}, & \text{if } r=18.\end{cases}$$

Here d_n is the "dragon sequence" defined by the rules $d_0=1$, $d_{2n}=d_n$, $d_{4n+1}=0$, $d_{4n+3}=1$; the dragon curve discussed in exercise 4.1–18 turns right at its nth step iff $d_n=1$.

Liouville's numbers with $l\ge 3$ are equal to $/l-1,l+1,l^2-1,1,l,l-1,l^{12}-1,1,l-2,l,1,l^2-1,l+1,l-1,l^{72}-1,\ldots/$. The nth partial quotient a_n depends on the dragon sequence on $m \bmod 4$ as follows: If $n \bmod 4 = 1$ it is $l-2+d_{n-1}+(\lfloor n/2\rfloor \bmod 4)$ and if $n \bmod 4 = 2$ it is $l+2-d_{n+2}-(\lfloor n/2\rfloor \bmod 4)$; if $n \bmod 4 = 0$ it is 1 or $l^{k!(k-1)}-1$, depending on whether or not $d_n=0$ or 1, where k is the largest power of 2 dividing n; and if $n \bmod 4 = 3$ it is $l^{k!(k-1)}-1$ or 1, depending on whether $d_{n+1}=0$ or 1, where k is the largest power of 2 dividing $n+1$. When $l=2$ the same rules apply, except that 0's must be removed, so there is a more complicated pattern depending on $n \bmod 24$.

[Cf. *J. Number Theory* **11** (1979), 209–217.]

42. Suppose that $\|qX\| = |qX - p|$. We can always find integers u and v such that $q = uq_{n-1} + vq_n$ and $p = up_{n-1} + vp_n$, where $p_n = Q_{n-1}(A_2, \ldots, A_n)$, since $q_np_{n-1} - q_{n-1}p_n = \pm 1$. We must have $uv < 0$, hence $u(q_{n-1}X - p_{n-1})$ has the same sign as $v(q_nX - p_n)$, and $|qX - p| = |u|\,|q_{n-1}X - p_{n-1}| + |v|\,|q_nX - p_n|$. This completes the proof, since $u \neq 0$. See Theorem 6.4S for a generalization.

43. If x is representable, so is the father of x in the Stern–Peirce tree of exercise 40; thus the representable numbers form a subtree of that binary tree. Let (u/u') and (v/v') be adjacent representable numbers. Then one is an ancestor of the other; say (u/u') is an ancestor of (v/v'), since the other case is similar. Then (u/u') is the nearest left ancestor of (v/v'), so all numbers between u/u' and v/v' are left descendants of (v/v') and the mediant $((u + v)/(u' + v'))$ is its left son. According to the relation between regular continued fractions and the binary tree, the mediant and all of its left descendants will have (u/u') as their last representable p_i/q_i, while all of the mediant's right descendants will have (v/v') as one of the p_i/q_i. (The numbers p_i/q_i label the *fathers* of the 'turning-point' nodes on the path to x.)

44. A counterexample for $M = N = 100$ is $(u/u') = \frac{1}{3}$, $(v/v') = \frac{67}{99}$. However, the identity is almost always true, because of (12); it fails only when $u/u' + v/v'$ is very nearly equal to a fraction that is simpler than (u/u').

45. See M. S. Waterman, *BIT* **17** (1977), 465–478.

SECTION 4.5.4

1. If d_k isn't prime, its prime factors are cast out before d_k is tried.

2. No; the algorithm would fail if $p_{t-1} = p_t$, giving "1" as a spurious prime factor.

3. Let P be the product of the first 168 primes. [*Note:* Although $P = 19590\ldots5910$ is a 416-digit number, such a gcd can be computed in much less time than it would take to do 168 divisions, if we just want to test whether or not n is prime.]

4. In the notation of exercise 3.1–11,

$$\sum_{\mu,\,\lambda} 2^{\lceil \lg \max(\mu+1,\lambda)\rceil} P(\mu,\lambda) = \frac{1}{m} \sum_{l \geq 1} f(l) \prod_{1 \leq k < l} \left(1 - \frac{k}{m}\right),$$

where $f(l) = \sum_{1 \leq \lambda \leq l} 2^{\lceil \lg \max(l-\lambda,\lambda)\rceil}$. If $l = 2^{k+\theta}$, where $0 < \theta \leq 1$, we have $f(l) = l^2(3 \cdot 2^{-\theta} - 2 \cdot 2^{-2\theta})$, where the function $3 \cdot 2^{-\theta} - 2 \cdot 2^{-2\theta}$ reaches a maximum of $\frac{9}{8}$ at $\theta = \lg(4/3)$ and has a minimum of 1 at $\theta = 0$ and 1. Therefore the average value of $2^{\lceil \lg \max(\mu+1,\lambda)\rceil}$ lies between 1.0 and 1.125 times the average value of $\mu + \lambda$, and the result follows.

[Algorithm B is a refinement of Pollard's original algorithm, which was based on exercise 3.1–6(b) instead of the (yet undiscovered) result in exercise 3.1–7. He showed that the least n such that $X_{2n} = X_n$ has average value $\sim (\pi^2/12)Q(m)$; this constant $\pi^2/12$ is explained by Eq. 4.5.3–21. Hence the average value of $3n$ in his original method is $\sim (\pi/2)^{5/2}\sqrt{m} = 3.092\sqrt{m}$. Richard Brent has observed that, as $m \to \infty$, the density $\prod_{1 \leq k < l}(1 - k/m) = \exp(-l(l-1)/2m + O(l^3/m^2))$ approaches a normal distribution, and we may assume that θ is uniformly distributed. Then $3 \cdot 2^{-\theta} - 2 \cdot 2^{-2\theta}$ takes the average value $3/(4 \ln 2)$, and the average number of iterations needed by Algorithm B comes to $\sim (3/(4 \ln 2) + \frac{1}{2})\sqrt{\pi m/2} = 1.983\sqrt{m}$. A similar analysis of the more general method in the answer to exercise 3.1–7 gives $\sim 1.926\sqrt{m}$, when $p = 2.4771$ is chosen "optimally" as the root of $(p^2 - 1)\ln p = p^2 - p + 1$.]

5. $x \bmod 3 = 0$; $x \bmod 5 = 0, 1, 4$; $x \bmod 7 = 0, 1, 6$; $x \bmod 8 = 1, 3, 5, 7$; $x > 103$. The first try is $x = 105$; and $(105)^2 - 10541 = 484 = 22^2$. This would also have been found by Algorithm C in a relatively short time. Thus $10541 = 83 \cdot 127$.

6. Let us count the number of solutions (x, y) of the congruence $N \equiv (x - y)(x + y)$ (modulo p), where $0 \le x, y < p$. Since $N \not\equiv 0$ and p is prime, $x + y \not\equiv 0$. For each $v \not\equiv 0$ there is a unique u (modulo p) such that $N \equiv uv$. The congruences $x - y \equiv u$, $x + y \equiv v$ now uniquely determine $x \bmod p$ and $y \bmod p$, since p is odd. Thus the stated congruence has exactly $p - 1$ solutions (x, y). If (x, y) is a solution, so is $(x, p - y)$ if $y \not\equiv 0$, since $(p - y)^2 \equiv y^2$; and if (x, y_1) and (x, y_2) are solutions with $y_1 \not\equiv y_2$, we have $y_1^2 \equiv y_2^2$; hence $y_1 = p - y_2$. Thus the number of different x values among the solutions (x, y) is $(p - 1)/2$ if $N \equiv x^2$ has no solutions, or $(p + 1)/2$ if $N \equiv x^2$ has solutions.

7. One procedure is to keep two indices for each modulus, one for the current word position and one for the current bit position; loading two words of the table and doing an indexed shift command will bring the table entries into proper alignment. (Many computers have special facilities for such bit manipulation.)

8. (We may assume that $N = 2M$ is even.) The following algorithm uses an auxiliary table $X[1], X[2], \ldots, X[M]$, where $X[k]$ represents the primality of $2k + 1$.

S1. Set $X[k] \leftarrow 1$ for $1 \le k \le M$. Also set $j \leftarrow 1$, $p \leftarrow 1$, $p \leftarrow 3$, $q \leftarrow 4$. (During this algorithm $p = 2j + 1$, $q = 2j + 2j^2$; the integer variables j, k, p, q may readily be manipulated in index registers.)

S2. If $X[j] = 0$, go to S4. Otherwise output p, which is prime, and set $k \leftarrow q$.

S3. If $k \le M$, then set $X[k] \leftarrow 0$, $k \leftarrow k + p$, and repeat this step.

S4. Set $j \leftarrow j + 1$, $p \leftarrow p + 2$, $q \leftarrow q + 2p - 2$. If $j \le M$, return to S2. ∎

A major part of this calculation could be made noticeably faster if q (instead of j) were tested against M in step S4, and if a new loop were appended that outputs $2j + 1$ for all remaining $X[j]$ that equal 1, suppressing the manipulation of p and q.

Improvements in the efficiency of sieve methods for generating primes are discussed in exercise 5.2.3–15 and in Section 7.1.

9. If p^2 is a divisor of n for some prime p, then p is a divisor of $\lambda(n)$, but not of $n - 1$. If $n = p_1 p_2$, where $p_1 < p_2$ are primes, then $p_2 - 1$ is a divisor of $\lambda(n)$ and therefore $p_1 p_2 - 1 \equiv 0$ (modulo $p_2 - 1$). Since $p_2 \equiv 1$, this means $p_1 - 1$ is a multiple of $p_2 - 1$, contradicting the assumption $p_1 < p_2$. [Values of n for which $\lambda(n)$ properly divides $n - 1$ are called "Carmichael numbers." For example, here are some small Carmichael numbers with up to six prime factors: $3 \cdot 11 \cdot 17$, $5 \cdot 13 \cdot 17$, $7 \cdot 11 \cdot 13 \cdot 41$, $5 \cdot 7 \cdot 17 \cdot 19 \cdot 73$, $5 \cdot 7 \cdot 17 \cdot 73 \cdot 89 \cdot 107$.]

10. Let k_p be the order of x_p modulo n, and let λ be the least common multiple of all the k_p's. Then λ is a divisor of $n - 1$ but not of any $(n - 1)/p$, so $\lambda = n - 1$. Since $x_p^{\varphi(n)} \bmod n = 1$, $\varphi(n)$ is a multiple of k_p for all p, so $\varphi(n) \ge \lambda$. But $\varphi(n) < n - 1$ when n is not prime. (Another way to carry out the proof is to construct an element x of order $n - 1$ from the x_p's, by the method of exercise 3.2.1.2–15.)

11.

U	V	A	P	S	T	Output
1984	1	0	992	0	—	
1981	1981	1	992	1	1981	
1983	4	495	993	0	1	$993^2 \equiv +2^2$
1983	991	2	98109	1	991	
1981	4	495	2	0	1	$2^2 \equiv +2^2$
1984	1981	1	99099	1	1981	
1984	1	1984	99101	0	1	$99101^2 \equiv +2^0$

The factorization $199 \cdot 991$ is evident from the first or last outputs. The shortness of the cycle, and the appearance of the well-known number 1984, are probably just coincidences.

12. The following algorithm makes use of an auxiliary $(m+1) \times (m+1)$ matrix of single-precision integers E_{jk}, $0 \le j, k \le m$; a single-precision vector $(b_0, b_1, \ldots, b_m)$; and a multiple-precision vector $(x_0, x_1, \ldots, x_m)$ with entries in the range $0 \le x_k < N$.

F1. [Initialize.] Set $b_i \leftarrow -1$ for $0 \le i \le m$; then set $j \leftarrow 0$.

F2. [Next solution.] Get the next output $(x, e_0, e_1, \ldots, e_m)$ produced by Algorithm E. (It is convenient to regard Algorithms E and F as coroutines.) Set $k \leftarrow 0$.

F3. [Search for odd.] If $k > m$ go to step F5. Otherwise if e_k is even, set $k \leftarrow k+1$ and repeat this step.

F4. [Linear dependence?] If $b_k \ge 0$, then set $i \leftarrow b_k$, $x \leftarrow (x_i x) \bmod N$, $e_r \leftarrow e_r + E_{ir}$ for $0 \le r \le m$; set $k \leftarrow k+1$ and return to F3. Otherwise set $b_k \leftarrow j$, $x_j \leftarrow x$, $E_{jr} \leftarrow e_r$ for $0 \le r \le m$; set $j \leftarrow j+1$ and return to F2. (In the latter case we have a new linearly independent solution, modulo 2, whose first odd component is e_k.)

F5. [Try to factor.] (Now e_0, e_1, ..., e_m are even.) Set

$$y \leftarrow \left((-1)^{e_0/2} p_1^{e_1/2} \ldots p_m^{e_m/2}\right) \bmod N.$$

If $x = y$ or if $x + y = N$, return to F2. Otherwise compute $\gcd(x - y, N)$, which is a proper factor of N, and terminate the algorithm. ∎

It can be shown that this algorithm finds a factor, whenever it is possible to deduce one from the given outputs of Algorithm E. [*Proof:* Let the outputs of Algorithm E be $(X_i, E_{i0}, \ldots, E_{im})$ for $1 \le i \le t$, and suppose that we could find a factorization $N = N_1 N_2$ when $x \equiv X_1^{a_1} \ldots X_t^{a_t}$ and $y \equiv (-1)^{e_0/2} p_1^{e_1/2} \ldots p_m^{e_m/2}$ (modulo N), where $e_j = a_1 E_{1j} + \cdots + a_t E_{tj}$ is even for all j. Then $x \equiv \pm y$ (modulo N_1) and $x \equiv \mp y$ (modulo N_2). It is not difficult to see that this solution can be transformed into a pair (x, y) that appears in step F5, by a series of steps that systematically replace (x, y) by (xx', yy') where $x' \equiv \pm y'$ (modulo N).]

13. There are 2^d values of x having the same exponents $(e_0, \ldots, e_m)$, since we can choose the sign of x modulo $q_i^{f_i}$ arbitrarily when $N = q_1^{f_1} \ldots q_d^{f_d}$. Exactly two of these 2^d values will fail to yield a factor.

14. Since $P^2 \equiv kNQ^2$ (modulo p) for any prime divisor p of V, we have $1 \equiv P^{2(p-1)/2} \equiv (kNQ^2)^{(p-1)/2} \equiv (kN)^{(p-1)/2}$ (modulo p), if $P \not\equiv 0$.

15. $U_n = (a^n - b^n)/\sqrt{D}$, where $a = \frac{1}{2}(P+\sqrt{D})$, $b = \frac{1}{2}(P-\sqrt{D})$, $D = P^2-4Q$. Then $2^{n-1}U_n = \sum_k \binom{n}{2k+1} P^{n-2k-1}D^k$; so $U_p \equiv D^{(p-1)/2}$ (modulo p) if p is an odd prime. Similarly, if $V_n = a^n + b^n = U_{n+1} - QU_{n-1}$, then $2^{n-1}V_n = \sum_k \binom{n}{2k} P^{n-2k}D^k$, and $V_p \equiv P^n \equiv P$. Thus if $U_p \equiv -1$, we find that $U_{p+1} \bmod p = 0$. If $U_p \equiv 1$, we find that $(QU_{p-1}) \bmod p = 0$; here if Q is a multiple of p, $U_n \equiv P^{n-1}$ (modulo p) for $n > 0$, so U_n is never a multiple of p; if Q is not a multiple of p, $U_{p-1} \bmod p = 0$. Therefore as in Theorem L, $U_t \bmod N = 0$ if $N = p_1^{e_1}\ldots p_r^{e_r}$, $\gcd(N, Q) = 1$, and $t = \operatorname{lcm}_{1\le j\le r}(p_j^{e_j-1}(p_j + \epsilon_j))$. Under the assumptions of this exercise, the rank of apparition of N is $N+1$; hence N is prime to Q and t is a multiple of $N+1$. Also, the assumptions of this exercise imply that each p_j is odd and each ϵ_j is ± 1, so $t \le 2^{1-r}\prod p_j^{e_j-1}(p_j + \frac{1}{3}p_j) = 2(\frac{2}{3})^r N$; hence $r = 1$ and $t = p_1^{e_1} + \epsilon_1 p_1^{e_1-1}$. Finally, therefore, $e_1 = 1$ and $\epsilon_1 = 1$.

Note: If this test for primality is to be any good, we must choose P and Q in such a way that the test will probably work. Lehmer suggests taking $P = 1$ so that $D = 1 - 4Q$, and choosing Q so that $\gcd(N, QD) = 1$. (If the latter condition fails, we know already that N is not prime, unless $|QD| \ge N$.) Furthermore, the derivation above shows that we will want $\epsilon_1 = 1$, that is, $D^{(N-1)/2} \equiv -1$ (modulo N). This is another condition that determines the choice of Q. Furthermore, if D satisfies this condition, and if $U_{N+1} \bmod N \neq 0$, we know that N is *not* prime.

Example: If $P = 1$ and $Q = -1$, we have the Fibonacci sequence, with $D = 5$. Since $5^{11} \equiv -1$ (modulo 23), we might attempt to prove that 23 is prime by using the Fibonacci sequence:

$$\langle F_n \bmod 23\rangle = 0, 1, 1, 2, 3, 5, 8, 13, 21, 11, 9, 20, 6, 3, 9, 12, 21, 10, 8, 18, 3, 21, 1, 22, 0, \ldots,$$

so 24 is the rank of apparition of 23 and the test works. However, the Fibonacci sequence cannot be used in this way to prove the primality of 13 or 17, since $F_7 \bmod 13 = 0$ and $F_9 \bmod 17 = 0$. When $p \equiv \pm 1$ (modulo 10), we have $5^{(p-1)/2} \bmod p = 1$, so F_{p-1} (not F_{p+1}) is divisible by p.

17. Let $f(q) = 2\lg q - 1$. When $q = 2$ or 3, the tree has at most $f(q)$ nodes. When $q > 3$ is prime, let $q = 1 + q_1 \ldots q_t$ where $t \ge 2$ and $q_1, \ldots, q_t$ are prime. The size of the tree is $\le 1 + \sum f(q_k) = 2 + f(q-1) - t < f(q)$. [*SIAM J. Computing* **7** (1975), 214–220.]

18. $x(G(\alpha) - F(\alpha))$ is the number of $n \le x$ whose second-largest prime factor is $\le x^\alpha$ and whose largest prime factor is $> x^\alpha$. Hence

$$xG'(t)\,dt = (\pi(x^{t+dt}) - \pi(x^t)) \cdot x^{1-t}(G(t/(1-t)) - F(t/(1-t))).$$

The probability that $p_{t-1} \le \sqrt{p_t}$ is $\int_0^1 F(t/2(1-t))t^{-1}\,dt$. [Curiously, it can be shown that this also equals $\int_0^1 F(t/(1-t))\,dt$, the average value of $\log p_t/\log x$, and it also equals Golomb's constant λ of exercises 1.3.3–23 and 3.1–13. The derivative $G'(0)$ can be shown to equal $\int_0^1 F(t/(1-t))t^{-2}\,dt = F(1) + 2F(\frac{1}{2}) + 3F(\frac{1}{3}) + \cdots = e^\gamma$. The third-largest prime factor has $H(\alpha) = \int_0^\alpha (H(t/(1-t)) - G(t/(1-t)))t^{-1}\,dt$ and $H'(0) = \infty$. See P. Billingsley, *Period. Math. Hungar.* **2** (1972), 283–289; J. Galambos, *Acta Arith.* **31** (1976), 213–218; D. E. Knuth and L. Trabb Pardo, *Theoretical Comp. Sci.* **3** (1976), 321–348.]

19. $M = 2^D - 1$ is a multiple of all p for which the order of 2 modulo p divides D. To extend this idea, let $a_1 = 2$ and $a_{j+1} = a_j^{q_j} \bmod N$, where $q_j = p_j^{e_j}$, p_j is the jth prime, and $e_j = \lfloor \log 1000/\log p_j \rfloor$; let $A = a_{169}$. Now compute $b_q = \gcd(A^q - 1, N)$ for all primes q between 10^3 and 10^5. One way to do this is to start with $A^{1009} \bmod N$ and then to multiply alternately by $A^4 \bmod N$ and $A^2 \bmod N$. (A similar method was used in the 1920s by D. N. Lehmer, but he didn't publish it.) As with Algorithm B we can avoid most of the gcd's by batching; e.g., since $b_{30r-k} = \gcd(A^{30r} - A^k, N)$, we might try batches of 8, computing $c_r = (A^{30r} - A^{29})(A^{30r} - A^{23}) \ldots (A^{30r} - A) \bmod N$, then $\gcd(c_r, N)$ for $33 < r \leq 3334$.

22. Algorithm P fails only when the random number x does not reveal the fact that n is nonprime. Say x is *bad* if $x^q \bmod n = 1$ or if one of the numbers $x^{2^j q}$ is $\equiv -1$ (modulo n) for $0 \leq j < k$. Since 1 is bad, we have $p_n = (b_n - 1)/(n-2) < b_n/(n-1)$, where b_n is the number of bad x such that $1 \leq x < n$, when n is not prime.

Every bad x satisfies $x^{n-1} \equiv 1$ (modulo n). When p is prime, the number of solutions to the congruence $x^q \equiv 1$ (modulo p^e) for $1 \leq x \leq p^e$ is the number of solutions of $qy \equiv 0$ (modulo $p^{e-1}(p-1)$) for $0 \leq y < p^{e-1}(p-1)$, namely $\gcd(q, p^{e-1}(p-1))$, since we may replace x by a^y where a is a primitive root.

Let $n = n_1^{e_1} \ldots n_r^{e_r}$, where the n_i are distinct primes. According to the Chinese remainder theorem, the number of solutions to the congruence $x^{n-1} \equiv 1$ (modulo n) is $\prod_{1 \leq i \leq r} \gcd(n-1, n_i^{e_i - 1}(n_i - 1))$, and this is at most $\prod_{1 \leq i \leq r}(n_i - 1)$ since n_i is relatively prime to $n-1$. If some $e_i > 1$, we have $n_i - 1 \leq \frac{2}{9} n_i^{e_i}$, hence the number of solutions is at most $\frac{2}{9}n$; in this case $b_n \leq \frac{2}{9}n \leq \frac{1}{4}(n-1)$, since $n \geq 9$.

Therefore we may assume that n is the product $n_1 \ldots n_r$ of distinct primes. Let $n_i = 1 + 2^{k_i} q_i$, where $k_1 \leq \cdots \leq k_r$. Then $\gcd(n-1, n_i - 1) = 2^{k_i'} q_i'$, where $k_i' = \min(k, k_i)$ and $q_i' = \gcd(q, q_i)$. Modulo n_i, the number of x such that $x^q \equiv 1$ is q_i'; and the number of x such that $x^{2^j q} \equiv -1$ is $2^j q_i'$ for $0 \leq j < k_i'$, otherwise 0. Since $k \geq k_1$, we have $b_n = q_1' \ldots q_r' (1 + \sum_{0 \leq j < k_1} 2^{jr})$.

To complete the proof, it suffices to show that $b_n \leq \frac{1}{4} q_1 \ldots q_r 2^{k_1 + \cdots + k_r} = \frac{1}{4}\varphi(n)$, since $\varphi(n) < n - 1$. We have

$$\begin{aligned}(1 + \textstyle\sum_{0 \leq j < k_1} 2^{jr})/2^{k_1 + \cdots + k_r} &\leq (1 + \textstyle\sum_{0 \leq j < k_1} 2^{jr})/2^{k_1 r} \\ &= 1/(2^r - 1) + (2^r - 2)/(2^{k_1 r}(2^r - 1)) \leq 1/2^{r-1},\end{aligned}$$

so the result follows unless $r = 2$ and $k_1 = k_2$. If $r = 2$, exercise 9 shows that $n - 1$ is not a multiple of both $n_1 - 1$ and $n_2 - 1$. Thus if $k_1 = k_2$ we cannot have both $q_1' = q_1$ and $q_2' = q_2$; it follows that $q_1' q_2' \leq \frac{1}{3} q_1 q_2$ and $b_n \leq \frac{1}{6}\varphi(n)$ in this case.

[*Reference: J. Number Theory* (1980), to appear. The above proof shows that p_n is near $\frac{1}{4}$ in only two cases, when $n = (1+2q_1)(1+4q_1)$ or $(1+2q_1)(1+2q_2)(1+2q_3)$. For example, when $n = 49939 \cdot 99877$ we have $b_n = \frac{1}{4}(49938 \cdot 99876)$ and $p_n \approx .2499925$. See the next answer for further remarks.]

23. (a) The proofs are simple except perhaps for the reciprocity law. Let $p = p_1 \ldots p_s$ and $q = q_1 \ldots q_r$, where the p_i and q_j are prime. Then

$$\left(\frac{p}{q}\right) = \prod_{i,j} \left(\frac{p_i}{q_j}\right) = \prod_{i,j} (-1)^{(p_i - 1)(q_j - 1)/4} \left(\frac{q_j}{p_i}\right) = (-1)^{\sum_{i,j} (p_i - 1)(q_j - 1)/4} \left(\frac{q}{p}\right),$$

so we need only verify that $\sum_{i,j}(p_i-1)(q_j-1)/4 \equiv (p-1)(q-1)/4$ (modulo 2). But $\sum_{i,j}(p_i-1)(q_j-1)/4 = (\sum_i(p_i-1)/2)(\sum_j(q_j-1)/2)$ is odd iff an odd number of the p_i and an odd number of the q_j are $\equiv 3$ (modulo 4), and this holds iff $(p-1)(q-1)/4$ is odd.

(b) As in exercise 22, we may assume that $n = n_1 \ldots n_r$ where the $n_i = 1 + 2^{k_i}q_i$ are distinct primes, and $k_1 \le \cdots \le k_r$; we let $\gcd(n-1, n_i-1) = 2^{k_i'}q_i'$ and we call x *bad* if it falsely makes n look prime. Let $\Pi_n = \prod_{1\le i\le r} q_i' 2^{\min(k_i,k-1)}$ be the number of solutions of $x^{(n-1)/2} \equiv 1$. The number of bad x with $(\frac{x}{n}) = 1$ is Π_n, times an extra factor of $\frac{1}{2}$ if $k_1 < k$. (This factor $\frac{1}{2}$ is needed to ensure that $(\frac{x}{n_i}) = -1$ for an even number of the n_i with $k_i < k$.) The number of bad x with $(\frac{x}{n}) = -1$ is Π_n if $k_1 = k$, otherwise 0. [If $x^{(n-1)/2} \equiv -1$ (modulo n_i), we have $(\frac{x}{n_i}) = -1$ if $k_i = k$, $(\frac{x}{n_i}) = +1$ if $k_i > k$, and a contradiction if $k_i < k$. If $k_1 = k$, there are an odd number of k_i equal to k.]

Notes: The probability of a bad guess is $> \frac{1}{4}$ only if n is a Carmichael number with $k_r < k$; for example, $n = 7 \cdot 13 \cdot 19 = 1729$, a number made famous by Rāmānujan in another context. Louis Monier has extended the above analyses to obtain the following closed formulas for the number of bad x in general:

$$b_n = \left(1 + \frac{2^{rk_1}-1}{2^r-1}\right) \prod_{1\le i\le r} q_i';$$

$$b_n' = \delta_n \prod_{1\le i\le r} \gcd\left(\frac{n-1}{2},\, n_i - 1\right).$$

Here b_n' is the number of bad x in this exercise, and δ_n is either 2 (if $k_1 = k$), or $\frac{1}{2}$ (if $k_i < k$ and e_i is odd for some i), or 1 (otherwise).

(c) If $x^q \bmod n = 1$, then $1 = (\frac{x^q}{n}) = (\frac{x}{n})^q = (\frac{x}{n})$. If $x^{2^j q} \equiv -1$ (modulo n), then the order of x modulo n_i must be an odd multiple of 2^{j+1} for all prime divisors n_i of n. Let $n = n_1^{e_1} \ldots n_r^{e_r}$ and $n_i = 1 + 2^{j+1}q_i''$; then $(\frac{x}{n_i}) = (-1)^{q_i''}$, so $(\frac{x}{n}) = +1$ or -1 according as $\sum e_i q_i''$ is even or odd. Since $n \equiv (1 + 2^{j+1}\sum e_i q_i'')$ (modulo 2^{j+2}), the sum $\sum e_i q_i''$ is odd iff $j + 1 = k$. [Univ. de Paris-Sud, Lab. Rech. Informatique, Rapport 20 (1978).]

24. Let M_1 be a matrix having one row for each nonprime odd number in the range $1 \le n \le N$ and having $N-1$ columns numbered from 2 to N; the entry in row n column x is 1 if n passes the x test of Algorithm P, otherwise it is zero. When $N = qn + r$ and $0 \le r < n$, we know that row n contains at most $\frac{1}{4}qn + \min(\frac{1}{2}n, r) = \frac{1}{4}N + \min(\frac{1}{4}n - \frac{1}{4}r, \frac{3}{4}r) \le \frac{1}{4}N + \frac{3}{16}n < \frac{1}{2}N$ entries equal to 0, so at least half of the entries in the matrix are 1. Thus, some column x_1 of M_1 has at least half of its entries equal to 1. Removing column x_1 and all rows in which this column contains 1 leaves a matrix M_2 having similar properties; a repetition of this construction produces matrix M_r with $N-r$ columns and fewer than $N/2^r$ rows, and with at least $\frac{1}{2}(N-1)$ entries per row equal to 1. [Cf. *Proc. IEEE Symp. Foundations of Comp. Sci.* **19** (1978), 78.]

[A similar proof implies the existence of a *single* infinite sequence $x_1 < x_2 < \cdots$ such that the number $n > 1$ is prime if and only if it passes the x test of Algorithm P for $x = x_1, \ldots, x = x_m$, where $m = \frac{1}{2}\lfloor \lg n \rfloor (\lfloor \lg n \rfloor - 1)$. Does there exist a sequence $x_1 < x_2 < \cdots$ having this property but with $m = O(\log n)$?]

25. This theorem was first proved rigorously by von Mangoldt [*J. für die reine und angew. Math.* **114** (1895), 255–305], who showed in fact that the $O(1)$ term is equal

to $\frac{1}{2}(x \text{ is prime}) + \text{constant} + \int_x^\infty dt/(t^2-1)t\ln t$. The constant is $\operatorname{li} 2 - \ln 2 = \gamma + \ln\ln 2 + \sum_{n\geq 2}(\ln 2)^n/nn! = 0.35201\ 65995\ 57547\ 47542\ 73567\ 67736\ 43656\ 84471+$.

26. If N is not prime, it has a prime factor $q \leq \sqrt{N}$. By hypothesis, every prime divisor p of f has an integer x_p such that the order of x_p modulo q is a divisor of $N-1$ but not of $(N-1)/p$. Therefore if p^k divides f, the order of x_p modulo q is a multiple of p^k. Exercise 3.2.1.2–15 now tells us that there is an element x of order f modulo q. But this is impossible, since it implies that $q^2 \geq (f+1)^2 \geq (f+1)r \geq N$, and equality cannot hold. [*Proc. Camb. Phil. Soc.* **18** (1914), 29–30.]

27. If k is not divisible by 3 and if $k \leq 2^n+1$, the number $k\cdot 2^n+1$ is prime iff $3^{2^{n-1}k} \equiv -1$ (modulo $k\cdot 2^n+1$). For if this condition holds, $k\cdot 2^n+1$ is prime by exercise 26; and if $k\cdot 2^n+1$ is prime, the number 3 is a quadratic nonresidue mod $k\cdot 2^n+1$ by the law of quadratic reciprocity, since $k\cdot 2^n+1 \bmod 12 = 5$. [This test was stated without proof by Proth in *Comptes Rendus* **87** (Paris, 1878), 926.]

To implement Proth's test with the necessary efficiency, we need to be able to compute $x^2 \bmod (k\cdot 2^n+1)$ with about the same speed as we can compute the quantity $x^2 \bmod (2^n-1)$. Let $x^2 = A\cdot 2^n + B$ and let $A = qk + r$ where $0 \leq r < k$; we can determine q and r rapidly, when k is a single-precision number. Then $x^2 \equiv B - q + r\cdot 2^n$ (modulo $k\cdot 2^n+1$), so the remainder is easily obtained.

[To test numbers of the form $3\cdot 2^n+1$ for primality, the job is only slightly more difficult; we first try random single-precision numbers until finding one that is a quadratic nonresidue mod $3\cdot 2^n+1$ by the law of quadratic reciprocity, then use this number in place of "3" in the above test. If $n \bmod 4 \neq 0$, the number 5 can be used. It turns out that $3\cdot 2^n+1$ is prime when $n = 1, 2, 5, 6, 8, 12, 18, 30, 36, 41, 66, 189, 201, 209, 276, 353, 408, 438, 534, 2208, 2816, 3168, 3189, 3912$; and $5\cdot 2^n+1$ is prime when $n = 1, 3, 7, 13, 15, 25, 39, 55, 75, 85, 127, 1947$. See R. M. Robinson, *Proc. Amer. Math. Soc.* **9** (1958), 673–681; the additional numbers listed here were found by M. F. Plass.]

28. $f(p, p^2d) = 2/(p+1) + f(p,d)/p$, since $1/(p+1)$ is the probability that A is a multiple of p. $f(p,pd) = 1/(p+1)$ when $d \bmod p \neq 0$. $f(2, 4k+3) = \frac{1}{3}$ since $A^2 - (4k+3)B^2$ cannot be a multiple of 4; $f(2, 8k+5) = \frac{2}{3}$ since $A^2 - (8k+5)B^2$ cannot be a multiple of 8; $f(2, 8k+1) = \frac{1}{3} + \frac{1}{3} + \frac{1}{3} + \frac{1}{6} + \frac{1}{12} + \cdots = \frac{4}{3}$. $f(p,d) = (2p/(p^2-1), 0)$ if $d^{(p-1)/2} \bmod p = (1, p-1)$, respectively, for odd p.

29. The number of solutions to the equation $x_1 + \cdots + x_m \leq r$ in nonnegative integers x_i is $\binom{m+r}{r} \geq m^r/r!$, and each of these corresponds to a unique integer $p_1^{x_1}\ldots p_m^{x_m} \leq n$. [For sharper estimates, in the special case that p_j is the jth prime for all j, see N. G. de Bruijn, *Indag. Math.* **28** (1966), 240–247; H. Halberstam, *Proc. London Math. Soc.* (3) **21** (1970), 102–107.]

30. If $p_1^{e_1}\ldots p_m^{e_m} \equiv x_i^2$ (modulo q_i), we can find y_i such that $p_1^{e_1}\ldots p_m^{e_m} \equiv (\pm y_i)^2$ (modulo $q_i^{d_i}$), hence by the Chinese remainder theorem we obtain 2^d values of X such that $X^2 \equiv p_1^{e_1}\ldots p_m^{e_m}$ (modulo N). Such $(e_1, \ldots, e_m)$ correspond to at most $\binom{r}{r/2}$ pairs $(e'_1, \ldots, e'_m; e''_1, \ldots, e''_m)$ having the hinted properties. Now for each of the 2^d binary numbers $a = (a_1 \ldots a_d)_2$, let n_a be the number of exponents $(e'_1, \ldots, e'_m)$ such that $(p_1^{e'_1}\ldots p_m^{e'_m})^{(q_i-1)/2} \equiv (-1)^{a_i}$ (modulo q_i); we have proved that the required number of integers X is $\geq 2^d(\sum_a n_a^2)/\binom{r}{r/2}$. Since $\sum_a n_a$ is the number of ways to choose at most $r/2$ objects from a set of m objects with repetitions permitted, namely $\binom{m+r/2}{r/2}$, we have $\sum_a n_a^2 \geq \binom{m+r/2}{r/2}^2/2^d \geq m^r/(2^d(r/2)!)$. [Cf. J. *Number Theory*, to appear.]

31. Set $r = M$, $qM = 4m$, $\epsilon M = 2m$.

32. Let $M = \lfloor \sqrt[3]{N} \rfloor$, and let the places x_i of each message be restricted to the range $0 \le x < M^3 - M^2$. If $x \ge M$, encode it as $x^3 \bmod N$ as before, but if $x < M$ change the encoding to $(x + yM)^3 \bmod N$, where y is a random number in the range $M^2 - M \le y \le M^2$. To decode, first take the cube root; and if the result is $M^3 - M^2$ or more, take the remainder mod M.

34. Let P be the probability that $x^m \bmod p = 1$ and let Q be the probability that $x^m \bmod q = 1$. The probability that $\gcd(x^m - 1, N) = p$ or q is $P(1-Q) + Q(1-P) = P + Q - 2PQ$. If $P \le \frac{1}{2}$ or $Q \le \frac{1}{2}$, this probability is $\ge \frac{1}{2}\max(P, Q) \ge \frac{1}{2}10^{-6}$, so we have a good chance of finding a factor after about $10^6 \log m$ arithmetic operations modulo N. On the other hand if $P > \frac{1}{2}$ and $Q > \frac{1}{2}$ then $P \approx Q \approx 1$, since we have the general formula $P = \gcd(m, p-1)/p$; thus m is a multiple of $\operatorname{lcm}(p-1, q-1)$ in this case. Let $m = 2^k r$ where r is odd, and form the sequence $x^r \bmod N$, $x^{2r} \bmod N$, $\ldots$, $x^{2^k r} \bmod N$; with high probability we find that the first appearance of 1 is preceded by a value y other than $N - 1$, as in Algorithm P, hence $\gcd(y - 1, N) = p$ or q.

35. Let $f = (p^{q-1} - q^{p-1}) \bmod N$. Since $p \bmod 4 = q \bmod 4 = 3$, we have $(\frac{-1}{p}) = (\frac{-1}{q}) = (\frac{f}{p}) = -(\frac{f}{q}) = -1$, and we also have $(\frac{2}{p}) = -(\frac{2}{q}) = 1$. Given a message x in the range $0 \le x \le \frac{1}{8}(N - 5)$, let $\bar{x} = 4x + 2$ or $8x + 4$, whichever satisfies $(\frac{\bar{x}}{N}) = 1$; then transmit the message $\bar{x}^2 \bmod N$.

To decode this message, we first use a SQRT box to find the unique number y such that $y^2 \equiv \bar{x}^2 \bmod N$ and $(\frac{y}{N}) = 1$ and y is even. Then $y = \bar{x}$, since the other three square roots of $\bar{x}^2$ are $N - \bar{x}$ and $(\pm f\bar{x}) \bmod N$; the first of these is odd, and the other two have negative Jacobi symbols. The decoding is now completed by setting $x = \lfloor y/4 \rfloor$ if $y \bmod 4 = 2$, otherwise $x \leftarrow \lfloor y/8 \rfloor$.

Anybody who can decode such encodings can also find the factors of N, because the decoding of a false message $\bar{x}^2 \bmod N$ when $(\frac{\bar{x}}{N}) = -1$ reveals $(\pm f) \bmod N$, a number that has a nontrivial gcd with N.

36. The mth prime equals

$$m \ln m + m \ln \ln m - m + m \ln \ln m / \ln m - 2m/\ln m + O(m(\log\log m)^2 (\log m)^{-2}),$$

by (4), although for this problem we need only the weaker estimate $p_m = m \ln m + O(m \log \log m)$. (We will assume that p_m is the mth prime, since this corresponds to the assumption that V is uniformly distributed.) If we choose $\ln m = \frac{1}{2}c\sqrt{\ln N \ln \ln N}$, where $c = O(1)$, we find that $r = c^{-1}\sqrt{\ln N / \ln \ln N} - c^{-2} - c^{-2}(\ln \ln \ln N / \ln \ln N) - 2c^{-2}(\ln \frac{1}{2}c)/\ln \ln N + O(\sqrt{\ln \ln N / \ln N})$. The estimated running time (21) now simplifies somewhat surprisingly to the formula $\exp(f(c, N)\sqrt{\ln N \ln \ln N} + O(\log \log N))$, where we have $f(c, N) = c + (1 - (1 + \ln 2)/\ln \ln N)c^{-1}$. The value of c that minimizes $f(c, N)$ is $\sqrt{1 - (1 + \ln 2)/\ln \ln N}$, so we obtain the estimate

$$\exp(2\sqrt{\ln N \ln \ln N}\sqrt{1 - (1 + \ln 2)/\ln \ln N} + O(\log \log N)).$$

When $N = 10^{50}$ this gives $\epsilon(N) \approx .33$, which is still much larger than the observed behavior.

Note: The partial quotients of $\sqrt{D}$ seem to behave according to the distribution obtained for random real numbers in Section 4.5.3. For example, the first million partial quotients of the number $10^{18} + 314159$ include exactly (415236, 169719, 93180, 58606)

cases where A_n is respectively $(1, 2, 3, 4)$. Since V_n lies between $(\sqrt{D}+1)/(A_n+1)$ and $2\sqrt{D}/A_n$, it is likely that $V_n \le 2\sqrt{D}y$ with probability about $\lg(1+y)$. This is not much different from a uniform distribution, so something besides the size of V_n must account for the unreasonable effectiveness of Algorithm E.

37. Apply exercise 4.5.3–12 to the number $\sqrt{D}+R$, to see that the periodic part begins immediately, and run the period backwards to verify the palindromic property. [It follows that the second half of the period gives the same V's as the first, and Algorithm E could be shut down earlier by terminating it when $U = U'$ or $V = V'$ in step E5. However, the period is generally so long, we never even get close to halfway through it, so there is no point in making the algorithm more complicated.]

38. [*Inf. Proc. Letters* **8** (1979), 28–31.] Note that $x \bmod y = x - y\lfloor x/y \rfloor$ can be computed easily on such a machine, and we can get simple constants like $0 = x - x$, $1 = \lfloor x/x \rfloor$, $2 = 1+1$; we can test $x > 0$ by testing whether $x = 1$ or $\lfloor x/(x-1) \rfloor \neq 0$.

(a) First compute $l = \lfloor \lg n \rfloor$ in $O(\log n)$ steps, by repeatedly dividing by 2; at the same time compute $k = 2^l$ and $A \leftarrow 2^{2^{l+1}}$ in $O(\log n)$ steps by repeatedly setting $k \leftarrow 2k$, $A \leftarrow A^2$. For the main computation, suppose we know that $t = A^m$, $u = (A+1)^m$, and $v = m!$; then we can increase the value of m by 1 by setting $m \leftarrow m+1$, $t \leftarrow At$, $u \leftarrow (A+1)u$, $v \leftarrow vm$; and we can *double* the value of m by setting $m \leftarrow 2m$, $u \leftarrow u^2$, $v \leftarrow (\lfloor u/t \rfloor \bmod A)v^2$, $t \leftarrow t^2$, provided that A is sufficiently large. (Consider the number u in radix-A notation; A must be greater than $\binom{2m}{m}$.) Now if $n = (a_l \ldots a_0)_2$, let $n_j = (a_l \ldots a_j)_2$; if $m = n_j$ and $k = 2^j$ and $j > 0$ we can decrease j by 1 by setting $k \leftarrow \lfloor k/2 \rfloor$, $m \leftarrow 2m + (\lfloor n/k \rfloor \bmod 2)$. Hence we can compute $n_j!$ for $j = l, l-1, \ldots, 0$ in $O(\log n)$ steps. [Another solution, due to Julia Robinson, is to compute $n! = \lfloor B^n / \binom{B}{n} \rfloor$ when $B > (2n)^{n+1}$; cf. *AMM* **80** (1973), 250–251, 266.]

(b) First compute $A = 2^{2^{l+2}}$ as in (a), then find the least $k \ge 0$ such that $2^{k+1}! \bmod n = 0$. If $\gcd(n, 2^k!) \neq 1$, let $f(n)$ be this value; note that this gcd can be computed in $O(\log n)$ steps by Euclid's algorithm. Otherwise we will find the least integer m such that $\binom{m}{\lfloor m/2 \rfloor} \bmod n = 0$, and let $f(n) = \gcd(m, n)$. (Note that in this case $2^k < m \le 2^{k+1}$, hence $\lceil m/2 \rceil \le 2^k$ and $\lceil m/2 \rceil !$ is relatively prime to n; therefore $\binom{m}{\lfloor m/2 \rfloor} \bmod n = 0$ iff $m! \bmod n = 0$. Furthermore $n \neq 4$.)

To compute m with a bounded number of registers, we can use Fibonacci numbers (cf. Algorithm 6.2.1F). Suppose we know that $s = F_j$, $s' = F_{j+1}$, $t = A^{F_j}$, $t' = A^{F_{j+1}}$, $u = (A+1)^{2F_j}$, $u' = (A+1)^{2F_{j+1}}$, $v = A^m$, $w = (A+1)^{2m}$, $\binom{2m}{m} \bmod n \neq 0$, and $\binom{2(m+s)}{m+s} = 0$. It is easy to reach this state of affairs with $m = F_{j+1}$, for suitably large j, in $O(\log n)$ steps; furthermore A will be larger than $2^{2(m+s)}$. If $s = 1$, we set $f(n) = \gcd(2m+1, n)$ or $\gcd(2m+2, n)$, whichever is $\neq 1$, and terminate the algorithm. Otherwise we reduce j by 1 as follows: Set $r \leftarrow s$, $s \leftarrow s' - s$, $s \leftarrow r$, $r \leftarrow t$, $t \leftarrow \lfloor t'/t \rfloor$, $t' \leftarrow r$, $r \leftarrow u$, $u \leftarrow \lfloor u'/u \rfloor$, $u' \leftarrow r$; then if $(\lfloor wu/vt \rfloor \bmod A) \bmod n \neq 0$, set $m \leftarrow m+s$, $w \leftarrow wu$, $v \leftarrow vt$.

[Can this problem be solved with fewer than $O(\log n)$ operations? Can the smallest, or the largest, prime factor of n be computed in $O(\log n)$ operations?]

SECTION 4.6

1. $9x^2 + 7x + 9$; $5x^3 + 7x^2 + 2x + 6$.

2. (a) True. (b) False if the algebraic system S contains "zero divisors," nonzero numbers whose product is zero, as in exercise 1; otherwise true. (c) True when $m \neq n$, but false in general when $m = n$, since the leading coefficients might cancel.

3. Assume that $r \leq s$. For $0 \leq k \leq r$ the maximum is $m_1m_2(k+1)$; for $r \leq k \leq s$ it is $m_1m_2(r+1)$; for $s \leq k \leq r+s$ it is $m_1m_2(r+s+1-k)$. The least upper bound valid for all k is $m_1m_2(r+1)$. (The solver of this exercise will know how to factor the polynomial $x^7 + 2x^6 + 3x^5 + 3x^4 + 3x^3 + 3x^2 + 2x + 1$.)

4. If one of the polynomials has fewer than 2^t nonzero coefficients, the product can be formed by putting exactly $t-1$ zeros between each of the coefficients, then multiplying in the binary number system, and finally using a logical **AND** instruction (present on most binary computers, cf. Section 4.5.4) to zero out the extra bits. For example, if $t = 3$, the multiplication in the text would become $(1001000001)_2 \times (1000001001)_2 = (1001001011001001001)_2$; if we **AND** this result with the constant $(1001001\ldots1001)_2$, the desired answer is obtained. A similar technique can be used to multiply polynomials with nonnegative coefficients, when it is known that the coefficients will not be too large.

5. Polynomials of degree $\leq 2n$ can be represented as $U_1(x)x^n + U_0(x)$ where $\deg(U_1)$ and $\deg(U_0) \leq n$; and $(U_1(x)x^n + U_0(x))(V_1(x)x^n + V_0(x)) = U_1(x)V_1(x)(x^{2n} + x^n) + (U_1(x) + U_0(x))(V_1(x) + V_0(x))x^n + U_0(x)V_0(x)(x^n + 1)$. (This equation assumes that arithmetic is being done modulo 2.) Thus Eqs. 4.3.3–3, 4, 5 hold.

Notes: S. A. Cook has shown that Algorithm 4.3.3C can be extended in a similar way, and exercise 4.6.4–57 describes a method requiring even fewer operations for large n. But these ideas are not useful for "sparse" polynomials (having mostly zero coefficients).

SECTION 4.6.1

1. $q(x) = 1 \cdot 2^3x^3 + 0 \cdot 2^2x^2 - 2 \cdot 2x + 8 = 8x^3 - 4x + 8$; $r(x) = 28x^2 + 4x + 8$.

2. The monic sequence of polynomials produced during Euclid's algorithm has the coefficients $(1, 5, 6, 6, 1, 6, 3)$, $(1, 2, 5, 2, 2, 4, 5)$, $(1, 5, 6, 2, 3, 4)$, $(1, 3, 4, 6)$, 0. Hence the greatest common divisor is $x^3 + 3x^2 + 4x + 6$. (The greatest common divisor of a polynomial and its reverse is always symmetric, in the sense that it is a unit multiple of its own reverse.)

3. The procedure of Algorithm 4.5.2X is valid, with polynomials over S substituted for integers. When the algorithm terminates, we have $U(x) = u_2(x)$, $V(x) = u_1(x)$. Let $m = \deg(u)$, $n = \deg(v)$. It is easy to prove by induction that $\deg(u_3) + \deg(v_1) = n$, $\deg(u_3) + \deg(v_2) = m$, after step X3, throughout the execution of the algorithm, provided that $m \geq n$. Hence if m and n are greater than $d = \deg(\gcd(u, v))$ we have $\deg(U) < m - d$, $\deg(V) < n - d$; the exact degrees are $m - d_1$ and $n - d_1$, where d_1 is the degree of the second-last nonzero remainder. If $d = \min(m, n)$, say $d = n$, we have $U(x) = 0$ and $V(x) = 1$.

When $u(x) = x^m - 1$ and $v(x) = x^n - 1$, the identity $(x^m - 1) \bmod (x^n - 1) = x^{m \bmod n} - 1$ shows that all polynomials occurring during the calculation are monic, with integer coefficients. When $u(x) = x^{21} - 1$ and $v(x) = x^{13} - 1$, we have $V(x) = x^{11} + x^8 + x^6 + x^3 + 1$ and $U(x) = -(x^{19} + x^{16} + x^{14} + x^{11} + x^8 + x^6 + x^3 + x)$. [See also Eq. 3.3.3–29, which gives an alternative formula for $U(x)$ and $V(x)$.]

4. Since the quotient $q(x)$ depends only on $v(x)$ and the first $m - n$ coefficients of $u(x)$, the remainder $r(x) = u(x) - q(x)v(x)$ is uniformly distributed and independent of $v(x)$. Hence each step of the algorithm may be regarded as independent of the others; this algorithm is much more well-behaved than Euclid's algorithm over the integers.

The probability that $n_1 = n - k$ is $p^{1-k}(1 - 1/p)$, and $t = 0$ with probability p^{-n}. Each succeeding step has essentially the same behavior; hence we can see that any given sequence of degrees $n, n_1, \ldots, n_t, -\infty$ occurs with probability $(p-1)^t/p^n$. To find the average value of $f(n_1, \ldots, n_t)$, let S_t be the sum of $f(n_1, \ldots, n_t)$ over all sequences $n > n_1 > \cdots > n_t \geq 0$ having a given value of t; then the average is $\sum_t S_t(p-1)^t/p^n$.

Let $f(n_1, \ldots, n_t) = t$; then $S_t = \binom{n}{t}(t+1)$, so the average is $n(1 - 1/p)$. Similarly, if $f(n_1, \ldots, n_t) = n_1 + \cdots + n_t$, then $S_t = \binom{n}{2}\binom{n-1}{t-1}$, and the average is $\binom{n}{2}(1 - 1/p)$. Finally, if $f(n_1, \ldots, n_t) = (n - n_1)n_1 + \cdots + (n_{t-1} - n_t)n_t$, then $S_t = \binom{n+2}{t+2} - (n+1)\binom{n+1}{t+1} + \binom{n+1}{2}\binom{n}{t}$, and the average is $\binom{n+1}{2} - (n+1)p/(p-1) + (p/(p-1))^2(1 - 1/p^{n+1})$.

As a consequence we can see that if p is large there is very high probability that $n_{j+1} = n_j - 1$ for all j. (If this condition fails over the rational numbers, it fails for all p, so we have further evidence for the text's claim that Algorithm C almost always finds $\delta_2 = \cdots = 1$.)

5. Using the formulas developed in exercise 4, with $f(n_1, \ldots, n_t) = \delta_{n_t 0}$, we find that the probability is $1 - 1/p$ if $n > 0$, 1 if $n = 0$.

6. Assuming that the constant terms $u(0)$ and $v(0)$ are nonzero, imagine a "right-to-left" division algorithm, $u(x) = v(x)q(x) + x^{m-n}r(x)$, where $\deg(r) < \deg(v)$. We obtain a gcd algorithm analogous to Algorithm 4.5.2B, which is essentially Euclid's algorithm applied to the "reverse" of the original inputs (cf. exercise 2), afterwards reversing the answer and multiplying by an appropriate power of x.

7. The units of S (as polynomials of degree zero).

8. If $u(x) = v(x)w(x)$, where $u(x)$ has integer coefficients while $v(x)$ and $w(x)$ have rational coefficients, there are integers m and n such that $m \cdot v(x)$ and $n \cdot w(x)$ have integer coefficients. Now $u(x)$ is primitive, so we have

$$u(x) = \pm\, \mathrm{pp}(m \cdot v(x))\, \mathrm{pp}(n \cdot w(x)).$$

9. We can extend Algorithm E as follows: Let $(u_1(x), u_2(x), u_3, u_4(x))$ and $(v_1(x), v_2(x), v_3, v_4(x))$ be quadruples that satisfy the relations $u_1(x)u(x) + u_2(x)v(x) = u_3u_4(x)$, $v_1(x)u(x) + v_2(x)v(x) = v_3v_4(x)$. The extended algorithm starts with the quadruples $(1, 0, \mathrm{cont}(u), \mathrm{pp}(u(x)))$ and $(0, 1, \mathrm{cont}(v), \mathrm{pp}(v(x)))$ and manipulates them in such a way as to preserve the above conditions, where $u_4(x)$ and $v_4(x)$ run through the same sequence as $u(x)$ and $v(x)$ do in Algorithm E. If $au_4(x) = q(x)v_4(x) + br(x)$, we have $av_3(u_1(x), u_2(x)) - q(x)u_3(v_1(x), v_2(x)) = (r_1(x), r_2(x))$, where $r_1(x)u(x) + r_2(x)v(x) = bu_3v_3r(x)$, so the extended algorithm can preserve the desired relations. If $u(x)$ and $v(x)$ are relatively prime, the extended algorithm eventually finds $r(x)$ of degree zero, and we obtain $U(x) = r_2(x)$, $V(x) = r_1(x)$ as desired. (In practice we would divide $r_1(x)$, $r_2(x)$, and bu_3v_3 by $\gcd(\mathrm{cont}(r_1), \mathrm{cont}(r_2))$.) Conversely, if such $U(x)$ and $V(x)$ exist, then $u(x)$ and $v(x)$ have no common prime divisors, since they are primitive and have no common divisors of positive degree.

10. By successively factoring reducible polynomials into polynomials of smaller degree, we must obtain a finite factorization of any polynomial into irreducibles. The factorization of the *content* is unique. To show that there is at most one factorization of the primitive part, the key result is to prove that if $u(x)$ is an irreducible factor of $v(x)w(x)$, but not a unit multiple of the irreducible polynomial $v(x)$, then $u(x)$ is a factor of $w(x)$. This can be proved by observing that $u(x)$ is a factor of $v(x)w(x)U(x) = rw(x) - w(x)u(x)V(x)$ by the result of exercise 9, where r is a nonzero constant.

11. The only row names needed would be A_1, A_0, B_4, B_3, B_2, B_1, B_0, C_1, C_0, D_0. In general, let $u_{j+2}(x) = 0$; then the rows needed for the proof are $A_{n_2-n_j}$ through A_0, $B_{n_1-n_j}$ through B_0, $C_{n_2-n_j}$ through C_0, $D_{n_3-n_j}$ through D_0, etc.

12. If $n_k = 0$, the text's proof of (24) shows that the value of the determinant is $\pm h_k$, and this equals $\pm \ell_k^{n_k-1} / \prod_{1<j<k} \ell_j^{\delta_{j-1}(\delta_j-1)}$. If the polynomials have a factor of positive degree, we can artificially assume that the polynomial zero has degree zero and use the same formula with $\ell_k = 0$.

Notes: The value $R(u, v)$ of Sylvester's determinant is called the *resultant* of u and v, and the quantity $(-1)^{\deg(u)(\deg(u)-1)/2}\ell(u)^{-1}R(u, u')$ is called the *discriminant* of u, where u' is the derivative of u. If $u(x)$ has the factored form $a(x-\alpha_1)\dots(x-\alpha_m)$, and if $v(x) = b(x-\beta_1)\dots(x-\beta_n)$, the resultant $R(u, v)$ is $a^n v(\alpha_1)\dots v(\alpha_m) = (-1)^{mn}b^m u(\beta_1)\dots u(\beta_n) = a^n b^m \prod_{1\le i\le m, 1\le j\le n}(\alpha_i - \beta_j)$. It follows that the polynomials of degree mn in y defined as the respective resultants with $v(x)$ of $u(y-x)$, $u(y+x)$, $x^m u(y/x)$, and $u(yx)$ have as respective roots the sums $\alpha_i + \beta_j$, differences $\alpha_i - \beta_j$, products $\alpha_i\beta_j$, and quotients α_i/β_j (when $v(0) \neq 0$). This idea has been used by R. G. K. Loos to construct algorithms for arithmetic on algebraic numbers [*SIAM J. Computing*, to appear].

If we replace each row A_i in Sylvester's matrix by

$$(b_0 A_i + b_1 A_{i+1} + \cdots + b_{n_2-1-i}A_{n_2-1}) - (a_0 B_i + a_1 B_{i+1} + \cdots + a_{n_2-1-i}B_{n_2-1}),$$

and then delete rows B_{n_2-1} through B_0 and the last n_2 columns, we obtain an $n_1 \times n_1$ determinant for the resultant instead of the original $(n_1+n_2)\times(n_1+n_2)$ determinant. In some cases the resultant can be evaluated efficiently by means of this determinant; see *CACM* **12** (1969), 23–30, 302–303.

J. T. Schwartz has shown that it is possible to evaluate resultants and Sturm sequences for polynomials of degree n with a total of $O(n(\log n)^2)$ arithmetic operations as $n \to \infty$. [*JACM*, to appear.]

13. One can show by induction on j that the values of $(u_{j+1}(x), g_{j+1}, h_j)$ are replaced respectively by $(\ell^{1+p_j} w(x)u_j(x), \ell^{2+p_j} g_j, \ell^{p_j} h_j)$ for $j \ge 2$, where $p_j = n_1 + n_2 - 2n_j$. [In spite of this growth, the bound (26) remains valid.]

14. Let p be a prime of the domain, and let j, k be maximum such that $p^k \setminus v_n = \ell(v)$, $p^j \setminus v_{n-1}$. Let $P = p^k$. By Algorithm R we may write $q(x) = a_0 + Pa_1 x + \cdots + P^s a_s x^s$, where $s = m - n \ge 2$. Let us look at the coefficients of x^{n+1}, x^n, and x^{n-1} in $v(x)q(x)$, namely $Pa_1 v_n + P^2 a_2 v_{n-1} + \cdots$, $a_0 v_n + Pa_1 v_{n-1} + \cdots$, and $a_0 v_{n-1} + Pa_1 v_{n-2} + \cdots$, each of which is a multiple of P^3. We conclude from the first that $p^j \setminus a_1$, from the second that $p^{\min(k,2j)} \setminus a_0$, then from the third that $P \setminus a_0$. Hence $P \setminus r(x)$. [If m were only $n+1$, the best we could prove would be that $p^{\lceil k/2 \rceil}$ divides $r(x)$; e.g., consider $u(x) = x^3 + 1$, $v(x) = 4x^2 + 2x + 1$, $r(x) = 18$. On the other hand, an argument based on determinants of matrices like (21) and (22) can be used to show that $\ell(r)^{\deg(v)-\deg(r)-1} r(x)$ is always a multiple of $\ell(v)^{(\deg(u)-\deg(v))(\deg(v)-\deg(r)-1)}$.]

15. Let $c_{ij} = a_{i1}a_{j1} + \cdots + a_{in}a_{jn}$; we may assume that $c_{ii} > 0$ for all i. If $c_{ij} \neq 0$ for some $i \neq j$, we can replace row i by $(a_{i1} - ca_{j1}, \ldots, a_{in} - ca_{jn})$, where $c = c_{ij}/c_{jj}$; this does not change the value of the determinant, and it decreases the value of the upper bound we wish to prove, since c_{ii} is replaced by $c_{ii} - c_{ij}^2/c_{ij}$. These replacements can be done in a systematic way for increasing i and for $j < i$, until $c_{ij} = 0$ for all $i \neq j$. [The latter algorithm is called "Schmidt's orthogonalization process"; see *Math. Annalen* **63** (1907), 442.] Then $\det(A)^2 = \det(AA^T) = c_{11} \ldots c_{nn}$.

16. Let $f(x_1, \ldots, x_n) = g_m(x_2, \ldots, x_n)x_1^m + \cdots + g_0(x_2, \ldots, x_n)$ and $g(x_2, \ldots, x_n) = g_m(x_2, \ldots, x_n)^2 + \cdots + g_0(x_2, \ldots, x_n)^2$; the latter is not identically zero. We have

$$a_N \leq m(2N+1)^{n-1} + (2N+1)b_N,$$

where b_N counts the integer solutions of $g(x_2, \ldots, x_n) = 0$ with variables bounded by N. Hence $\lim_{N\to\infty} a_N/(2N+1)^n = \lim_{N\to\infty} b_N/(2N+1)^{n-1}$, and this is zero by induction.

17. (a) For convenience, let us describe the algorithm only for $A = \{a, b\}$. The hypotheses imply that $\deg(Q_1U) = \deg(Q_2V) \geq 0$, $\deg(Q_1) \leq \deg(Q_2)$. If $\deg(Q_1) = 0$, then Q_1 is just a nonzero rational number, so we set $Q = Q_2/Q_1$. Otherwise we let $Q_1 = aQ_{11} + bQ_{12} + r_1$, $Q_2 = aQ_{21} + bQ_{22} + r_2$, where r_1 and r_2 are rational numbers; it follows that

$$Q_1U - Q_2V = a(Q_{11}U - Q_{21}V) + b(Q_{12}U - Q_{22}V) + r_1U - r_2V.$$

We must have either $\deg(Q_{11}) = \deg(Q_1) - 1$ or $\deg(Q_{12}) = \deg(Q_1) - 1$. In the former case, $\deg(Q_{11}U - Q_{21}V) < \deg(Q_{11}U)$, by considering the terms of highest degree that start with a; so we may replace Q_1 by Q_{11}, Q_2 by Q_{21}, and repeat the process. Similarly in the latter case, we may replace (Q_1, Q_2) by (Q_{12}, Q_{22}) and repeat the process.

(b) We may assume that $\deg(U) \geq \deg(V)$. If $\deg(R) \geq \deg(V)$, note that $Q_1U - Q_2V = Q_1R - (Q_2 - Q_1Q)V$ has degree less than $\deg(V) \leq \deg(Q_1R)$, so we can repeat the process with U replaced by R; we obtain $R = Q'V + R'$, $U = (Q + Q')V + R'$, where $\deg(R') < \deg(R)$, so eventually a solution will be obtained.

(c) The algorithm of (b) gives $V_1 = UV_2 + R$, $\deg(R) < \deg(V_2)$; by homogeneity, $R = 0$ and U is homogeneous.

(d) We may assume that $\deg(V) \leq \deg(U)$. If $\deg(V) = 0$, set $W \leftarrow U$; otherwise use (c) to find $U = QV$, so that $QVV = VQV$, $(QV - VQ)V = 0$. This implies that $QV = VQ$, so we can set $U \leftarrow V$, $V \leftarrow Q$ and repeat the process.

For further details about the subject of this exercise, see P. M. Cohn, *Proc. Cambridge Phil. Soc.* **57** (1961), 18–30. The considerably more difficult problem of characterizing *all* string polynomials such that $UV = VU$ has been solved by G. M. Bergman [Ph.D. thesis, Harvard University, 1967].

18. [P. M. Cohn, *Transactions Amer. Math. Soc.* **109** (1963), 332–356.]

C1. Set $u_1 \leftarrow U_1$, $u_2 \leftarrow U_2$, $v_1 \leftarrow V_1$, $v_2 \leftarrow V_2$, $z_1 \leftarrow z_2' \leftarrow w_1 \leftarrow w_2' \leftarrow 1$, $z_1' \leftarrow z_2 \leftarrow w_1' \leftarrow w_2 \leftarrow 0$, $n \leftarrow 0$.

C2. (At this point the identities given in the exercise hold, and $u_1v_1 = u_1v_2$; $v_2 = 0$ if and only if $u_1 = 0$.) If $v_2 = 0$, the algorithm terminates with $\operatorname{gcrd}(V_1, V_2) = v_1$, $\operatorname{lclm}(V_1, V_2) = z_1'V_1 = -z_2'V_2$. (Also, by symmetry, we have $\operatorname{gcld}(U_1, U_2) = u_2$ and $\operatorname{lcrm}(U_1, U_2) = U_1w_1 = -U_2w_2$.)

C3. Find Q and R such that $v_1 = Qv_2 + R$, where $\deg(R) < \deg(v_2)$. (We have $u_1(Qv_2 + R) = u_2v_2$, so $u_1R = (u_2 - u_1Q)v_2 = R'v_2$.)

C4. Set $(w_1, w_2, w_1', w_2', z_1, z_2, z_1', z_2', u_1, u_2, v_1, v_2) \leftarrow (w_1' - w_1Q, w_2' - w_2Q, w_1, w_2, z_1', z_2', z_1 - Qz_1', z_2 - Qz_2', u_2 - u_1Q, u_1, v_2, v_1 - Qv_2)$ and $n \leftarrow n + 1$. Go back to C2. ∎

This extension of Euclid's algorithm includes most of the features we have seen in previous extensions, all at the same time, so it provides new insight into the special cases already considered. To prove that it is valid, note first that $\deg(v_2)$ decreases in step C4, so the algorithm certainly terminates. At the conclusion of the algorithm, v_1 is a common right divisor of V_1 and V_2, since $w_1v_1 = (-1)^nV_1$ and $-w_2v_1 = (-1)^nV_2$; also if d is any common right divisor of V_1 and V_2, it is a right divisor of $z_1V_1 + z_2V_2 = v_1$. Hence $v_1 = \gcrd(V_1, V_2)$. Also if m is any common left multiple of V_1 and V_2, we may assume without loss of generality that $m = U_1V_1 = U_2V_2$, since the sequence of values of Q does not depend on U_1 and U_2. Hence $m = (-1)^n(-u_2z_1')V_1 = (-1)^n(u_2z_2')V_2$ is a multiple of $z_1'V_1$.

In practice, if we just want to calculate $\gcrd(V_1, V_2)$, we may suppress the computation of n, w_1, w_2, w_1', w_2', z_1, z_2, z_1', z_2'. These additional quantities were added to the algorithm primarily to make its validity more readily established.

Note: Nontrivial factorizations of string polynomials, such as the example given with this exercise, can be found from matrix identities such as

$$\begin{pmatrix} a & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} b & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} c & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & -c \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & -b \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & -a \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$

since these identities hold even when multiplication is not commutative. For example,

$$(abc + a + c)(1 + ba) = (ab + 1)(cba + a + c).$$

(Compare this with the "continuant polynomials" of Section 4.5.3.)

19. [Cf. Eugène Cahen, *Théorie des Nombres* **1** (Paris: A. Hermann & fils, 1914), 336–338.] If such an algorithm exists, D is a gcrd by the argument in exercise 18. Let us regard A and B as a single $2n \times n$ matrix C whose first n rows are those of A, and whose second n rows are those of B. Similarly, P and Q can be combined into a $2n \times n$ matrix R; X and Y can be combined into an $n \times 2n$ matrix Z. The desired conditions now reduce to two equations $C = RD$, $D = ZC$. If we can find a $2n \times 2n$ integer matrix U with determinant ± 1 such that the last n rows of $U^{-1}C$ are all zero, then $R =$ (first n columns of U), $D =$ (first n rows of $U^{-1}C$), $Z =$ (first n rows of U^{-1}) solves the desired conditions. Hence, for example, the following algorithm may be used (with $m = 2n$):

Algorithm T (*Triangularization*). Let C be an $m \times n$ matrix of integers. This algorithm finds $m \times m$ integer matrices U and V such that $UV = I$ and VC is "upper triangular." (The entry in row i and column j of VC is zero if $i > j$.)

T1. [Initialize.] Set $U \leftarrow V \leftarrow I$, the $m \times m$ identity matrix; and set $T \leftarrow C$. (Throughout the algorithm we will have $T = VC$ and $UV = 1$.)

T2. [Iterate on j.] Do step T3 for $j = 1, 2, \ldots, \min(m, n)$, and terminate the algorithm.

T3. [Zero out column j.] Perform the following transformation zero or more times until T_{ij} is zero for all $i > j$: Let T_{kj} be a nonzero element of $\{T_{ij}, T_{(j+1)j}, \ldots, T_{mj}\}$ having the smallest absolute value. Interchange rows k and j of T and of V; interchange columns k and j of U. Then subtract $\lfloor T_{ij}/T_{jj} \rfloor$ times row j from row i, in matrices T and V, and add the same multiple of column i to column j in matrix U, for $j < i \leq m$. ∎

For the stated example, the algorithm yields $\left(\begin{smallmatrix}1&2\\3&4\end{smallmatrix}\right) = \left(\begin{smallmatrix}1&0\\3&2\end{smallmatrix}\right)\left(\begin{smallmatrix}1&2\\0&-1\end{smallmatrix}\right)$, $\left(\begin{smallmatrix}4&3\\2&1\end{smallmatrix}\right) = \left(\begin{smallmatrix}4&5\\2&3\end{smallmatrix}\right)\left(\begin{smallmatrix}1&2\\0&-1\end{smallmatrix}\right)$, $\left(\begin{smallmatrix}1&2\\0&-1\end{smallmatrix}\right) = \left(\begin{smallmatrix}1&0\\2&-2\end{smallmatrix}\right)\left(\begin{smallmatrix}1&2\\3&4\end{smallmatrix}\right) + \left(\begin{smallmatrix}0&0\\1&0\end{smallmatrix}\right)\left(\begin{smallmatrix}4&3\\2&1\end{smallmatrix}\right)$. (Actually *any* matrix with determinant ± 1 would be a gcrd in this particular case.)

20. It may be helpful to consider the construction of exercise 4.6.2–22, with p^m replaced by a small number ϵ.

21. Note that Algorithm R is used only when $m - n \leq 1$; furthermore, the coefficients are bounded by (25) with $m = n$. [The stated formula is, in fact, the execution time observed in practice, not merely an upper bound. For more detailed information see G. E. Collins, *Proc. 1968 Summer Inst. on Symbolic Math. Comp.*, Robert G. Tobey, ed. (IBM Federal Systems Center: June 1969), 195–231.]

22. A sequence of signs cannot contain two consecutive zeros, since $u_{k+1}(x)$ is a nonzero constant in (29). Moreover we cannot have "+, 0, +" or "−, 0, −" as subsequences. The formula $V(u, a) - V(u, b)$ is clearly valid when $b = a$, so we must only verify it as b increases. The polynomials $u_j(x)$ have finitely many roots, and $V(u, b)$ changes only when b encounters or passes such roots. Let x be a root of some (possibly several) u_j. When b increases from $x - \epsilon$ to x, the sign sequence near j goes from "+, $\pm$, −" to "+, 0, −" or from "−, $\pm$, +" to "−, 0, +" if $j > 0$; and from "+, −" to "0, −" or from "−, +" to "0, +" if $j = 0$. (Since $u'(x)$ is the derivative, $u'(x)$ is negative when $u(x)$ is decreasing.) Thus the net change in V is $-\delta_{j0}$. When b increases from x to $x + \epsilon$, a similar argument shows that V remains unchanged.

[L. E. Heindel, *JACM* **18** (1971), 533–548, has applied these ideas to construct algorithms for isolating the real zeros of a given polynomial $u(x)$, in time bounded by a polynomial in $\deg(u)$ and $\log N$, where all coefficients y_j are integers with $|u_j| \leq N$, and all operations are guaranteed to be exact.]

23. If v has $n-1$ real roots occurring between the n real roots of u, then (by considering sign changes) $u(x) \bmod v(x)$ has $n - 2$ real roots lying between the $n - 1$ roots of v.

24. First show that $h_j = g_j^{\delta_{j-1}} g_{j-1}^{\delta_{j-2}(1-\delta_{j-1})} \ldots g_2^{\delta_1(1-\delta_2)\ldots(1-\delta_{j-1})}$. Then show that the exponent of g_2 on the left-hand side of (18) has the form $\delta_2 + \delta_1 x$, where $x = \delta_2 + \cdots + \delta_{j-1} + 1 - \delta_2(\delta_3 + \cdots + \delta_{j-1} + 1) - \delta_3(1 - \delta_2)(\delta_4 + \cdots + \delta_{j-1} + 1) - \cdots - \delta_{j-1}(1 - \delta_2)\ldots(1 - \delta_{j-2})(1)$. But $x = 1$, since it is seen to be independent of δ_{j-1} and we can set $\delta_{j-1} = 0$, etc. A similar derivation works for g_3, g_4, ..., and a simpler derivation works for (23).

25. Each coefficient of $u_j(x)$ can be expressed as a determinant in which one column contains only $\ell(u)$, $\ell(v)$, and zeros. To use this fact, modify Algorithm C as follows: In step C1, set $g \leftarrow \gcd(\ell(u), \ell(v))$ and $h \leftarrow 0$. In step C3, if $h = 0$, set $u(x) \leftarrow v(x)$, $v(x) \leftarrow r(x)/g$, $h \leftarrow \ell(u)^\delta/g$, $g \leftarrow \ell(u)$, and return to C2; otherwise proceed as in the unmodified algorithm. The effect of this new initialization is simply to replace $u_j(x)$ by $u_j(x)/\gcd(\ell(u), \ell(v))$ for all $j \geq 3$; thus, ℓ^{2j-4} will become ℓ^{2j-5} in (28).

26. In fact, even more is true. Note that the algorithm in exercise 3 computes $\pm p_n(x)$ and $\mp q_n(x)$ for $n \geq -1$. Let $e_n = \deg(q_n)$ and $d_n = \deg(p_n u - q_n v)$; we observed in exercise 3 that $d_{n-1} + e_n = \deg(u)$ for $n \geq 0$. We shall prove that the conditions $\deg(q) < e_n$ and $\deg(pu - qv) < d_{n-2}$ imply that $p(x) = c(x)p_{n-1}(x)$ and $q(x) = c(x)q_{n-1}(x)$: Given such p and q, we can find $c(x)$ and $d(x)$ such that $p(x) = c(x)p_{n-1}(x) + d(x)p_n(x)$ and $q(x) = c(x)q_{n-1}(x) + d(x)q_n(x)$, since $p_{n-1}(x)q_n(x) - p_n(x)q_{n-1}(x) = \pm 1$. Hence $pu - qv = c(p_{n-1}u - q_{n-1}v) + d(p_n u - q_n v)$. If $d(x) \neq 0$, we must have $\deg(c) + e_{n-1} = \deg(d) + e_n$, since $\deg(q) < \deg(q_n)$; it follows that $\deg(c) + d_{n-1} > \deg(d) + d_n$, since this is surely true if $d_n = -\infty$ and otherwise we

have $d_{n-1}+e_n = d_n+e_{n+1} > d_n+e_{n-1}$. Therefore $\deg(pu-qv) = \deg(c)+d_{n-1}$; but we have assumed that $\deg(pu-qv) < d_{n-2} = d_{n-1}+e_n-e_{n-1}$, so $\deg(c) < e_n - e_{n-1}$ and $\deg(d) < 0$, a contradiction.

[This result is essentially due to L. Kronecker, *Monatsberichte Königl. Preuß. Akad. Wiss.* (Berlin: 1881), 535–600. It implies the following theorem: "Let $u(x)$ and $v(x)$ be relatively prime polynomials over a field and let $d \le \deg(v) < \deg(u)$. If $q(x)$ is a polynomial of least degree such that there exist polynomials $p(x)$ and $r(x)$ with $p(x)u(x) - q(x)v(x) = r(x)$ and $\deg(r) = d$, then $p(x)/q(x) = p_n(x)/q_n(x)$ for some n." For if $d_{n-2} > d \ge d_{n-1}$, there are solutions $q(x)$ with $\deg(q) = e_{n-1}+d-d_{n-1} < e_n$, and we have proved that all solutions of such low degree have the stated property.]

SECTION 4.6.2

1. By the principle of inclusion and exclusion (Section 1.3.3), the number of polynomials without linear factors is $\sum_{k\le n}\binom{p}{k}p^{n-k}(-1)^k = p^{n-p}(p-1)^p$. The stated probability when $n \ge p$ is therefore $1-(1-1/p)^p$, which is greater than $\frac{1}{2}$. [In fact, the stated probability is greater than $\frac{1}{2}$ for all $n \ge 1$.] The average number of linear factors is p times the average number of times x is a factor, so it is $\frac{p}{p-1}(1-p^{-n}) = 1+p^{-1}+\cdots+p^{1-n}$. [See answer 38 for further comments on the average number of linear factors.]

2. (a) We know that $u(x)$ has a representation as a product of irreducible polynomials; and the leading coefficients of these polynomials must be units, since they divide the leading coefficient of $u(x)$. Therefore we may assume that $u(x)$ has a representation as a product of monic irreducible polynomials $p_1(x)^{e_1}\ldots p_r(x)^{e_r}$, where $p_1(x)$, ..., $p_r(x)$ are distinct. This representation is unique, except for the order of the factors, so the conditions on $u(x)$, $v(x)$, $w(x)$ are satisfied if and only if

$$v(x) = p_1(x)^{\lfloor e_1/2\rfloor}\ldots p_r(x)^{\lfloor e_r/2\rfloor}, \qquad w(x) = p_1(x)^{e_1 \bmod 2}\ldots p_r(x)^{e_r \bmod 2}.$$

(b) The generating function for the number of monic polynomials of degree n is $1+pz+p^2z^2+\cdots = 1/(1-pz)$. The generating function for the number of polynomials of degree n having the form $v(x)^2$, where $v(x)$ is monic, is $1+pz^2+p^2z^4+\cdots = 1/(1-pz^2)$. If the generating function for the number of monic squarefree polynomials of degree n is $q(z)$, then by part (a) we must have $1/(1-pz) = q(z)/(1-pz^2)$. Hence $q(z) = (1-pz^2)/(1-pz) = 1+pz+(p^2-p)z^2+(p^3-p^2)z^3+\cdots$. The answer is p^n-p^{n-1} for $n \ge 2$. [Curiously, this proves that $\gcd(u(x), u'(x)) = 1$ with probability $1-1/p$; it is the same as the probability that $\gcd(u(x), v(x)) = 1$ when $u(x)$ and $v(x)$ are *independent*, by exercise 4.6.1–5.]

Note: By a similar argument, every $u(x)$ has a unique representation $v(x)w(x)^r$, where $v(x)$ is not divisible by the rth power of any irreducible; the number of such monic polynomials $v(x)$ is $p^n - p^{n-r+1}$ for $n \ge r$.

3. Let $u(x) = u_1(x)\ldots u_r(x)$. There is *at most* one such $v(x)$, by the argument of Theorem 4.3.2C. There is *at least* one if, for each j, we can solve the system with $w_j(x) = 1$ and $w_k(x) = 0$ for $k \ne j$. A solution to the latter is $v_1(x)\prod_{k\ne j}u_k(x)$, where $v_1(x)$ and $v_2(x)$ can be found satisfying

$$v_1(x)\textstyle\prod_{k\ne j}u_k(x) + v_2(x)u_j(x) = 1, \qquad \deg(v_1) < \deg(u_j),$$

by the extension of Euclid's algorithm (exercise 4.6.1–3).

4. By unique factorization, we have $(1-pz)^{-1} = \prod_{n\ge1}(1-z^n)^{-a_{np}}$; after taking logarithms, this can be rewritten

$$\sum_{j\ge1} G_p(z^j)/j = \sum_{k,j\ge1} a_{kp} z^{kj}/j = \ln(1/(1-pz)).$$

The stated identity now yields the answer $G_p(z) = \sum_{m\ge1} \mu(m) m^{-1} \ln(1/(1-pz^m))$, from which we obtain $a_{np} = \sum_{d\backslash n} \mu(n/d) n^{-1} p^d$; thus $\lim_{p\to\infty} a_{np}/p^n = 1/n$. To prove the stated identity, note that

$$\sum_{n,j\ge1} \mu(n) g(z^{nj}) n^{-t} j^{-t} = \sum_{m\ge1} g(z^m) m^{-t} \sum_{n\backslash m} \mu(n) = g(z).$$

5. Let a_{npr} be the number of monic polynomials of degree n modulo p having exactly r irreducible factors. Then $\mathcal{G}_p(z,w) = \sum_{n,r\ge0} a_{npr} z^n w^r = \exp(\sum_{k\ge1} G_p(z^k) w^k/k)$, where G_p is the generating function in exercise 4; cf. Eq. 1.2.9–34. We have

$$\begin{aligned}\sum_{n\ge0} A_{np} z^n = d\mathcal{G}_p(z/p,w)/dw\,|_{w=1} &= (\textstyle\sum_{k\ge1} G_p(z^k/p^k)) \exp(\ln(1/(1-z))) \\ &= (\textstyle\sum_{n\ge1} \ln(1/(1-p^{1-n}z^n))\varphi(n)/n)/(1-z),\end{aligned}$$

hence $A_{np} = H_n + 1/2p + O(p^{-2})$ for $n \ge 2$. The average value of 2^r is the coefficient of z^n in $\mathcal{G}_p(z/p, 2)$, namely $n+1+(n-1)/p+O(p^{-2})$. (The variance is of order n^3, however: set $w = 4$.)

6. For $0 \le s < p$, $x - s$ is a factor of $x^p - x$ (modulo p) by Fermat's theorem. So $x^p - x$ is a multiple of $\operatorname{lcm}(x, x-1, \ldots, x-(p-1)) = x^{\underline{p}}$. [*Note:* Therefore the Stirling numbers $\left[{p \atop k}\right]$ are multiples of p except when $k = 1$, $k = p$. Equation 1.2.6–41 shows that the same statement is valid for Stirling numbers $\left\{{p \atop k}\right\}$ of the other kind.]

7. The factors on the right are relatively prime, and each is a divisor of $u(x)$, so their product divides $u(x)$. On the other hand, $u(x)$ divides

$$v(x)^p - v(x) = \prod_{0\le s<p} (v(x) - s),$$

so it divides the right-hand side by exercise 4.5.2–2.

8. The vector (18) is the only output whose kth component is nonzero.

9. For example, start with $x \leftarrow 1$ and $y \leftarrow 1$; then repeatedly set $R[x] \leftarrow y$, $x \leftarrow 2x \bmod 101$, $y \leftarrow 51y \bmod 101$, one hundred times.

10. The matrix $Q - I$ below has a null space generated by the two vectors $v^{[1]} = (1,0,0,0,0,0,0,0)$, $v^{[2]} = (0,1,1,0,0,1,1,1)$. The factorization is

$$(x^6 + x^5 + x^4 + x + 1)(x^2 + x + 1).$$

$p = 2$

$$\begin{pmatrix} 0&0&0&0&0&0&0&0\\ 0&1&1&0&0&0&0&0\\ 0&0&1&0&1&0&0&0\\ 0&0&0&1&0&0&1&0\\ 1&0&0&1&0&0&1&0\\ 1&0&1&1&1&0&0&0\\ 0&0&1&0&1&1&0&1\\ 1&1&0&1&1&1&0&1 \end{pmatrix}$$

$p = 5$

$$\begin{pmatrix} 0&0&0&0&0&0&0\\ 0&4&0&0&0&1&0\\ 0&2&2&0&4&3&4\\ 0&1&4&4&4&2&1\\ 2&2&2&3&4&3&2\\ 0&0&4&0&1&3&2\\ 3&0&2&1&4&2&1 \end{pmatrix}$$

11. Removing the trivial factor x, the matrix $Q - I$ on the previous page has a null space generated by $(1,0,0,0,0,0,0)$ and $(0,3,1,4,1,2,1)$. The factorization is

$$x(x^2 + 3x + 4)(x^5 + 2x^4 + x^3 + 4x^2 + x + 3).$$

12. If $p = 2$, $(x+1)^4 = x^4 + 1$. If $p = 8k+1$, $Q - I$ is the zero matrix, so there are four factors. For other values of p we have

$$\begin{matrix} & p = 8k+3 & & p = 8k+5 & & p = 8k+7 \\ Q - I = & \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -2 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & -1 \\ 0 & 0 & -2 & 0 \\ 0 & -1 & 0 & -1 \end{pmatrix} \end{matrix}$$

Here $Q-I$ has rank 2, so there are $4-2 = 2$ factors. [But it is easy to prove that x^4+1 is irreducible over the integers, since it has no linear factors and the coefficient of x in any factor of degree two must be less than or equal to 2 in absolute value by exercise 20. For all $k \geq 2$, H. P. F. Swinnerton-Dyer has exhibited polynomials of degree 2^k that are irreducible over the integers, but they split completely into linear and quadratic factors modulo every prime. For degree 8, his example is $x^8 - 16x^6 + 88x^4 + 192x^2 + 144$, having roots $\pm\sqrt{2} \pm \sqrt{3} \pm i$ [see *Math. Comp.* **24** (1970), 733–734]. According to the theorem of Frobenius cited in exercise 37, any irreducible polynomial of degree n whose Galois group contains no n-cycles will have factors modulo almost all primes.]

13. $p = 8k+1$: $(x + (1+\sqrt{-1})/\sqrt{2})(x + (1-\sqrt{-1})/\sqrt{2})(x - (1+\sqrt{-1})/\sqrt{2})\times$ $(x - (1-\sqrt{-1})/\sqrt{2})$. $p = 8k+3$: $(x^2 - \sqrt{-2}x - 1)(x^2 - \sqrt{-2}x - 1)$. $p = 8k+5$: $(x^2 - \sqrt{-1})(x^2 - \sqrt{-1})$. $p = 8k+7$: $(x^2 - \sqrt{2}x + 1)(x^2 + \sqrt{2}x + 1)$. The latter factorization also holds over the field of real numbers.

14. Algorithm N can be adapted to find the coefficients of w: Let A be the $(r+1) \times n$ matrix whose kth row contains the coefficients of $v(x)^k \bmod u(x)$, for $0 \leq k \leq r$. Apply the method of Algorithm N until the first dependence is found in step N3; then the algorithm terminates with $w(x) = v_0 + v_1 x + \cdots + v_k x^k$, where v_j is defined in (18). At this point $2 \leq k \leq r$; it is not necessary to know r in advance, since we can check for dependency after generating each row of A.

15. We may assume that $u \neq 0$ and that p is odd. Berlekamp's method applied to the polynomial $x^2 - u$ tells us that a square root exists if and only if $u^{(p-1)/2} \bmod p = 1$; let us assume that this condition holds.

Let $p - 1 = 2^e \cdot q$, where q is odd. Zassenhaus's factoring procedure suggests the following square-root extraction algorithm: Set $t \leftarrow 0$. Evaluate

$$\gcd((x+t)^q - 1, x^2 - u),\ \gcd((x+t)^{2q} - 1, x^2 - u),$$
$$\gcd((x+t)^{4q} - 1, x^2 - u),\ \gcd((x+t)^{8q} - 1, x^2 - u),\ \ldots,$$

until finding the first case where the gcd is not 1 (modulo p). If the gcd is $x - v$, then $\sqrt{u} = \pm v$. If the gcd is $x^2 - u$, set $t \leftarrow t + 1$ and repeat the calculation.

Notes: If $(x+t)^k \bmod (x^2 - u) = ax + b$, then we have $(x+t)^{k+1} \bmod (x^2 - u) = (b + at)x + (bt + au)$, and $(x+t)^{2k} \bmod (x^2 - u) = 2abx + (b^2 + a^2 u)$; hence $(x+t)^q$, $(x+t)^{2q}$, ... are easy to evaluate efficiently, and the calculation for fixed t takes $O((\log p)^3)$ units of time. The square root will be found when $t = 0$ with probability

$1/2^{e-1}$; thus it will always be found immediately when $p \bmod 4 = 3$. If we choose t at random instead of increasing it sequentially, exercise 29 gives a rigorous proof that each t gives success at least about half of the time; but for practical purposes this random choice isn't needed.

Another square-root method has been suggested by D. Shanks. When $e > 1$ it requires an auxiliary constant z (depending only on p) such that $z^{2^{e-1}} \equiv -1$ (modulo p). The value $z = n^q \bmod p$ will work for almost one half of all integers n; once z is known, the following algorithm requires no more probabilistic calculation:

S1. Set $y \leftarrow z$, $r \leftarrow e$, $v \leftarrow u^{(q+1)/2} \bmod p$, $w \leftarrow u^q \bmod p$.

S2. If $w = 1$, stop; v is the answer. Otherwise find the smallest k such that $w^{2^k} \bmod p$ is equal to 1. If $k = r$, stop (there is no answer); otherwise set

$$(y, r, v, w) \leftarrow (y^{2^{r-k}}, k, vy^{2^{r-k-1}}, wy^{2^{r-k}})$$

and repeat step S2. ∎

The validity of this algorithm follows from the invariant congruences $uw \equiv v^2$, $y^{2^{r-1}} \equiv -1$, $w^{2^{r-1}} \equiv 1$ (modulo p). On the average, step S2 will require about $\frac{1}{4}e^2$ multiplications mod p. Reference: *Proc. Second Manitoba Conf. Numer. Math.* (1972), 58–62. A related method was published by A. Tonelli, *Göttinger Nachrichten* (1891), 344–346.

16. (a) Substitute polynomials modulo p for integers, in the proof for $n = 1$. (b) The proof for $n = 1$ carries over to any finite field. (c) Since $x = \xi^k$ for some k, $x^{p^n} = x$ in the field defined by $f(x)$. Furthermore, the elements y that satisfy the equation $y^{p^m} = y$ in the field are closed under addition, and closed under multiplication; so if $x^{p^m} = x$, then ξ (being a polynomial in x with integer coefficients) satisfies $\xi^{p^m} = \xi$.

17. If ξ is a primitive root, each nonzero element is some power of ξ. Hence the order must be a divisor of $13^2 - 1 = 2^3 \cdot 3 \cdot 7$, and $\varphi(f)$ elements have order f.

f	$\varphi(f)$	f	$\varphi(f)$	f	$\varphi(f)$	f	$\varphi(f)$
1	1	3	2	7	6	21	12
2	1	6	2	14	6	42	12
4	2	12	4	28	12	84	24
8	4	24	8	56	24	168	48

18. (a) $\operatorname{pp}(p_1(u_n x)) \ldots \operatorname{pp}(p_r(u_n x))$, by Gauss's lemma. For example, let

$$u(x) = 6x^3 - 3x^2 + 2x - 1, \qquad v(x) = x^3 - 3x^2 + 12x - 36 = (x^2 + 12)(x - 3);$$

then $\operatorname{pp}(36x^2 + 12) = 3x^2 + 1$, $\operatorname{pp}(6x - 3) = 2x - 1$. (This is a modern version of a fourteenth-century trick used for many years to help solve algebraic equations.)

(b) Let $\operatorname{pp}(w(u_n x)) = \bar{w}_m x^m + \cdots + \bar{w}_0 = w(u_n x)/c$, where c is the content of $w(u_n x)$ as a polynomial in x. Then $w(x) = (c\bar{w}/u_n^m)x^m + \cdots + c\bar{w}_0$, hence $c\bar{w}_m = u_n^m$; since $\bar{w}_m$ is a divisor of u_n, c is a multiple of u_n^{m-1}.

19. If $u(x) = v(x)w(x)$ with $\deg(v)\deg(w) \geq 1$, then $u_n x^n \equiv v(x)w(x)$ (modulo p). By unique factorization modulo p, all but the leading coefficients of v and w are multiples of p, and p^2 divides $v_0 w_0 = u_0$.

20. (a) $\sum(\alpha u_j - u_{j-1})(\bar\alpha\bar u_j - \bar u_{j-1}) = \sum(u_j - \bar\alpha u_{j-1})(\bar u_j - \alpha\bar u_{j-1})$. (b) We may assume that $u_0 \neq 0$. Let $m(u) = \prod_{1\le j\le n}\min(1,|\alpha_j|) = u_0/M(u)u_n$. Whenever $|\alpha_j| < 1$, change the factor $x - \alpha_j$ to $\bar\alpha_j x - 1$ in $u(x)$; this doesn't affect $|u|$. Now looking at the leading and trailing coefficients only, we have $|u|^2 \ge |u_n|^2 m(u)^2 + |u_n|^2 M(u)^2$; hence we obtain the slightly stronger result $M(u)^2 \le (|u|^2 + (|u|^4 - 4|u_0u_n|^2)^{1/2})/2|u_n|^2$. [A further improvement in the estimate of $M(u)$ can often be obtained by replacing $u(x)$ by the polynomial $\hat u(x) = (x^k - s_k/|u|^2)u(x)$, where $s_k = \sum_j \bar u_j u_{j-k}$; since $M(\hat u) = M(u)$ and $|\hat u|^2 = |u|^2 - |s_k|^2/|u|^2$, we obtain the inequality $|u|^2 - |s_k|^2/|u|^2 \ge |u_0|^2|s_k|^2/(|u|^4 M(u)^2) + |u_n|^2 M(u)^2$ for $1 \le k < n$. In the case of polynomial (22), we have $s_2 = -72$ and we obtain the bound $M(u) < 8.1837$.] (c) $u_j = u_m \sum \alpha_{i_1}\dots\alpha_{i_{m-j}}$, an elementary symmetric function, hence $|u_j| \le |u_m|\sum \beta_{i_1}\dots\beta_{i_{m-j}}$ where $\beta_i = \max(1,|\alpha_i|)$. We complete the proof by showing that when $x_1 \ge 1, \dots, x_n \ge 1$, and $x_1\dots x_n = M$, the elementary symmetric function $\sigma_{nk} = \sum x_{i_1}\dots x_{i_k}$ is $\le \binom{n-1}{k-1}M + \binom{n-1}{k}$, the value assumed when $x_1 = \cdots = x_{n-1} = 1$ and $x_n = M$. (For if $x_1 \le \cdots \le x_n < M$, the transformation $x_n \leftarrow x_{n-1}x_n$, $x_{n-1} \leftarrow 1$ increases σ_{nk} by $\sigma_{(n-2)(k-1)}(x_n - 1)(x_{n-1} - 1)$, which is positive.) (d) $|v_j| \le |v_m|\big(\binom{m-1}{j}M(v) + \binom{m-1}{j-1}\big) \le |u_n|\big(\binom{m-1}{j}M(u) + \binom{m-1}{j-1}\big)$ since $|v_m| \le |u_n|$ and $M(v) \le M(u)$. [See J. Vicente Gonçalves, *Revista de Faculdade de Ciências* (2) A **1** (Univ. Lisbon, 1950), 167–171; M. Mignotte, *Math. Comp.* **28** (1974), 1153–1157; Mignotte and Payafar, *RAIRO Analyse numér.* **13** (1979), 181–192.]

21. (a) $\int_0^1 (u_n e(n\theta) + \cdots + u_0)(\bar u_n e(-n\theta) + \cdots + \bar u_0)\,d\theta = |u_n|^2 + \cdots + |u_0|^2$ since $\int_0^1 e(j\theta)e(-k\theta)\,d\theta = \delta_{jk}$; now use induction on t. (b) Since $|v_j| \le \binom{m}{j}M(v)|v_m|$ we conclude that $|v|^2 \le \binom{2m}{m}M(v)^2|v_m|^2$. Hence $|v|^2|w|^2 \le \binom{2m}{m}\binom{2k}{k}M(v)^2M(w)^2|v_m w_k|^2 = f(m,k)M(u)^2|u_n|^2 \le f(m,k)|u|^2$. [Slightly better values for $f(m,k)$ are possible based on the more detailed information in exercise 20.] (c) The case $t = 3$ suffices to show how to get from $t-1$ to t. When $t = 2$ we have shown that, for all θ_1,

$$\int_0^1\int_0^1\int_0^1\int_0^1 \big|v(e(\theta_1),e(\phi_2),e(\phi_3))\big|^2\big|w(e(\theta_1),e(\psi_2),e(\psi_3))\big|^2\,d\phi_2\,d\phi_3\,d\psi_2\,d\psi_3$$
$$\le f(m_2,k_2)f(m_3,k_3)\int_0^1\int_0^1 \big|v(e(\theta_1),e(\theta_2),e(\theta_3))\big|^2\big|w(e(\theta_1),e(\theta_2),e(\theta_3))\big|^2\,d\theta_2\,d\theta_3.$$

For all ϕ_2, ϕ_3, ψ_2, ψ_3 we have also shown that

$$\int_0^1\int_0^1 \big|v(e(\phi_1),e(\phi_2),e(\phi_3))\big|^2\big|w(e(\psi_1),e(\psi_2),e(\psi_3))\big|^2\,d\phi_1\,d\psi_1$$
$$\le f(m_1,k_1)\int_0^1 \big|v(e(\theta_1),e(\phi_2),e(\phi_3))\big|^2\big|w(e(\theta_1),e(\psi_2),e(\psi_3))\big|^2\,d\theta_1.$$

Integrate the former inequality with respect to θ_1 and the latter with respect to ϕ_2, ϕ_3, ψ_2, ψ_3. [This method was used by A. O. Gel'fond in *Transcendental and Algebraic Numbers* (New York: Dover, 1960), Section 3.4, to derive a slightly different result.]

22. More generally, assume that $u(x) \equiv v(x)w(x)$ (modulo q), $a(x)v(x) + b(x)w(x) \equiv 1$ (modulo p), and $c\ell(v) \equiv 1$ (modulo r), $\deg(a) < \deg(w)$, $\deg(b) < \deg(v)$, $\deg(u) = \deg(v) + \deg(w)$, where $r = \gcd(p,q)$ and p, q needn't be prime. We shall construct polynomials $V(x) \equiv v(x)$ and $W(x) \equiv w(x)$ (modulo q) such that $u(x) \equiv V(x)W(x)$ (modulo qr), $\ell(V) = \ell(v)$, $\deg(V) = \deg(v)$, $\deg(W) = \deg(w)$; furthermore, if r is prime, the results will be unique modulo qr.

The problem asks us to find $\bar v(x)$ and $\bar w(x)$ with $V(x) = v(x) + q\bar v(x)$, $W(x) = w(x) + q\bar w(x)$, $\deg(\bar v) < \deg(v)$, $\deg(\bar w) \le \deg(w)$; and the other condition

$$(v(x) + q\bar v(x))(w(x) + q\bar w(x)) \equiv u(x) \pmod{qr}$$

is equivalent to $\bar{w}(x)v(x) + \bar{v}(x)w(x) \equiv f(x)$ (modulo r), where $f(x)$ satisfies $u(x) \equiv v(x)w(x) + qf(x)$ (modulo qr). We have

$$(a(x)f(x) + t(x)w(x))v(x) + (b(x)f(x) - t(x)v(x))w(x) \equiv f(x) \pmod{r}$$

for all $t(x)$. Since $\ell(v)$ has an inverse modulo r, we can find a quotient $t(x)$ by Algorithm 4.6.1D such that $\deg(bf - tv) < \deg(v)$; for this $t(x)$, $\deg(af + tw) \leq \deg(w)$, since we have $\deg(f) \leq \deg(u) = \deg(v) + \deg(w)$. Thus the desired solution is $\bar{v}(x) = b(x)f(x) - t(x)v(x) = b(x)f(x) \bmod v(x)$, $\bar{w}(x) = a(x)f(x) + t(x)w(x)$. If $(\bar{\bar{v}}(x), \bar{\bar{w}}(x))$ is another solution, we have $(\bar{w}(x) - \bar{\bar{w}}(x))v(x) \equiv (\bar{\bar{v}}(x) - \bar{v}(x))w(x)$ (modulo r). Thus if r is prime, $v(x)$ must divide $\bar{\bar{v}}(x) - \bar{v}(x)$; but $\deg(\bar{\bar{v}} - \bar{v}) < \deg(v)$, so $\bar{\bar{v}}(x) = \bar{v}(x)$ and $\bar{\bar{w}}(x) = \bar{w}(x)$.

For $p = 2$, the factorization proceeds as follows (writing only the coefficients, and using bars for negative digits): Exercise 10 says that $v_1(x) = (\bar{1}\,\bar{1}\,\bar{1})$, $w_1(x) = (\bar{1}\,\bar{1}\,\bar{1}\,0\,0\,\bar{1}\,\bar{1})$ in one-bit two's complement notation. Euclid's extended algorithm yields $a(x) = (1\,0\,0\,0\,0\,1), b(x) = (1\,0)$. The factor $v(x) = x^2 + c_1x + c_0$ must have $|c_1| \leq \lfloor 1 + \sqrt{113} \rfloor = 11$, $|c_0| \leq 10$, by exercise 20. Three applications of Hensel's lemma yield $v_4(x) = (1\,3\,\bar{1})$, $w_4(x) = (1\,\bar{3}\,\bar{5}\,\bar{4}\,4\,\bar{3}\,5)$. Thus $c_1 \equiv 3$ and $c_0 \equiv -1$ (modulo 16); the only possible quadratic factor of $u(x)$ is $x^2 + 3x - 1$. Division fails, so $u(x)$ is irreducible. (Since we have now proved the irreducibility of this beloved polynomial by four separate methods, it is unlikely that it has any factors.)

Hans Zassenhaus has observed that we can often speed up such calculations by increasing p as well as q: In the above notation, we can find $A(x)$, $B(x)$ such that $A(x)V(x) + B(x)W(x) \equiv 1$ (modulo pr), namely by taking $A(x) = a(x) + p\bar{a}(x)$, $B(x) = b(x) + p\bar{b}(x)$, where $\bar{a}(x)V(x) + \bar{b}(x)W(x) \equiv g(x)$ (modulo r), $a(x)V(x) + b(x)W(x) \equiv 1 - pg(x)$ (modulo pr). We can also find C with $\ell(V)C \equiv 1$ (modulo pr). In this way we can lift a squarefree factorization $u(x) \equiv v(x)w(x)$ (modulo p) to its unique extensions modulo p^2, p^4, p^8, p^{16}, etc. However, this "accelerated" procedure reaches a point of diminishing returns in practice, as soon as we get to double-precision moduli, since the time for multiplying multiprecision numbers in practical ranges outweighs the advantage of squaring the modulus directly. From a computational standpoint it seems best to work with the successive moduli p, p^2, p^4, p^8, ..., p^E, p^{E+e}, p^{E+2e}, p^{E+3e}, ..., where E is the smallest power of 2 with p^E greater than single precision and e is the largest integer such that p^e has single precision.

Hensel's lemma, which K. Hensel introduced in order to demonstrate the factorization of polynomials over the field of p-adic numbers (see exercise 4.1–31), can be generalized in several ways. First, if there are more factors, say $u(x) \equiv v_1(x)v_2(x)v_3(x)$ (modulo p), we can find $a_1(x)$, $a_2(x)$, $a_3(x)$ such that $a_1(x)v_2(x)v_3(x) + a_2(x)v_1(x)v_3(x) + a_3(x)v_1(x)v_2(x) \equiv 1$ (modulo p) and $\deg(a_i) < \deg(v_i)$. (In essence, $1/u(x)$ is expanded in partial fractions as $\sum a_i(x)/v_i(x)$.) An exactly analogous construction now allows us to lift the factorization without changing the leading coefficients of v_1 and v_2; we take $\bar{v}_1(x) = a_1(x)f(x) \bmod v_1(x)$, $\bar{v}_2(x) = a_2(x)f(x) \bmod v_2(x)$, etc. Another important generalization is to several simultaneous moduli, of the respective forms p^e, $(x_2 - a_2)^{n_2}$, ..., $(x_t - a_t)^{n_t}$, when performing multivariate gcds and factorizations. Cf. D. Y. Y. Yun, Ph.D. Thesis (M.I.T., 1974).

23. The discriminant of $\mathrm{pp}(u(x))$ is a nonzero integer (cf. exercise 4.6.1–12), and there are multiple factors modulo p iff p divides the discriminant. [The factorization of (22) modulo 3 is $(x+1)(x^2 - x - 1)^2(x^3 + x^2 - x + 1)$; squared factors for this polynomial occur only for $p = 3$, 23, 233, and 121702457. It is not difficult to prove that the

smallest prime that is not unlucky is at most $O(n \log Nn)$, if $n = \deg(u)$ and N bounds the coefficients of $u(x)$.]

24. Multiply a monic polynomial with rational coefficients by a suitable nonzero integer, to get a primitive polynomial over the integers. Factor this polynomial over the integers, and then convert the factors back to monic. (No factorizations are lost in this way; see exercise 4.6.1–8.)

25. Consideration of the constant term shows there are no factors of degree 1, so if the polynomial is reducible, it must have one factor of degree 2 and one of degree 3. Modulo 2 the factors are $x(x+1)^2(x^2+x+1)$; this is not much help. Modulo 3 the factors are $(x+2)^2(x^3+2x+2)$. Modulo 5 they are $(x^2+x+1)(x^3+4x+2)$. So we see that the answer is $(x^2+x+1)(x^3-x+2)$.

26. Begin with $D \leftarrow (0\ldots01)$, representing the set $\{0\}$. Then for $1 \le j \le r$, set $D \leftarrow D \vee (D \uparrow d_j)$, where $\vee$ denotes logical "or" and $D \uparrow d$ denotes D shifted left d bit positions. (Actually we need only work with a bit vector of length $\lceil (n+1)/2 \rceil$, since $n-m$ is in the set iff m is.)

27. Exercise 4 says that a random polynomial of degree n is irreducible modulo p with rather low probability, about $1/n$. But the Chinese remainder theorem implies that a random monic polynomial of degree n over the integers will be reducible with respect to each of k distinct primes with probability about $(1-1/n)^k$, and this approaches zero as $k \to \infty$. Hence almost all polynomials over the integers are irreducible with respect to infinitely many primes; and almost all primitive polynomials over the integers are irreducible. [Another proof has been given by W. S. Brown, *AMM* **70** (1963), 965–969. See also the generalization cited in the answer to exercise 36.]

28. Cf. exercise 4; the probability is the coefficient of z^n in $(1+a_{1p}z/p)(1+a_{2p}z^2/p^2)\times(1+a_{3p}z^3/p^3)\ldots$, which has the limiting value $g(z) = (1+z)(1+\frac{1}{2}z^2)(1+\frac{1}{3}z^3)\ldots$. For $1 \le n \le 10$ the answers are $1, \frac{1}{2}, \frac{5}{6}, \frac{7}{12}, \frac{37}{60}, \frac{79}{120}, \frac{173}{280}, \frac{101}{168}, \frac{127}{210}, \frac{1033}{1680}$. [Let $f(y) = \ln(1+y) - y = O(y^2)$. We have

$$g(z) = \exp\Big(\sum_{n\ge1} z^n/n + \sum_{n\ge1} f(z^n/n)\Big) = h(z)/(1-z),$$

and it can be shown that the limiting probability is $h(1) = \exp\big(\sum_{n\ge1} f(1/n)\big) = e^{-\gamma} \approx .56146$ as $n \to \infty$; cf. D. H. Lehmer, *Acta Arith.* **21** (1972), 379–388. Indeed, N. G. de Bruijn has established the asymptotic formula $\lim_{p\to\infty} a_{np} = e^{-\gamma} + e^{-\gamma}/n + O(n^{-2}\log n)$.]

29. Let $q_1(x)$ and $q_2(x)$ be any two of the irreducible divisors of $g(x)$. By the Chinese remainder theorem (exercise 3), choosing a random polynomial $t(x)$ of degree $< 2d$ is equivalent to choosing two random polynomials $t_1(x)$ and $t_2(x)$ of degrees $< d$, where $t_i(x) = t(x) \bmod q_i(x)$. The gcd will be a proper factor if $t_1(x)^{(p^d-1)/2} \bmod q_1(x) = 1$ and $t_2(x)^{(p^d-1)/2} \bmod q_1(x) \neq 1$, or vice versa, and this condition holds for exactly $2((p^d-1)/2)((p^d+1)/2) = (p^{2d}-1)/2$ choices of $t_1(x)$ and $t_2(x)$.

Notes: We are considering here only the behavior with respect to two irreducible factors, but the true behavior is probably much better. Suppose that each irreducible factor $q_i(x)$ has probability $\frac{1}{2}$ of dividing $t(x)^{(p^d-1)/2} - 1$ for each $t(x)$, independent of the behavior for other $q_j(x)$ and $t(x)$; and assume that $g(x)$ has r irreducible factors in all. Then if we encode each $q_i(x)$ by a sequence of 0's and 1's according as $q_i(x)$ does or doesn't divide $t(x)^{(p^d-1)/2} - 1$ for the successive t's tried, we obtain a random binary

trie with r lieves (cf. Section 6.3). The cost associated with an internal node of this trie, having m lieves as descendants, is $O(m^2(\log p))$; and the solution to the recurrence $A_n = \binom{n}{2} + 2^{1-n}\sum\binom{n}{k}A_k$ is $A_n = 2\binom{n}{2}$, by exercise 5.2.2–36. Hence the sum of costs in the given random trie—representing the expected time to factor $g(x)$ *completely*—is $O(r^2(\log p)^3)$ under this plausible assumption. The plausible assumption becomes rigorously true if we choose $t(x)$ at random of degree $< rd$ instead of restricting it to degree $< 2d$.

30. Let $T(x) = x + x^p + \cdots + x^{p^{d-1}}$ and $v(x) = T(t(x)) \bmod q(x)$. Since $t(x)^{p^d} = t(x)$ in the field of polynomial remainders modulo $q(x)$, we have $v(x)^p = v(x)$ in that field; in other words, $v(x)$ is one of the p roots of the equation $y^p - y = 0$. Hence $v(x)$ is an integer.

It follows that $\prod_{0\le s<p}\gcd(g_d(x), T(t(x)) - s) = g_d(x)$. In particular, when $p = 2$ we can argue as in exercise 29 that $\gcd(g_d(x), T(t(x)))$ will be a proper factor of $g_d(x)$ with probability $\ge \frac{1}{2}$ when $g_d(x)$ has at least two irreducible factors and $t(x)$ is a random binary polynomial of degree $< 2d$.

[Note that $T(t(x)) \bmod g(x)$ can be computed by starting with $u(x) \leftarrow t(x)$ and setting $u(x) \leftarrow (u(x) + u(x)^p) \bmod g(x)$ repeatedly, $d - 1$ times. The method of this exercise is based on the polynomial factorization $x^{p^d} - x = \prod_{0\le s<p}(T(x) - s)$, which holds for any p, while formula (21) is based on the polynomial factorization $x^{p^d} - x = x(x^{(p^d-1)/2} + 1)(x^{(p^d-1)/2} - 1)$ for odd p.]

31. If α is an element of the field of p^d elements, let $d(\alpha)$ be the "degree" of α, namely the smallest exponent e such that $\alpha^{p^e} = \alpha$. Then

$$P_\alpha(x) = (x - \alpha)(x - \alpha^p)\ldots(x - \alpha^{p^{d-1}}) = q_\alpha(x)^{d/d(\alpha)},$$

where $q_\alpha(x)$ is an irreducible polynomial of degree $d(\alpha)$. As α runs through all elements of the field, the corresponding $q_\alpha(x)$ runs through every irreducible polynomial of degree e dividing d, where every such irreducible occurs exactly e times. We have $(x + t)^{(p^d-1)/2} \bmod q_\alpha(x) = 1$ if and only if $(\alpha + t)^{(p^d-1)/2} = 1$ in the field. If t is an integer, we have $d(\alpha + t) = d(\alpha)$, hence $n(p, d)$ is d^{-1} times the number of elements α of degree d such that $\alpha^{(p^d-1)/2} = 1$. Similarly, if $t_1 \ne t_2$ we want to count the number of elements of degree d such that $(\alpha + t_1)^{(p^d-1)/2} = (\alpha + t_2)^{(p^d-1)/2}$, i.e., $((\alpha + t_1)/(a + t_2))^{(p^d-1)/2} = 1$. As α runs through all the elements of degree d, so does the quantity $(\alpha + t_1)/(\alpha + t_2) = 1 + (t_1 - t_2)/(\alpha + t_2)$.

[We have $n(p, d) = \frac{1}{4}d^{-1}\sum_{c\setminus d}(3 + (-1)^c)\mu(c)(p^{d/c} - 1)$, which is about half the total number of irreducibles—exactly half, in fact, when d is odd. This proves that $\gcd(g_d(x), (x + t)^{(p^d-1)/2} - 1)$ has a good chance of finding factors of $g_d(x)$ when t is fixed and $g_d(x)$ is chosen at random; but a probabilistic algorithm is supposed to work with guaranteed probability for fixed $g_d(x)$ and random t, as in exercise 29.]

32. (a) Clearly $x^n - 1 = \prod_{d\setminus n}\Psi_d(x)$, since every complex nth root of unity is a primitive dth root for some unique $d \setminus n$. The second identity follows from the first; and $\Psi_n(x)$ has integer coefficients since it is expressed in terms of products and quotients of monic polynomials with integer coefficients. (b) The condition in the hint suffices to prove that $f(x) = \Psi_n(x)$, so we shall take the hint. When p does not divide n, we have $\gcd(x^n - 1, nx^{n-1}) = 1$ modulo p, hence $x^n - 1$ is squarefree modulo p. Given $f(x)$ and ζ as in the hint, let $g(x)$ be the irreducible factor of $\Psi_n(x)$ such that $g(\zeta^p) = 0$. If

$g(x) \neq f(x)$ then both $f(x)$ and $g(x)$ are distinct factors of $\Psi_n(x)$, hence they are distinct factors of $x^n - 1$, hence they have no irreducible factors in common modulo p. However, ζ^p is a root of $f(x^p)$, so $\gcd(g(x), f(x^p)) \neq 1$ over the integers, hence $g(x)$ is a divisor of $f(x^p)$. By (5), $g(x)$ is a divisor of $f(x)^p$, modulo p, contradicting the assumption that $f(x)$ and $g(x)$ have no irreducible factors in common. Therefore $f(x) = g(x)$. [The irreducibility of $\Psi_n(x)$ was first proved for prime n by K. F. Gauss in *Disquisitiones Arithmeticæ* (Leipzig, 1801), Art. 341, and for general n by L. Kronecker, *J. de Math. Pures et Appliquées* **19** (1854), 177–192.]

(c) $\Psi_1(x) = x - 1$; and when p is prime, $\Psi_p(x) = 1 + x + \cdots + x^{p-1}$. If $n > 1$ is odd, it is not difficult to prove that $\Psi_{2n}(x) = \Psi_n(-x)$. If p divides n, the second identity in (a) shows that $\Psi_{pn}(x) = \Psi_n(x^p)$. If p does not divide n, we have $\Psi_{pn}(x) = \Psi_n(x^p)/\Psi_n(x)$. For nonprime $n \leq 15$ we have $\Psi_4(x) = x^2 + 1$, $\Psi_6(x) = x^2 - x + 1$, $\Psi_8(x) = x^4 + 1$, $\Psi_9(x) = x^6 + x^3 + 1$, $\Psi_{10}(x) = x^4 - x^3 + x^2 - x + 1$, $\Psi_{12}(x) = x^4 - x^2 + 1$, $\Psi_{14}(x) = x^6 - x^5 + x^4 - x^3 + x^2 - x + 1$, $\Psi_{15}(x) = x^8 - x^7 + x^5 - x^4 + x^3 - x + 1$. [The formula $\Psi_{pq}(x) = (1 + x^p + \cdots + x^{(q-1)p})(x-1)/(x^q - 1)$ can be used to show that $\Psi_{pq}(x)$ has all coefficients ± 1 or 0 when p and q are prime; but the coefficients of $\Psi_{pqr}(x)$ can be arbitrarily large.]

33. False; we lose all p_j with e_j divisible by p. True if $p \geq \deg(u)$. [See exercise 36.]

34. [D. Y. Y. Yun, *Proc. ACM Symp. Symbolic and Algebraic Comp.* (1976), 26–35.] Set $(t(x), v_1(x), w_1(x)) \leftarrow \mathrm{GCD}(u(x), u'(x))$. If $t(x) = 1$, set $e \leftarrow 1$; otherwise set $(u_i(x), v_{i+1}(x), w_{i+1}(x)) \leftarrow \mathrm{GCD}(v_i(x), w_i(x) - v_i'(x))$ for $i = 1, 2, \ldots, e-1$, until finding $w_e(x) - v_e'(x) = 0$. Finally set $u_e(x) \leftarrow v_e(x)$.

To prove the validity of this algorithm, we observe that it computes the polynomials $t(x) = u_2(x)u_3(x)^2u_4(x)^3\ldots$, $v_i(x) = u_i(x)u_{i+1}(x)u_{i+2}(x)\ldots$, and

$$w_i(x) = u_i'(x)u_{i+1}(x)u_{i+2}(x)\ldots + 2u_i(x)u_{i+1}'(x)u_{i+2}(x)\ldots + 3u_i(x)u_{i+1}(x)u_{i+2}'(x)\ldots + \cdots.$$

We have $\gcd(t(x), w_1(x)) = 1$, since an irreducible factor of $u_i(x)$ divides all but the ith term of $w_1(x)$, and it is relatively prime to that term. Furthermore $\gcd(u_i(x), v_{i+1}(x))$ is clearly 1.

[Exercise 2(b) indicates that comparatively few polynomials are squarefree, but non-squarefree polynomials actually occur often in practice; hence this method turns out to be quite important. See Paul S. Wang and Barry M. Trager, *SIAM J. Computing* **8** (1979), 300–305, for suggestions on how to improve the efficiency when the given polynomial is already squarefree.]

35. We have $w_j(x) = \gcd(u_j(x), v_j^*(x)) \cdot \gcd(u_{j+1}^*(x), v_j(x))$, where

$$u_j^*(x) = u_j(x)u_{j+1}(x)\ldots \qquad \text{and} \qquad v_j^*(x) = v_j(x)v_{j+1}(x)\ldots.$$

[Yun notes that the running time for squarefree factorization by the method of exercise 34 is at most about twice the running time to calculate $\gcd(u(x), u'(x))$. Furthermore if we are given an arbitrary method for discovering squarefree factorization, the method of this exercise leads to a gcd procedure. (When $u(x)$ and $v(x)$ are squarefree, their gcd is simply $w_2(x)$ where $w(x) = u(x)v(x) = w_1(x)w_2(x)^2$; the polynomials $u_j(x)$, $v_j(x)$, $u_j^*(x)$, and $v_j^*(x)$ are all squarefree.) Hence the problem of converting a primitive polynomial of degree n to its squarefree representation is computationally *equivalent* to the problem of calculating the gcd of two nth degree polynomials, in the sense of asymptotic worst-case running time.]

36. Let $U_j(x)$ be the value computed for "$u_j(x)$" by the procedure of exercise 34. If $\deg(U_1)+2\deg(U_2)+\cdots=\deg(u)$, then $u_j(x)=U_j(x)$ for all j. But in general we will have $e<p$ and $U_j(x)=\prod_{k\geq 0}u_{j+pk}(x)$ for $1\leq j<p$. To separate these factors further, we can calculate $t(x)/(U_2(x)U_3(x)^2\ldots U_{p-1}(x)^{p-2})=\prod_{j\geq p}u_j(x)^{p\lfloor j/p\rfloor}=z(x^p)$. After recursively finding the squarefree representation of $z(x)=(z_1(x),z_2(x),\ldots)$, we will have $z_k(x)=\prod_{0\leq j<p}u_{j+pk}(x)$, so we can calculate the individual $u_i(x)$ by the formula $\gcd(U_j(x),z_k(x))=u_{j+pk}(x)$ for $1\leq j<p$. The polynomial $u_{pk}(x)$ will be left when the other factors of $z_k(x)$ have been removed.

Note: This procedure is fairly simple but the program is lengthy. If one's goal is to have a short program for complete factorization modulo p, rather than an extremely efficient one, it is probably easiest to modify the distinct-degree factorization routine so that it casts out $\gcd(x^{p^d}-x,u(x))$ several times for the same value of d until the gcd is 1. In this case you needn't begin by calculating $\gcd(u(x),u'(x))$ and removing multiple factors as suggested in the text, since the polynomial $x^{p^d}-x$ is squarefree.

37. The exact probability is $\prod_{j\geq 1}(a_{jp}/p^j)^{k_j}/k_j!$, where k_j is the number of d_i that are equal to j. Since $a_{jp}/p^j\approx 1/j$ by exercise 4, we get the formula of exercise 1.3.3–21.

Notes: This exercise says that if we fix the prime p and let the polynomial $u(x)$ be random, it will have certain probability of splitting in a given way modulo p. A much harder problem is to fix the polynomial $u(x)$ and to let p be "random"; it turns out that the same asymptotic result holds for almost all $u(x)$. G. Frobenius proved in 1880 that the integer polynomial $u(x)$ splits modulo p into factors of degrees $d_1,\ldots,d_r$, when p is a large prime chosen at random, with probability equal to the number of permutations in the Galois group G of $u(x)$ having cycle lengths $\{d_1,\ldots,d_r\}$ divided by the total number of permutations in G. (If $u(x)$ has rational coefficients and distinct roots $\xi_1,\ldots,\xi_n$ over the complex numbers, its Galois group is the (unique) group G of permutations such that the polynomial $\prod_{p(1)\ldots p(n)\in G}(z+\xi_{p(1)}y_1+\cdots+\xi_{p(n)}y_n)=U(z,y_1,\ldots,y_n)$ has rational coefficients and is irreducible over the rationals.) Furthermore B. L. van der Waerden proved in 1934 that almost all polynomials of degree n have the set of all $n!$ permutations as their Galois group. Therefore almost all fixed irreducible polynomials $u(x)$ will factor as we might expect them to, with respect to randomly chosen large primes p. References: *Sitzungsberichte Königl. Preuß. Akad. Wiss.* (Berlin: 1896), 689–703; *Math. Annalen* **109** (1934), 13–16. See also N. Chebotarev, *Math. Annalen* **95** (1926), for a generalization of Frobenius's theorem to conjugacy classes of the Galois group.

38. (Partial solution by Peter Weinberger.) The average number of 1-cycles in a randomly chosen element of any transitive permutation group G on n objects is 1, since the probability is $1/n$ that any given object is fixed. Since Galois groups are always transitive, the remarks in the previous answer show that a fixed irreducible polynomial has exactly one linear factor modulo p, on the average, as $p\to\infty$. Thus, *the average number of linear factors of $u(x)$ modulo p is the number of irreducible factors of $u(x)$ over the integers.*

Weinberger [*Proc. Symp. Pure Math.* **24** (Amer. Math. Soc., 1973), 321–332] has proved that if the generalized Riemann hypothesis (GRH) holds—this is a conjecture about the zeros of a generalized zeta function—then there is an absolute constant A_1 with the following property: The number of prime ideals of norm $\leq x$, in any algebraic number field of degree n, differs from $\int_2^x dt/\ln t$ by at most $A_1 n x^{1/2}\ln(x\Delta)$, where Δ is the absolute value of the discriminant of the irreducible polynomial u that defines the

field. The number of prime ideals of norm $\leq x$ is at most $nx^{1/2}$ different from the total number of linear factors of u modulo primes $\leq x$, since such linear factors correspond to ideals of norm p, while other prime ideals have norms $\geq p^2$. Consequently if we let $N(x)$ be the total number of linear factors of a given primitive polynomial u, modulo all primes $\leq x$, then the GRH implies that there is an absolute constant A_2 such that $|N(x)/\pi(x) - r| < A_2 n x^{-1/2}(\ln x)(\ln x\Delta)$. A proof of GRH would yield a "short" proof of the number of irreducible factors of any primitive polynomial u over the integers, since we could evaluate $N(x)$ for a value of x sufficiently large to make this error bound less than $\frac{1}{2}$. Unfortunately A_2 is quite large.

SECTION 4.6.3

1. x^m, where $m = 2^{\lambda(n)}$, the highest power of 2 less than or equal to n.

2. Assume that x is input in register A, and n in location NN; the output is in register X.

```
01  A1 ENTX 1      1          A1. Initialize.
02     STX  Y      1          Y ← 1.
03     STA  Z      1          Z ← x.
04     LDA  NN     1          N ← n.
05     JMP  2F     1          To A2.
06  5H SRB  1      L+1-K
07     STA  N      L+1-K      N ← ⌊N/2⌋.
08  A5 LDA  Z      L          A5. Square Z.
09     MUL  Z      L          Z × Z mod w
10     STX  Z      L              → Z.
11  A2 LDA  N      L          A2. Halve N.
12  2H JAE  5B     L+1        To A5 if N is even.
13     SRB  1      K
14  A4 JAZ  4F     K          Jump if N = 1.
15     STA  N      K-1        N ← ⌊N/2⌋.
16  A3 LDA  Z      K-1        A3. Multiply Y by Z.
17     MUL  Y      K-1        Z × Y mod w
18     STX  Y      K-1            → Y.
19     JMP  A5     K-1        To A5.
20  4H LDA  Z      1
21     MUL  Y      1          Do the final multiplication. ∎
```

[It would be better programming practice to change the instruction in line 05 to "JAP", followed by an error indication. The running time is $21L + 16K + 8$, where $L = \lambda(n)$ is one less than the number of bits in the binary representation of n, and $K = \nu(n)$ is the number of 1 bits in that representation.]

For the serial program, we may assume that n is small enough to fit in an index register; otherwise serial exponentiation is out of the question. The following program leaves the output in register A:

```
01  S1 LD1  NN     1      rI1 ← n.
02     STA  X      1      X ← x.
03     JMP  2F     1
```

```
04  1H MUL  X  N-1   rA × X mod w
05     SLAX 5  N-1        → rA.
06  2H DEC1 1    N     rI1 ← rI1 - 1.
07     J1P  1B   N     Multiply again if rI1 > 0.  ∎
```

The running time for this program is $14N - 7$; it is faster than the previous program when $n \leq 7$, slower when $n \geq 8$.

3. The sequences of exponents are: (a) 1, 2, 3, 6, 7, 14, 15, 30, 60, 120, 121, 242, 243, 486, 487, 974, 975 [16 multiplications]; (b) 1, 2, 3, 4, 8, 12, 24, 36, 72, 108, 216, 324, 325, 650, 975 [14 multiplications]; (c) 1, 2, 3, 6, 12, 15, 30, 60, 120, 240, 243, 486, 972, 975 [13 multiplications]; (d) 1, 2, 3, 6, 12, 15, 30, 60, 75, 150, 300, 600, 900, 975 [13 multiplications]. [The smallest possible number of multiplications is 12; this is obtainable by combining the factor method with the binary method, since $975 = 15 \cdot (2^6 + 1)$.]

4. $(777777)_8 = 2^{18} - 1$.

5. T1. [Initialize.] Set LINKU$[j] \leftarrow 0$ for $1 \leq j \leq 2^r$, and set $k \leftarrow 0$, LINKR$[0] \leftarrow 1$, LINKR$[1] \leftarrow 0$.

T2. [Change level.] (Now level k of the tree has been linked together from left to right, starting at LINKR[0].) If $k = r$, the algorithm terminates. Otherwise set $n \leftarrow$ LINKR$[0]$, $m \leftarrow 0$.

T3. [Prepare for n.] (Now n is a node on level k, and m points to the rightmost node currently on level $k + 1$.) Set $q \leftarrow 0$, $s \leftarrow n$.

T4. [Already in tree?] (Now s is a node in the path from the root to n.) If LINKU$[n + s] \neq 0$, go to T6 (the value $n + s$ is already in the tree).

T5. [Insert below n.] If $q = 0$, set $m' \leftarrow n + s$. Then set LINKR$[n + s] \leftarrow q$, LINKU$[n + s] \leftarrow n$, $q \leftarrow n + s$.

T6. [Move up.] Set $s \leftarrow$ LINKU$[s]$. If $s \neq 0$, return to T4.

T7. [Attach group.] If $q \neq 0$, set LINKR$[m] \leftarrow q$, $m \leftarrow m'$.

T8. [Move n.] Set $n \leftarrow$ LINKR$[n]$. If $n \neq 0$, return to T3.

T9. [End of level.] Set LINKR$[m] \leftarrow 0$, $k \leftarrow k + 1$, and return to T2. ∎

6. Prove by induction that the path to the number $2^{e_0} + 2^{e_1} + \cdots + 2^{e_t}$, if $e_0 > e_1 > \cdots > e_t \geq 0$, is $1, 2, 2^2, \ldots, 2^{e_0}, 2^{e_0} + 2^{e_1}, \ldots, 2^{e_0} + 2^{e_1} + \cdots + 2^{e_t}$; furthermore, the sequences of exponents on each level are in decreasing lexicographic order.

7. The binary and factor methods require one more step to compute x^{2n} than x^n; the power tree method requires at most one more step. Hence (a) $15 \cdot 2^k$; (b) $33 \cdot 2^k$; (c) $23 \cdot 2^k$; $k = 0, 1, 2, 3, \ldots$.

8. The power tree always includes the node $2m$ at one level below m, unless it occurs at the same level or an earlier level; and it always includes the node $2m + 1$ at one level below $2m$, unless it occurs at the same level or an earlier level. [It is not true that $2m$ is a son of m in the power tree for all m; the smallest example where this fails is $m = 2138$, which appears on level 15, while 4276 appears elsewhere on level 16. In fact, $2m$ sometimes occurs on the same level as m; the smallest example is $m = 6029$.]

9. Start with $N \leftarrow n$, $Z \leftarrow x$, and $Y_k \leftarrow 1$ for $1 \leq k < m$; in general we will have $x^n = Y_1 Y_2^2 \ldots Y_{m-1}^{m-1} Z^N$. If $N > 0$, set $k \leftarrow N \bmod m$, and if $k \neq 0$ set $Y_k \leftarrow Y_k \cdot Z$. Then set $Z \leftarrow Z^m$, $N \leftarrow \lfloor N/m \rfloor$, and repeat. Finally set $Y_k \leftarrow Y_k \cdot Y_{k+1}$ for $k = m-2$, $m - 3, \ldots, 1$; the answer is $Y_1 \ldots Y_{m-1}$.

10. By using the "FATHER" representation discussed in Section 2.3.3: Make use of a table $f[j]$, $1 \le j \le 100$, such that $f[1] = 0$ and $f[j]$ is the number of the node just above j for $j \ge 2$. (The fact that each node of this tree has degree at most two has no effect on the efficiency of this representation; it just makes the tree look prettier as an illustration.)

11. 1, 2, 3, 5, 10, 20, (23 or 40), 43; 1, 2, 4, 8, 9, 17, (26 or 34), 43; 1, 2, 4, 8, 9, 17, 34, (43 or 68), 77; 1, 2, 4, 5, 9, 18, 36, (41 or 72), 77. If either of the latter two paths were in the tree we would have no possibility for $n = 43$, since the tree must contain either 1, 2, 3, 5 or 1, 2, 4, 8, 9.

12. No such infinite tree can exist, since $l(n) \neq l^*(n)$ for some n.

13. For Case 1, use a Type-1 chain followed by $2^{A+C} + 2^{B+C} + 2^A + 2^B$; or use the factor method. For Case 2, use a Type-2 chain followed by $2^{A+C+1} + 2^{B+C} + 2^A + 2^B$. For Case 3, use a Type-5 chain followed by addition of $2^A + 2^{A-1}$, or use the factor method. For Case 4, $n = 135 \cdot 2^D$, so we may use the factor method.

14. (a) It is easy to verify that steps $r-1$ and $r-2$ are not both small, so let us assume that step $r-1$ is small and step $r-2$ is not. If $c = 1$, then $\lambda(a_{r-1}) = \lambda(a_{r-k})$, so $k = 2$; and since $4 \le \nu(a_r) = \nu(a_{r-1}) + \nu(a_{r-k}) - 1 \le \nu(a_{r-1}) + 1$, we have $\nu(a_{r-1}) \ge 3$, making $r-1$ a star step (lest $a_0, a_1, \ldots, a_{r-3}, a_{r-1}$ include only one small step). Then $a_{r-1} = a_{r-2} + a_{r-q}$ for some q, and if we replace a_{r-2}, a_{r-1}, a_r by $a_{r-2}, 2a_{r-2}, 2a_{r-2} + a_{r-q} = a_r$, we obtain another counterexample chain in which step r is small; but this is impossible. On the other hand, if $c \ge 2$, then $4 \le \nu(a_r) \le \nu(a_{r-1}) + \nu(a_{r-k}) - 2 \le \nu(a_{r-1})$; hence $\nu(a_{r-1}) = 4$, $\nu(a_{r-k}) = 2$, and $c = 2$. This leads readily to an impossible situation by a consideration of the six types in the proof of Theorem B.

(b) If $\lambda(a_{r-k}) < m-1$, we have $c \ge 3$, so $\nu(a_{r-k}) + \nu(a_{r-1}) \ge 7$ by (22); therefore both $\nu(a_{r-k})$ and $\nu(a_{r-1})$ are ≥ 3. All small steps must be $\le r-k$, and $\lambda(a_{r-k}) = m-k+1$. If $k \ge 4$, we must have $c = 4$, $k = 4$, $\nu(a_{r-1}) = \nu(a_{r-4}) = 4$; thus $a_{r-1} \ge 2^m + 2^{m-1} + 2^{m-2}$, and a_{r-1} must equal $2^m + 2^{m-1} + 2^{m-2} + 2^{m-3}$; but $a_{r-4} \ge \frac{1}{8} a_{r-1}$ now implies that $a_{r-1} = 8a_{r-4}$. Thus $k = 3$ and $a_{r-1} > 2^m + 2^{m-1}$. Since $a_{r-2} < 2^m$ and $a_{r-3} < 2^{m-1}$, step $r-1$ must be a doubling; but step $r-2$ is a nondoubling, since $a_{r-1} \neq 4a_{r-3}$. Furthermore, since $\nu(a_{r-3}) \ge 3$, $r-3$ is a star step; and $a_{r-2} = a_{r-3} + a_{r-5}$ would imply that $a_{r-5} = 2^{m-2}$, hence we must have $a_{r-2} = a_{r-3} + a_{r-4}$. As in a similar case treated in the text, the only possibility is now seen to be $a_{r-4} = 2^{m-2} + 2^{m-3}$, $a_{r-3} = 2^{m-2} + 2^{m-3} + 2^{d+1} + 2^d$, $a_{r-1} = 2^m + 2^{m-1} + 2^{d+2} + 2^{d+1}$, and even this possibility is impossible.

16. $l^B(n) = \lambda(n) + \nu(n) - 1$; so if $n = 2^k$, $l^B(n)/\lambda(n) = 1$, but if $n = 2^{k+1} - 1$, $l^B(n)/\lambda(n) = 2$.

17. Let $i_1 < \cdots < i_t$. Delete any intervals I_k that can be removed without affecting the union $I_1 \cup \cdots \cup I_t$. (The interval $(j_k, i_k]$ may be dropped out if either $j_{k+1} \le j_k$ or $j_1 < j_2 < \cdots$ and $j_{k+1} \le i_{k-1}$.) Now combine overlapping intervals $(j_1, i_1], \ldots, (j_d, i_d]$ into an interval $(j', i'] = (j_1, i_d]$ and note that

$$a_{i'} < a_{j'}(1+\delta)^{i_1 - j_1 + \cdots + i_d - j_d} \le a_{j'}(1+\delta)^{2(i'-j')},$$

since each point of $(j', i']$ is covered at most twice in $(j_i, i_1] \cup \cdots \cup (j_d, i_d]$.

18. Call $f(m)$ a "nice" function if $(\log f(m))/m \to 0$ as $m \to \infty$. A polynomial in m is nice. The product of nice functions is nice. If $g(m) \to 0$ and c is a positive constant, then $c^{mg(m)}$ is nice; also $\binom{2m}{mg(m)}$ is nice, for by Stirling's approximation this is equivalent to saying that $g(m)\log(1/g(m)) \to 0$.

Now replace each term of the summation by the maximum term that is attained for any s, t, v. The total number of terms is nice, and so are $\binom{m+s}{t+v}$, $\binom{t+v}{v} \le 2^{t+v}$, and β^{2v}, because $(t+v)/m \to 0$. Finally, $\binom{(m+s)^2}{t} \le (2m)^{2t}/t! < (4m^2/t)^t e^t$, where $(4e)^t$ is nice; setting t to its maximum value $(1-\frac{1}{2}\epsilon)m/\lambda(m)$, we have the upper bound $(m^2/t)^t = (m\lambda(m)/(1-\frac{1}{2}\epsilon))^t = 2^{m(1-\epsilon/2)} \cdot f(m)$, where $f(m)$ is nice. Hence the entire sum is less than α^m for large m, if $\alpha = 2^{1-\eta}$, $0 < \eta < \frac{1}{2}\epsilon$.

19. (a) $M \cap N$, $M \cup N$, $M \uplus N$, respectively; see Eqs. 4.5.2–6, 4.5.2–7.

(b) $f(z)g(z)$, $\operatorname{lcm}(f(z), g(z))$, $\gcd(f(z), g(z))$. (For the same reasons as (a), because the monic irreducible polynomials over the complex numbers are precisely the polynomials $z - \zeta$.)

(c) Commutative laws $A \uplus B = B \uplus A$, $A \cup B = B \cup A$, $A \cap B = B \cap A$. Associative laws $A \uplus (B \uplus C) = (A \uplus B) \uplus C$, $A \cup (B \cup C) = (A \cup B) \cup C$, $A \cap (B \cap C) = (A \cap B) \cap C$. Distributive laws $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$, $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$, $A \uplus (B \cup C) = (A \uplus B) \cup (A \uplus C)$, $A \uplus (B \cap C) = (A \uplus B) \cap (A \uplus C)$. Idempotent laws $A \cup A = A$, $A \cap A = A$. Absorption laws $A \cup (A \cap B) = A$, $A \cap (A \cup B) = A$, $A \cap (A \uplus B) = A$, $A \cup (A \uplus B) = A \uplus B$. Identity and zero laws $\emptyset \uplus A = A$, $\emptyset \cup A = A$, $\emptyset \cap A = \emptyset$, where $\emptyset$ is the empty multiset. Counting law $A \uplus B = (A \cup B) \uplus (A \cap B)$. Further properties analogous to those of sets come from the partial ordering defined by the rule $A \subseteq B$ iff $A \cap B = A$ (iff $A \cup B = B$).

Notes: Other common applications of multisets are zeros and poles of meromorphic functions, invariants of matrices in canonical form, invariants of finite Abelian groups, etc.; multisets can be useful in combinatorial counting arguments and in the development of measure theory. The terminal strings of a noncircular context-free grammar form a multiset that is a set if and only if the grammar is unambiguous. Although multisets appear frequently in mathematics, they often must be treated rather clumsily because there is currently no standard way to treat sets with repeated elements. Several mathematicians have voiced their belief that the lack of adequate terminology and notation for this common concept has been a definite handicap to the development of mathematics. (A multiset is, of course, formally equivalent to a mapping from a set into the nonnegative integers, but this formal equivalence is of little or no practical value for creative mathematical reasoning.) The author has discussed this matter with many people in an attempt to find a good remedy. Some of the names suggested for the concept were list, bunch, bag, heap, sample, weighted set, collection; but these words either conflict with present terminology, have an improper connotation, or are too much of a mouthful to say and to write conveniently. It does not seem out of place to coin a new word for such an important concept, and "multiset" has been suggested by N. G. de Bruijn. The notation "$A \uplus B$" has been selected by the author to avoid conflict with existing notations and to stress the analogy with set union. It would not be as desirable to use "$A + B$" for this purpose, since algebraists have found that $A + B$ is a good notation for $\{\alpha + \beta \mid \alpha \in A \text{ and } \beta \in B\}$. If A is a multiset of nonnegative integers, let $G(z) = \sum_{n \in A} z^n$ be a generating function corresponding to A. (Generating functions with nonnegative integer coefficients obviously correspond one-to-one with multisets of nonnegative integers.) If $G(z)$ corresponds to A and $H(z)$ to B, then $G(z) + H(z)$ corresponds to $A \uplus B$ and $G(z)H(z)$ corresponds to $A + B$.

If we form "Dirichlet" generating functions $g(z) = \sum_{n\in A} 1/n^z$, $h(z) = \sum_{n\in B} 1/n^z$, then the product $g(z)h(z)$ corresponds to the multiset product AB.

20. Type 3: $(S_0, \ldots, S_r) = (M_{00}, \ldots, M_{r0}) = (\{0\}, \ldots, \{A\}, \{A-1, A\}, \{A-1, A, A\}, \{A-1, A-1, A, A, A\}, \ldots, \{A+C-3, A+C-3, A+C-2, A+C-2, A+C-2\})$. Type 5: $(M_{00}, \ldots, M_{r0}) = (\{0\}, \ldots, \{A\}, \{A-1, A\}, \ldots, \{A+C-1, A+C\}, \{A+C-1, A+C-1, A+C\}, \ldots, \{A+C+D-1, A+C+D-1, A+C+D\})$; $(M_{01}, \ldots, M_{r1}) = (\emptyset, \ldots, \emptyset, \emptyset, \ldots, \emptyset, \{A+C-2\}, \ldots, \{A+C+D-2\})$, $S_i = M_{i0} \uplus M_{i1}$.

21. For example, let $u = 2^{8q+5}$, $x = (2^{(q+1)u} - 1)/(2^u - 1) = 2^{qu} + \cdots + 2^u + 1$, $y = 2^{(q+1)u} + 1$. Then $xy = (2^{2(q+1)u} - 1)/(2^u - 1)$. If $n = 2^{4(q+1)u} + xy$, we have $l(n) \le 4(q+1)u + q + 2$ by Theorem F, but $l^*(n) = 4(q+1)u + 2q + 2$ by Theorem H.

22. Underline everything except the $u - 1$ insertions used in the calculation of x.

23. Theorem G (everything underlined).

24. Use the numbers $(B^{a_i} - 1)/(B-1)$, $0 \le i \le r$, underlined when a_i is underlined; and $c_k B^{i-1}(B^{b_j} - 1)/(B-1)$ for $0 \le j < t$, $0 < i \le b_{j+1} - b_j$, $1 \le k \le l^0(B)$, underlined when c_k is underlined, where $c_0, c_1, \ldots$ is a minimum length l^0-chain for B. To prove the second inequality, let $B = 2^m$ and use (3). (The second inequality is rarely, if ever, an improvement on Theorem G.)

25. We may assume that $d_k = 1$. Use the rule R $A_{k-1} \ldots A_1$, where A_j = "XR" if $d_j = 1$, A_j = "R" otherwise, and where "R" means take the square root, "X" means multiply by x. For example, if $y = (.1101101)_2$, the rule is R R XR XR R XR XR. (There exist binary square-root extraction algorithms suitable for computer hardware, requiring an execution time comparable to that of division; computers with such hardware could therefore calculate more general fractional powers using the technique in this exercise.)

26. If we know the pair (F_k, F_{k-1}), then we have $(F_{k+1}, F_k) = (F_k + F_{k-1}, F_k)$ and $(F_{2k}, F_{2k-1}) = (F_k^2 + 2F_kF_{k-1}, F_k^2 + F_{k-1}^2)$; so a binary method can be used to calculate (F_n, F_{n-1}), using $O(\log n)$ arithmetic operations. Perhaps better is to use the pair of values (F_k, L_k), where $L_k = F_{k-1} + F_{k+1}$ (cf. Section 4.5.4); then we have $(F_{k+1}, L_{k+1}) = (\frac{1}{2}(F_k + L_k), \frac{1}{2}(5F_k + L_k))$, $(F_{2k}, L_{2k}) = (F_kL_k, L_k^2 - 2(-1)^k)$.

For the general linear recurrence $x_n = a_1x_{n-1} + \cdots + a_dx_{n-d}$, we can compute x_n in $O(d^3 \log n)$ arithmetic operations by computing the nth power of an appropriate $d \times d$ matrix. [This observation is due to J. C. P. Miller and D. J. S. Brown, *Comp. J.* **9** (1966), 188–190.]

27. First form the $2^m - m - 1$ products $x_1^{e_1} \ldots x_m^{e_m}$, for all sequences of exponents such that $0 \le e_j \le 1$ and $e_1 + \cdots + e_m \ge 2$. Let $n_j = (d_{j\lambda} \ldots d_{j1}d_{j0})_2$; to complete the calculation, take $x_1^{d_{1\lambda}} \ldots x_m^{d_{m\lambda}}$, then square and multiply by $x_1^{d_{1i}} \ldots x_m^{d_{mi}}$, for $i = \lambda - 1, \ldots, 1, 0$. [Straus showed in *AMM* **71** (1964), 807–808, that $2\lambda(n)$ may be replaced by $(1+\epsilon)\lambda(n)$ for any $\epsilon > 0$, by generalizing this binary method to 2^k-ary as in Theorem D. See exercise 39 for later developments.]

28. (a) $x \nabla y = x \vee y \vee (x+y)$, where "$\vee$" is logical "or", cf. exercise 4.6.2–26; clearly $\nu(x \nabla y) \le \nu(x \vee y) + \nu(x \wedge y) = \nu(x) + \nu(y)$. (b) Note first that $A_{i-1}/2^{d_i - 1} \subseteq A_i/2^{d_i}$ for $1 \le i \le r$. Secondly, note that $d_j = d_{i-1}$ in a nondoubling; for otherwise $a_{i-1} \ge 2a_j \ge a_j + a_k = a_i$. Hence $A_j \subseteq A_{i-1}$ and $A_k \subseteq A_{i-1}/2^{d_j - d_k}$. (c) An easy induction on i, except that close steps need closer attention. Let us say that m has property $P(\alpha)$ if the 1's in its binary representation all appear in consecutive

blocks of $\geq \alpha$ in a row. If m and m' have $P(\alpha)$, so does $m \nabla m'$; if m has $P(\alpha)$ then $\rho(m)$ has $P(\alpha+\delta)$. Hence B_i has $P(1+\delta c_i)$. Finally if m has $P(\alpha)$ then $\nu(\rho(m)) \leq (\alpha+\delta)\nu(m)/\alpha$; for $\nu(m) = \nu_1 + \cdots + \nu_q$, where each block size ν_j is $\geq \alpha$, hence $\nu(\rho(m)) \leq (\nu_1+\delta)+\cdots+(\nu_q+\delta) \leq (1+\delta/\alpha)\nu_1+\cdots+(1+\delta/\alpha)\nu_q$. (d) Let $f = b_r + c_r$ be the number of nondoublings and s the number of small steps. If $f \geq 3.271 \lg \nu(n)$ we have $s \geq \lg \nu(n)$ as desired, by (16). Otherwise we have $a_i \leq (1+2^{-\delta})^{b_i} 2^{c_i+d_i}$ for $0 \leq i \leq r$, hence $n \leq ((1+2^{-\delta})/2)^{b_r} 2^r$, and $r \geq \lg n + b_r - b_r \lg(1+2^{-\delta}) \geq \lg n + \lg \nu(n) - \lg(1+\delta c_r) - b_r \lg(1+2^{-\delta})$. Let $\delta = \lceil \lg(f+1) \rceil$; then $\ln(1+2^{-\delta}) \leq \ln(1+1/(f+1)) \leq 1/(f+1) \leq \delta/(1+\delta f)$, and it follows that $\lg(1+\delta x) + (f-x)\lg(1+2^{-\delta}) \leq \lg(1+\delta f)$ for $0 \leq x \leq f$. Hence finally $l(n) \geq \lg n + \lg \nu(n) - \lg(1+(3.271 \lg \nu(n))\lceil \lg(1+3.271 \lg \nu(n)) \rceil)$. [*Theoretical Comp. Sci.* **1** (1975), 1–12.]

29. In the paper just cited, Schönhage refined the method of exercise 28 to prove that $l(n) \geq \lg n + \lg \nu(n) - 2.13$ for all n. Can the remaining gap be closed?

30. $n = 31$ is the smallest example; $l(31) = 7$, but 1, 2, 4, 8, 16, 32, 31 is an addition-subtraction chain of length 6. [After proving Theorem E, Erdős stated that the same result holds also for addition-subtraction chains. Schönhage has extended the lower bound of exercise 28 to addition-subtraction chains, with $\nu(n)$ replaced by $\bar{\nu}(n)$ = minimum number of nonzero digits to represent $n = (n_q \ldots n_0)_2$ where each n_j is -1, 0, or $+1$. This quantity $\bar{\nu}(n)$ is the number of 1's, in the ordinary binary representation of n, that are immediately preceded by 0 or by the string $00(10)^k1$ for some $k \geq 0$.]

32. First compute 2^i for $1 \leq i \leq \lambda(n_m)$, then compute each $n = n_j$ by the following variant of the 2^k-ary method: For all odd $i < 2^k$, compute $f_i = \sum\{ 2^{kt+e} \mid d_t = 2^e i \}$ where $n = (\ldots d_1 d_0)_{2^k}$, in at most $\lfloor \frac{1}{k} \lg n \rfloor$ steps; then compute $n = \sum i f_i$ in at most $\sum l(i) + 2^{k-1}$ further steps. The number of steps per n_j is $\leq \lfloor \frac{1}{k} \lg n \rfloor + O(k2^k)$, and this is $\lambda(n)/\lambda\lambda(n) + O(\lambda(n)\lambda\lambda\lambda(n)/\lambda\lambda(n)^2)$ when $k = \lfloor \lg \lg n - 3 \lg \lg \lg n \rfloor$.

[A generalization of Theorem E gives the corresponding lower bound. Reference: *SIAM J. Computing* **5** (1976), 100–103. See also exercise 39.]

33. The following construction due to D. J. Newman provides the best upper bound currently known: Let $k = p_1 \ldots p_r$ be the product of the first r primes. Compute k and all quadratic residues mod k by the method of exercise 32, in $O(2^{-r} k \log k)$ steps (because there are approximately $2^{-r}k$ quadratic residues). Also compute all multiples of k that are $\leq m^2$, in about m^2/k further steps. Now m additions suffice to compute $1^2, 2^2, \ldots, m^2$. We have $k = \exp(p_r + O(p_r/(\log p_r)^{1000}))$ where p_r is given by the formula in exercise 4.5.4–35; so by choosing

$$r = \lfloor (1 + (1 + \tfrac{1}{2}\ln 2)/\lg\lg m) \ln m / \ln\ln m \rfloor$$

it follows that $l(1^2, \ldots, m^2) = m + O(m \cdot \exp(-\frac{1}{2}\ln 2 \ln m / \ln\ln m))$.

On the other hand, D. Dobkin and R. Lipton have shown that, for any $\epsilon > 0$, $l(1^2, \ldots, m^2) > m + m^{2/3-\epsilon}$ when m is sufficiently large [*SIAM J. Computing* **9** (1980), 121–125].

35. See *Discrete Math.* **23** (1978), 115–119.

36. Eight; there are four ways to compute $39 = 12+12+12+3$ and two ways to compute $79 = 39+39+1$.

37. The statement is true. The labels in the reduced graph of the binary chain are $\lfloor n/2^k \rfloor$ for $k = e_0, \ldots, 0$; they are $1, 2, \ldots, 2^{e_0}, n$ in the dual graph. [Similarly, the right-to-left m-ary method of exercise 9 is the dual of the left-to-right method.]

38. 2^t are equivalent to the binary chain; it would be 2^{t-1} if $e_0 = e_1 + 1$. The number of chains equivalent to the scheme of Algorithm A is the number of ways to compute the sum of $t+2$ numbers of which two are identical. This is $\frac{1}{2}f_{t+1} + \frac{1}{2}f_t$, where f_m is the number of ways to compute the sum of $m+1$ distinct numbers. When we take commutativity into account, we see that f_m is 2^{-m} times $(m+1)!$ times the number of binary trees on m nodes, so $f_m = (2m-1)(2m-3)\ldots 1$.

39. The quantity $l([n_1, n_2, \ldots, n_m])$ is the minimum of arcs $-$ vertices $+ m$ taken over all directed graphs having m vertices s_j whose in-degree is zero and one vertex t whose out-degree is zero, where there are exactly n_j oriented paths from s_j to t for $1 \le j \le m$. The quantity $l(n_1, n_2, \ldots, n_m)$ is the minimum of arcs $-$ vertices $+1$ taken over all directed graphs having one vertex s whose in-degree is zero and m vertices t_j whose out-degree is zero, where there are exactly n_j oriented paths from s to t_j for $1 \le j \le m$. These problems are dual to each other, if we change the direction of all the arcs.

Note: C. H. Papadimitriou points out that this is a special case of a much more general theorem. Let $N = (n_{ij})$ be an $m \times p$ matrix of nonnegative integers having no row or column entirely zero. We can define $l(N)$ to be the minimum number of multiplications needed to compute the set of monomials $\{ x_1^{n_{1j}} \ldots x_m^{n_{mj}} \mid 1 \le j \le p \}$. Now $l(N)$ is also the minimum of arcs $-$ vertices $+ m$ taken over all directed graphs having m vertices s_i whose in-degree is zero and p vertices t_j whose out-degree is zero, where there are exactly n_{ij} oriented paths from s_i to t_j for each i and j. By duality we have $l(N) = l(N^T) + m - p$.

N. Pippenger has proved a comprehensive generalization of the results of exercises 27 and 32. Let $L(m, p, n)$ be the maximum of $l(N)$ taken over all $m \times p$ matrices N of nonnegative integers $n_{ij} \le n$. Then $L(m, p, n) = \min(m, p) \lg n + H/\lg H + O(m + p + H(\log\log H)^{1/2}(\log H)^{-3/2})$, where $H = mp \lg(n+1)$. [*SIAM J. Computing* **9** (1980), 230–250.]

40. By exercise 39, it suffices to show that $l(m_1 n_1 + \cdots + m_t n_t) \le l(m_1, \ldots, m_t) + l([n_1, \ldots, n_t])$. But this is clear, since we can first form $\{x^{m_1}, \ldots, x^{m_t}\}$ and then compute the monomial $(x^{m_1})^{n_1} \ldots (x^{m_t})^{n_t}$.

Notes: Theorem F is a special case of this result, since we clearly have $l(2^m, x) \le m + \nu(x) - 1$ when $\lambda(x) \le m$. One strong way to state Olivos's theorem is that if $a_0, \ldots, a_r$ and $b_0, \ldots, b_s$ are any addition chains, then $l(\sum c_{ij} a_i b_j) \le r + s + \sum c_{ij} - 1$ for any $(r+1) \times (s+1)$ matrix of nonnegative integers c_{ij}.

41. [To appear.] The stated formula can be proved whenever $A \ge 9m^2$. Since this is a polynomial in m, and since the problem of finding a minimum vertex cover is NP hard (cf. Section 7.9), the problem of computing $l(n_1, \ldots, n_m)$ is NP complete. [It is unknown whether or not the problem of computing $l(n)$ is NP complete.]

SECTION 4.6.4

1. Set $y \leftarrow x^2$, then compute $((\ldots(u_{2n+1}y + u_{2n-1})y + \cdots)y + u_1)x$.

2. Replacing x in (2) by the polynomial $x + x_0$ leads to the following procedure:

G1. Do step G2 for $k = n, n-1, \ldots, 0$ (in this order), and stop.

G2. Set $v_k \leftarrow u_k$, and then set $v_j \leftarrow v_j + x_0 v_{j+1}$ for $j = k, k+1, \ldots, n-1$. (When $k = n$, this step simply sets $v_n \leftarrow u_n$.) ∎

The computations turn out to be identical to those in H1 and H2, but performed in a different order. (This application was, in fact, Newton's original motivation for using scheme (2).)

3. The coefficient of x^k is a polynomial in y that may be evaluated by Horner's rule: $(\ldots(u_{n,0}x+(u_{n-1,1}y+u_{n-1,0}))x+\cdots)x+((\ldots(u_{0,n}y+u_{0,n-1})y+\cdots)y+u_{0,0})$. [For a "homogeneous" polynomial, such as $u_n x^n + u_{n-1}x^{n-1}y + \cdots + u_1 xy^{n-1} + u_0 y^n$, another scheme is more efficient: if $0 < |x| \le |y|$, first divide x by y, evaluate a polynomial in x/y, then multiply by y^n.]

4. Rule (2) involves $4n$ or $3n$ real multiplications and $4n$ or $7n$ real additions; (3) is worse, it takes $4n+2$ or $4n+1$ mults, $4n+2$ or $4n+5$ adds.

5. One multiplication to compute x^2; $\lfloor n/2 \rfloor$ multiplications and $\lfloor n/2 \rfloor$ additions to evaluate the first line; $\lceil n/2 \rceil$ multiplications and $\lceil n/2 \rceil - 1$ additions to evaluate the second line; and one addition to add the two lines together. Total: $n+1$ multiplications and n additions.

6. **J1.** Compute and store the values $x_0^2, \ldots, x_0^{\lceil n/2 \rceil}$.

J2. Set $v_j \leftarrow u_j x_0^{j-\lfloor n/2 \rfloor}$ for $0 \le j \le n$.

J3. For $k = 0, 1, \ldots, n-1$, set $v_j \leftarrow v_j + v_{j+1}$ for $j = n-1, \ldots, k+1, k$.

J4. Set $v_j \leftarrow v_j x_0^{\lfloor n/2 \rfloor - j}$ for $0 \le j \le n$. ∎

There are $(n^2+n)/2$ additions, $n+\lceil n/2 \rceil - 1$ multiplications, n divisions. Another multiplication and division can be saved by treating v_n and v_0 as special cases. *Reference: SIGACT News* **7**, 3 (Summer 1975), 32–34.

7. Let $x_j = x_0 + jh$, and consider (42), (44). Set $y_j \leftarrow u(x_j)$ for $0 \le j \le n$. For $k = 1, 2, \ldots, n$ (in this order), set $y_j \leftarrow y_j - y_{j-1}$ for $j = k, k+1, \ldots, n$ (in this order). Now $\beta_j = y_j$ for all j.

8. See (43).

9. [*Combinatorial Mathematics* (Buffalo: Math. Assoc. of America, 1963), 26–28.] This formula can be regarded as an application of the principle of inclusion and exclusion (Section 1.3.3), since the sum of the terms for $n - \epsilon_1 - \cdots - \epsilon_n = k$ is the sum of all $x_{1j_1}x_{2j_2}\ldots x_{nj_n}$ for which k values of the j_i do not appear. A direct proof can be given by observing that the coefficient of $x_{1j_1}\ldots x_{nj_n}$ is

$$\sum (-1)^{n-\epsilon_1-\cdots-\epsilon_n} \epsilon_{j_1} \ldots \epsilon_{j_n};$$

if the j's are distinct, this equals unity, but if $j_1, \ldots, j_n \neq k$ then it is zero, since the terms for $\epsilon_k = 0$ cancel the terms for $\epsilon_k = 1$.

To evaluate the sum efficiently, we can start with $\epsilon_1 = 1$, $\epsilon_2 = \cdots = \epsilon_n = 0$, and we can then proceed through all combinations of the ϵ's in such a way that only one ϵ changes from one term to the next. (See "Gray code" in Chapter 7.) The work to compute the first term is $n-1$ multiplications; the subsequent $2^n - 2$ terms each involve n additions, then $n-1$ multiplications, then one more addition. Total: $(2^n-1)(n-1)$ multiplications, and $(2^n-2)(n+1)$ additions. Only $n+1$ temp storage locations are needed, one for the main partial sum and one for each factor of the current product.

10. $\sum_{1\le k<n}(k+1)\binom{n}{k+1} = n(2^{n-1}-1)$ multiplications and $\sum_{1\le k<n} k\binom{n}{k+1} = n2^{n-1}-2^n+1$ additions. This is approximately half as many arithmetic operations as the method of exercise 9, although it requires a more complicated program to control the sequence. Approximately $\binom{n}{\lceil n/2\rceil}+\binom{n}{\lceil n/2\rceil-1}$ temporary storage locations must be used, and this grows exponentially large (on the order of $2^n/\sqrt{n}$).

The method in this exercise is equivalent to the unusual matrix factorization of the permanent function given by Jurkat and Ryser in *J. Algebra* **5** (1967), 342–357. It may also be regarded as an application of (39) and (40), in an appropriate sense.

12. Here is a brief summary of progress on this famous research problem: J. Hopcroft and L. R. Kerr proved, among other things, that seven multiplications are necessary in 2×2 matrix multiplication [*SIAM J. Appl. Math.* **20** (1971), 30–36]. R. L. Probert showed that all 7-multiplication schemes, in which each multiplication takes a linear combination of elements from one matrix and multiplies by a linear combination of elements from the other, must have at least 15 additions [*SIAM J. Computing* **5** (1976), 187–203]. For $n=3$, the best method known is due to J. D. Laderman [*Bull. Amer. Math. Soc.* **82** (1976), 126–128], who showed that 23 noncommutative multiplications suffice. His construction has been generalized by Ondrej Sýkora, who exhibited a method requiring $n^3-(n-1)^2$ noncommutative multiplications and $n^3-n^2+11(n-1)^2$ additions, a result that also reduces to (36) when $n=2$ [*Lecture Notes in Comp. Sci.* **53** (1977), 504–512]. The best lower bound known to hold for all n is the fact that $2n^2-1$ nonscalar multiplications are necessary [Jean-Claude Lafon and S. Winograd, *Theoretical Comp. Sci.*, to appear]. Pan has generalized this to a lower bound of $mn+ns+m-n-1$ in the $m\times n\times s$ case [see *SIAM J. Computing* **9** (1980), 341]. The best upper bounds known for large n were changing rapidly as this book was being prepared for publication; see exercises 60–64.

13. By summing geometric series, we find that $F(t_1,\ldots,t_n)$ equals

$$\sum_{0\le s_1<m_1,\ldots,0\le s_n<m_n} \exp(-2\pi i(s_1t_1/m_1+\cdots+s_nt_n/m_n)f(s_1,\ldots,s_n))/m_1\ldots m_n.$$

The inverse transform times $m_1\ldots m_n$ can be found by doing a regular transform and interchanging t_j with m_j-t_j when $t_j\neq 0$; cf. exercise 4.3.3–9.

[If we regard $F(t_1,\ldots,t_n)$ as the coefficient of $x_1^{t_1}\ldots x_n^{t_n}$ in a multivariate polynomial, the finite Fourier transform amounts to evaluation of this polynomial at roots of unity, and the inverse transform amounts to finding the interpolating polynomial.]

14. Let $m_1=\cdots=m_n=2$, $F(t_1,t_2,\ldots,t_n)=F(2^{n-1}t_n+\cdots+2t_2+t_1)$, and $f(s_1,s_2,\ldots,s_n)=f(2^{n-1}s_1+2^{n-2}s_2+\cdots+s_n)$; note the reversed treatment between t's and s's. Also let $g_k(s_k,\ldots,s_n,t_k)$ be ω raised to the $2^{k-1}t_k(s_n+2s_{n-1}+\cdots+2^{n-k}s_k)$ power.

At each iteration we essentially take 2^{n-1} pairs of complex numbers (α,β) and replace them by $(\alpha+\zeta\beta,\alpha-\zeta\beta)$, where ζ is a suitable power of ω, hence $\zeta=\cos\theta+i\sin\theta$ for some θ. If we take advantage of simplifications when $\zeta=\pm 1$ or $\pm i$, the total work comes to $((n-3)\cdot 2^{n-1}+2)$ complex multiplications and $n\cdot 2^n$ complex additions; the techniques of exercise 41 can be used to reduce the real multiplications and additions used to implement these complex operations.

The number of complex multiplications can be reduced about 25 per cent without changing the number of additions by combining passes k and $k+1$ for $k=1,3,\ldots$; this means that 2^{n-2} quadruples $(\alpha,\beta,\gamma,\delta)$ are being replaced by $(\alpha+\zeta\beta+\zeta^2\gamma+\zeta^3\delta,\ \alpha+i\zeta\beta-\zeta^2\gamma-i\zeta^3\delta,\ \alpha-\zeta\beta+\zeta^2\gamma-\zeta^3\delta,\ \alpha-i\zeta\beta-\zeta^2\gamma+i\zeta^3\delta)$. The number of complex multiplications when n is even is thereby reduced to $(3n-2)2^{n-3}-5\lfloor 2^{n-1}/3\rfloor$.

These calculations assume that the given numbers $F(t)$ are complex. If the $F(t)$ are real, then $f(s)$ is the complex conjugate of $f(2^n - s)$, so we can avoid the redundancy by computing only the 2^n independent real numbers $f(0)$, $\Re f(1)$, ..., $\Re f(2^{n-1} - 1)$, $f(2^{n-1})$, $\Im f(1)$, ..., $\Im f(2^{n-1} - 1)$. The entire calculation in this case can be done by working with 2^n real values, using the fact that $f^{[k]}(s_{n-k+1}, \ldots, s_n, t_1, \ldots, t_{n-k})$ will be the complex conjugate of $f^{[k]}(s'_{n-k+1}, \ldots, s'_n, t_1, \ldots, t_{n-k})$ when $(s_1 \ldots s_n)_2 + (s'_1 \ldots s'_n)_2 \equiv 0$ (modulo 2^n). About half as many multiplications and additions are needed as in the complex case.

[The fast Fourier transform algorithm is essentially due to C. Runge and H. König in 1924, and it was generalized by J. W. Cooley and J. W. Tukey, ***Math. Comp.* 19 (1965), 297–301**. Its interesting history has been traced by J. W. Cooley, P. A. W. Lewis, and P. D. Welch, *Proc. IEEE* **55** (1967), 1675–1677. Details concerning its use have been discussed by R. C. Singleton, *CACM* **10** (1967), 647–654; M. C. Pease, *JACM* **15** (1968), 252–264; **G. D. Berglund, *Math. Comp.* 22 (1968), 275–279**, *CACM* **11** (1968), 703–710; A. M. Macnaghten and C. A. R. Hoare, *Comp. J.* **20** (1977), 78–83. See also exercises 53, 57, and 59.]

15. (a) The hint follows by integration and induction. Let $f^{(n)}(\theta)$ take on all values between A and B inclusive, as θ varies from $\min(x_0, \ldots, x_n)$ to $\max(x_0, \ldots, x_n)$. Replacing $f^{(n)}$ by each of these bounds, in the stated integral, yields $A/n! \le f(x_0, \ldots, x_n) \le B/n!$. (b) It suffices to prove this for $j = n$. Let f be Newton's interpolation polynomial, then $f^{(n)}$ is the constant $n!\alpha_n$.

16. Carry out the multiplications and additions of (43) as operations on polynomials. (The special case $x_0 = x_1 = \cdots = x_n$ is considered in exercise 2. We have used this method in step C8 of Algorithm 4.3.3C.)

17. T. M. Vari has shown that $n - 1$ multiplications are necessary, by proving that n multiplications are necessary to compute $x_1^2 + \cdots + x_n^2$ [Cornell Computer Science Report 120 (Jan. 1972)].

18. $\alpha_0 = \frac{1}{2}(u_3/u_4 + 1)$, $\beta = u_2/u_4 - \alpha_0(\alpha_0 - 1)$, $\alpha_1 = \alpha_0\beta - u_1/u_4$, $\alpha_2 = \beta - 2\alpha_1$, $\alpha_3 = u_0/u_4 - \alpha_1(\alpha_1 + \alpha_2)$, $\alpha_4 = u_4$.

19. Since α_5 is the leading coefficient, we may assume without loss of generality that $u(x)$ is monic (i.e., $u_5 = 1$). Then α_0 is a root of the cubic equation $40z^3 - 24u_4z^2 + (4u_4^2 + 2u_3)z + (u_2 - u_3u_4) = 0$; this equation always has at least one real root, and it may have three. Once α_0 is determined, we have $\alpha_3 = u_4 - 4\alpha_0$, $\alpha_1 = u_3 - 4\alpha_0\alpha_3 - 6\alpha_0^2$, $\alpha_2 = u_1 - \alpha_0(\alpha_0\alpha_1 + 4\alpha_0^2\alpha_3 + 2\alpha_1\alpha_3 + \alpha_0^3)$, $\alpha_4 = u_0 - \alpha_3(\alpha_0^4 + \alpha_1\alpha_0^2 + \alpha_2)$.

For the given polynomial we are to solve the cubic equation $40z^3 - 120z^2 + 80z = 0$; this leads to three solutions $(\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5) = (0, -10, 13, 5, -5, 1)$, $(1, -20, 68, 1, 11, 1)$, $(2, -10, 13, -3, 27, 1)$.

20.
```
LDA   X                 FADD  =α1=
FADD  =α3=              FMUL  TEMP2
STA   TEMP1             FADD  =α2=
FADD  =α0-α3=           FMUL  TEMP1
STA   TEMP2             FADD  =α4=
FMUL  TEMP2             FMUL  =α5=
STA   TEMP2
```

21. $z = (x+1)x-2$, $w = (x+5)z+9$, $u(x) = (w+z-8)w-8$; or $z = (x+9)x+26$, $w = (x-3)z + 73$, $u(x) = (w + z - 24)w - 12$.

22. $\alpha_6 = 1$, $\alpha_0 = -1$, $\alpha_1 = 1$, $\beta_1 = -2$, $\beta_2 = -2$, $\beta_3 = -2$, $\beta_4 = 1$, $\alpha_3 = -4$, $\alpha_2 = 0$, $\alpha_4 = 4$, $\alpha_5 = -2$. We form $z = (x-1)x + 1$, $w = z + x$, and $u(x) = ((z - x - 4)w + 4)z - 2$. In this special case we see that one of the seven additions can be saved if we compute $w = x^2 + 1$, $z = w - x$.

23. (a) We may use induction on n; the result is trivial if $n < 2$. If $f(0) = 0$, then the result is true for the polynomial $f(z)/z$, so it holds for $f(z)$. If $f(iy) = 0$ for some real $y \neq 0$, then $g(+iy) = h(+iy) = 0$; since the result is true for $f(z)/(z^2 + y^2)$, it holds

[redacted]

of $f(z)$ inside the region, which is at most 1. When R is large, the path $f(Re^{it})$ for $\pi/2 \leq t \leq 3\pi/2$ will circle the origin clockwise approximately $n/2$ times; so the path $f(it)$ for $-R \leq t \leq R$ must go counterclockwise around the origin at least $n/2 - 1$ times. For n even, this implies that $f(it)$ crosses the imaginary axis at least $n-2$ times, and the real axis at least $n - 3$ times; for n odd, $f(it)$ crosses the real axis at least $n - 2$ times and the imaginary axis at least $n - 3$ times. These are roots respectively of $g(it) = 0$, $h(it) = 0$.

(b) If not, g or h would have a root of the form $a + bi$ with $a \neq 0$ and $b \neq 0$. But this would imply the existence of at least three other such roots, namely $a - bi$ and $-a \pm bi$, while $g(z)$ and $h(z)$ have at most n roots.

24. The roots of u are -7, $-3 \pm i$, $-2 \pm i$, and -1; permissible values of c are 2 and 4 (but *not* 3, since $c = 3$ makes the sum of the roots equal to zero). Case 1, $c = 2$: $p(x) = (x+5)(x^2+2x+2)(x^2+1)(x-1) = x^6 + 6x^5 + 6x^4 + 4x^3 - 5x^2 - 2x - 10$; $q(x) = 6x^2 + 4x - 2 = 6(x+1)(x - \frac{1}{3})$. Let $\alpha_2 = -1$, $\alpha_1 = \frac{1}{3}$; $p_1(x) = x^4 + 6x^3 + 5x^2 - 2x - 10 = (x^2 + 6x + \frac{16}{3})(x^2 - \frac{1}{3}) - \frac{74}{9}$; $\alpha_0 = 6$, $\beta_0 = \frac{16}{3}$, $\beta_1 = -\frac{74}{9}$. Case 2, $c = 4$: A similar analysis gives $\alpha_2 = 9$, $\alpha_1 = -3$, $\alpha_0 = -6$, $\beta_0 = 12$, $\beta_1 = -26$.

25. $\beta_1 = \alpha_2$, $\beta_2 = 2\alpha_1$, $\beta_3 = \alpha_7$, $\beta_4 = \alpha_6$, $\beta_5 = \beta_6 = 0$, $\beta_7 = \alpha_1$, $\beta_8 = 0$, $\beta_9 = 2\alpha_1 - \alpha_8$.

26. (a) $\lambda_1 = \alpha_1 \times \lambda_0$, $\lambda_2 = \alpha_2 + \lambda_1$, $\lambda_3 = \lambda_2 \times \lambda_0$, $\lambda_4 = \alpha_3 + \lambda_3$, $\lambda_5 = \lambda_4 \times \lambda_0$, $\lambda_6 = \alpha_4 + \lambda_5$. (b) $\kappa_1 = 1 + \beta_1 x$, $\kappa_2 = 1 + \beta_2\kappa_1 x$, $\kappa_3 = 1 + \beta_3\kappa_2 x$, $u(x) = \beta_4\kappa_3 = \beta_1\beta_2\beta_3\beta_4 x^3 + \beta_2\beta_3\beta_4 x^2 + \beta_3\beta_4 x + \beta_4$. (c) If any coefficient is zero, the coefficient of x^3 must also be zero in (b), while (a) yields an arbitrary polynomial $\alpha_1 x^3 + \alpha_2 x^2 + \alpha_3 x + \alpha_4$ of degree ≤ 3.

27. Otherwise there would be a nonzero polynomial $f(q_n, \ldots, q_1, q_0)$ with integer coefficients such that $q_n \cdot f(q_n, \ldots, q_1, q_0) = 0$ for all sets $(q_n, \ldots, q_0)$ of real numbers. This cannot happen, since it is easy to prove by induction on n that a nonzero polynomial always takes on some nonzero value. (Cf. exercise 4.6.1–16. However, this result is false for *finite* fields in place of the real numbers.)

28. The indeterminate quantities $\alpha_1, \ldots, \alpha_s$ form an algebraic basis for the polynomial domain $Q[\alpha_1, \ldots, \alpha_s]$, where Q is the field of rational numbers. Since $s + 1$ is greater than the number of elements in a basis, the polynomials $f_j(\alpha_1, \ldots, \alpha_s)$ are algebraically dependent; this means that there is a nonzero polynomial g with rational coefficients such that $g(f_0(\alpha_1, \ldots, \alpha_s), \ldots, f_s(\alpha_1, \ldots, \alpha_s))$ is identically zero.

29. Given $j_0, \ldots, j_t \in \{0, 1, \ldots, n\}$, there are nonzero polynomials with integer coefficients such that $g_j(q_{j_0}, \ldots, q_{j_t}) = 0$ for all $(q_n, \ldots, q_0)$ in R_j, $1 \leq j \leq m$. The product $g_1 g_2 \ldots g_m$ is therefore zero for all $(q_n, \ldots, q_0)$ in $R_1 \cup \cdots \cup R_m$.

30. Starting with the construction in Theorem M, we will prove that $m_p+(1-\delta_{0m_c})$ of the β's may effectively be eliminated: If μ_i corresponds to a parameter multiplication, we have $\mu_i = \beta_{2i-1} \times (T_{2i} + \beta_{2i})$; add $c\beta_{2i-1}\beta_{2i}$ to each β_j for which $c\mu_i$ occurs in T_j, and replace β_{2i} by zero. This removes one parameter for each parameter multiplication. If μ_i is the first chain multiplication, then μ_i is the first chain multiplication, then $\mu_i = (\gamma_1 x + \theta_1 + \beta_{2i-1}) \times (\gamma_{2x} + \theta_2 + \beta_{2i})$, where γ_1, γ_2, θ_1, θ_2 are polynomials in β_1, $\ldots$, β_{2i-2} with integer coefficients. Here θ_1 and θ_2 can be "absorbed" into β_{2i-1} and β_{2i}, respectively, so we may assume that $\theta_1 = \theta_2 = 0$. Now add $c\beta_{2i-1}\beta_{2i}$ to each β_j for which $c\mu_i$ occurs in T_j; add $\beta_{2i-1}\gamma_2/\gamma_1$ to β_{2i}; and set β_{2i-1} to zero. The result set is unchanged by this elimination of β_{2i-1}, except for the values of $\alpha_1, \ldots, \alpha_s$ such that γ_1 is zero. [This proof is essentially due to V. Îa. Pan, *Russian Mathematical Surveys* **21**,1 (1966), 105–136.] The latter case can be handled as in the proof of Theorem A, since the polynomials with $\gamma_1 = 0$ can be evaluated by eliminating β_{2i} (as in the first construction, where μ_i corresponds to a parameter multiplication).

31. Otherwise we could add one parameter multiplication as a final step, and violate Theorem C. (The exercise is an improvement over Theorem A, in this special case, since there are only n degrees of freedom in the coefficients of a monic polynomial of degree n.)

32. $\lambda_1 = \lambda_0 \times \lambda_0$, $\lambda_2 = \alpha_1 \times \lambda_1$, $\lambda_3 = \alpha_2 + \lambda_2$, $\lambda_4 = \lambda_3 \times \lambda_1$, $\lambda_5 = \alpha_3 + \lambda_4$. We need at least three multiplications to compute u_4x^4 (see Section 4.6.3), and at least two additions by Theorem A.

33. We must have $n+1 \le 2m_c+m_p+\delta_{0m_c}$, and $m_c+m_p = (n+1)/2$; so there are no parameter multiplications. Now the first λ_i whose leading coefficient (as a polynomial in x) is not an integer must be obtained by a chain addition; and there must be at least $n+1$ parameters, so there are at least $n+1$ parameter additions.

34. Transform the given chain step by step, and also define the "content" c_i of λ_i, as follows: (Intuitively, c_i is the leading coefficient of λ_i.) Define $c_0 = 1$. (a) If the step has the form $\lambda_i = \alpha_j + \lambda_k$, replace it by $\lambda_i = \beta_j + \lambda_k$, where $\beta_j = \alpha_j/c_k$; and define $c_i = c_k$. (b) If the step has the form $\lambda_i = \alpha_j - \lambda_k$, replace it by $\lambda_i = \beta_j + \lambda_k$, where $\beta_j = -\alpha_j/c_k$; and define $c_i = -c_k$. (c) If the step has the form $\lambda_i = \alpha_j \times \lambda_k$, replace it by $\lambda_i = \lambda_k$ (the step will be deleted later); and define $c_i = \alpha_j c_k$. (d) If the step has the form $\lambda_i = \lambda_j \times \lambda_k$, leave it unchanged; and define $c_i = c_j c_k$.

After this process is finished, delete all steps of the form $\lambda_i = \lambda_k$, replacing λ_i by λ_k in each future step that uses λ_j. Then add a final step $\lambda_{r+1} = \beta \times \lambda_r$, where $\beta = c_r$. This is the desired scheme, since it is easy to verify that the new λ_i are just the old ones divided by the factor c_i. The β's are given functions of the α's; division by zero is no problem, because if any $c_k = 0$ we must have $c_r = 0$ (hence the coefficient of x^n is zero), or else λ_k never contributes to the final result.

35. Since there are at least five parameter steps, the result is trivial unless there is at least one parameter multiplication; considering the ways in which three multiplications can form u_4x^4, we see that there must be one parameter multiplication and two chain multiplications. Therefore the four addition-subtractions must each be parameter steps, and exercise 34 applies. We can now assume that only additions are used, and that we have a chain to compute a general *monic* fourth-degree polynomial with *two* chain multiplications and four parameter additions. The only possible scheme of this type

that calculates a fourth-degree polynomial has the form

$$\begin{aligned}\lambda_1 &= \alpha_1 + \lambda_0\\ \lambda_2 &= \alpha_2 + \lambda_0\\ \lambda_3 &= \lambda_1 \times \lambda_2\\ \lambda_4 &= \alpha_3 + \lambda_3\\ \lambda_5 &= \alpha_4 + \lambda_3\\ \lambda_6 &= \lambda_4 \times \lambda_5\\ \lambda_7 &= \alpha_5 + \lambda_6\end{aligned}$$

Actually this chain has one addition too many, but any correct scheme can be put into this form if we restrict some of the α's to be functions of the others. Now λ_7 has the form $(x^2+Ax+B)(x^2+Ax+C)+D = x^4+2Ax^3+(E+A^2)x^2+EAx+F$, where $A=\alpha_1+\alpha_2$, $B=\alpha_1\alpha_2+\alpha_3$, $C=\alpha_1\alpha_2+\alpha_4$, $D=\alpha_6$, $E=B+C$, $F=BC+D$; and since this involves only three independent parameters it cannot represent a general monic fourth-degree polynomial.

36. As in the solution to exercise 35, we may assume that the chain computes a general monic polynomial of degree six, using only three chain multiplications and six parameter additions. The computation must take one of two general forms

$$\begin{aligned}\lambda_1 &= \alpha_1 + \lambda_0\\ \lambda_2 &= \alpha_2 + \lambda_0\\ \lambda_3 &= \lambda_1 \times \lambda_2\\ \lambda_4 &= \alpha_3 + \lambda_0\\ \lambda_5 &= \alpha_4 + \lambda_3\\ \lambda_6 &= \lambda_4 \times \lambda_5\\ \lambda_7 &= \alpha_5 + \lambda_6\\ \lambda_8 &= \alpha_6 + \lambda_6\\ \lambda_9 &= \lambda_7 \times \lambda_8\\ \lambda_{10} &= \alpha_7 + \lambda_9\end{aligned}\qquad\qquad\begin{aligned}\lambda_1 &= \alpha_1 + \lambda_0\\ \lambda_2 &= \alpha_2 + \lambda_0\\ \lambda_3 &= \lambda_1 \times \lambda_2\\ \lambda_4 &= \alpha_3 + \lambda_3\\ \lambda_5 &= \alpha_4 + \lambda_3\\ \lambda_6 &= \lambda_4 \times \lambda_5\\ \lambda_7 &= \alpha_5 + \lambda_3\\ \lambda_8 &= \alpha_6 + \lambda_6\\ \lambda_9 &= \lambda_7 \times \lambda_8\\ \lambda_{10} &= \alpha_7 + \lambda_9\end{aligned}$$

where, as in exercise 35, an extra addition has been inserted to cover a more general case. Neither of these schemes can calculate a general sixth-degree monic polynomial, since the first case is a polynomial of the form

$$(x^3+Ax^2+Bx+C)(x^3+Ax^2+Bx+D)+E,$$

and the second case (cf. exercise 35) is a polynomial of the form

$$(x^4+2Ax^3+(E+A^2)x^2+EAx+F)(x^2+Ax+G)+H;$$

both of these involve only five independent parameters.

37. Let $u^{[0]}(x) = u_n x^n + u_{n-1}x^{n-1} + \cdots + u_0$, $v^{[0]}(x) = x^n + v_{n-1}x^{n-1} + \cdots + v_0$. For $1 \le j \le n$, divide $u^{[j-1]}(x)$ by the monic polynomial $v^{[j-1]}(x)$, obtaining $u^{[j-1]}(x) = \alpha_j v^{[j-1]}(x) + \beta_j v^{[j]}(x)$. Assume that a monic polynomial $v^{[j]}(x)$ of degree $n-j$ exists satisfying this relation; this will be true for almost all rational functions. Let $u^{[j]}(x) = v^{[j-1]}(x) - xv^{[j]}(x)$. These definitions imply that $\deg(u^{[n]}) < 1$, so we may let $\alpha_{n+1} = u^{[n]}(x)$.

For the given rational function we have

α_j	β_j	$v^{[j]}(x)$	$u^{[j]}(x)$
1	2	$x+5$	$3x+19$
3	4	1	5

so $u^{[0]}(x)/v^{[0]}(x) = 1 + 2/(x + 3 + 4/(x + 5))$.

Notes: A general rational function of the stated form has $2n + 1$ "degrees of freedom," in the sense that it can be shown to have $2n + 1$ essentially independent parameters. If we generalize polynomial chains to "arithmetic chains," which allow division operations as well as addition, subtraction, and multiplication, we can obtain the following results with slight modifications to the proofs of Theorems A and M: *An arithmetic chain with q addition-subtraction steps has at most $q+1$ degrees of freedom. An arithmetic chain with m multiplication-division steps has at most $2m + 1$ degrees of freedom.* Therefore an arithmetic chain that computes almost all rational functions of the stated form must have at least $2n$ addition-subtractions, and n multiplication-divisions; the method in this exercise is "optimal."

38. The theorem is certainly true if $n = 0$. Assume that n is positive, and that a polynomial chain computing $P(x; u_0, \ldots, u_n)$ is given, where each of the parameters α_j has been replaced by a real number. Let $\lambda_i = \lambda_j \times \lambda_k$ be the first chain multiplication step that involves one of $u_0, \ldots, u_n$; such a step must exist because of the rank of A. Without loss of generality, we may assume that λ_j involves u_n; thus, λ_j has the form $h_0u_0 + \cdots + h_nu_n + f(x)$, where $h_0, \ldots, h_n$ are real, $h_n \neq 0$, and $f(x)$ is a polynomial with real coefficients. (The h's and the coefficients of $f(x)$ are derived from the values assigned to the α's.)

Now change step i to $\lambda_i = \alpha \times \lambda_k$, where α is an arbitrary real number. (We could take $\alpha = 0$; general α is used here merely to show that there is a certain amount of flexibility available in the proof.) Add further steps to calculate

$$\lambda = (\alpha - f(x) - h_0u_0 - \cdots - h_{n-1}u_{n-1})/h_n;$$

these new steps involve only additions and parameter multiplications (by suitable new parameters). Finally, replace $\lambda_{-n-1} = u_n$ everywhere in the chain by this new element λ. The result is a chain that calculates

$$Q(x; u_0, \ldots, u_{n-1}) = P(x; u_0, \ldots, u_{n-1}, (\alpha - f(x) - h_0u_0 - \cdots - h_{n-1}u_{n-1})/h_n);$$

and this chain has one less chain multiplication. The proof will be complete if we can show that Q satisfies the hypotheses. The quantity $(\alpha - f(x))/h_n$ leads to a possibly increased value of m, and a new vector B'. If the columns of A are A_0, A_1, $\ldots$, A_n (these vectors being linearly independent over the reals), the new matrix A' corresponding to Q has the column vectors

$$A_0 - (h_0/h_n)A_n, \qquad \ldots, \qquad A_{n-1} - (h_{n-1}/h_n)A_n,$$

plus perhaps a few rows of zeros to account for an increased value of m, and these columns are clearly also linearly independent. By induction, the chain that computes Q has at least $n - 1$ chain multiplications, so the original chain has at least n.

[Pan showed also that the use of division would give no improvement; cf. *Problemy Kibernetiki* **7** (1962), 21–30. Generalizations to the computation of several polynomials in several variables, with and without various kinds of preconditioning, have been given by S. Winograd, *Comm. Pure and Applied Math.* **23** (1970), 165–179.]

39. By induction on m. Let $w_m(x) = x^{2m} + u_{2m-1}x^{2m-1} + \cdots + u_0$, $w_{m-1}(x) = x^{2m-2} + v_{2m-3}x^{2m-3} + \cdots + v_0$, $a = \alpha_1 + \gamma_m$, $b = \alpha_m$, and let

$$f(r) = \textstyle\sum_{i,j\geq 0}(-1)^{i+j}\binom{i+j}{j}u_{r+i+2j}\,a^i b^j.$$

It follows that $v_r = f(r+2)$ for $r \geq 0$, and $\delta_m = f(1)$. If $\delta_m = 0$ and a is given, we have a polynomial of degree $m-1$ in b, with leading coefficient $\pm(u_{2m-1} - ma) = \pm(\gamma_2 + \cdots + \gamma_m - m\gamma_m)$.

In Motzkin's unpublished notes he arranged to make $\delta_k = 0$ almost always, by choosing γ's so that this leading coefficient is $\neq 0$ when m is even and $= 0$ when m is odd; then we almost always can let b be a (real) root of an odd-degree polynomial.

40. No; S. Winograd found a way to compute all polynomials of degree 13 with only 7 (possibly complex) multiplications [*Comm. Pure and Applied Math.* **25** (1972), 455–457]. L. Revah found schemes that evaluate almost all polynomials of degree $n \geq 9$ with $\lfloor n/2 \rfloor + 1$ (possibly complex) multiplications [*SIAM J. Computing* **4** (1975), 381–392]; she also showed that when $n = 9$ it is possible to achieve $\lfloor n/2 \rfloor + 1$ multiplications only with at least $n+3$ additions. By appending sufficiently many additions (cf. exercise 39), the "almost all" and "possibly complex" provisos disappear. V. Ia. Pan [*Proc. ACM Symp. Theory Comp.* **10** (1978), 162–172; IBM Research Report RC7754 (1979)] found schemes with $\lfloor n/2 \rfloor + 1$ (complex) multiplications and the minimum number $n+2+\delta_{n9}$ of (complex) additions, for all odd $n \geq 9$; his method for $n = 9$ is

$$\begin{aligned} v(x) &= ((x+\alpha)^2 + \beta)(x+\gamma), \qquad w(x) = v(x) + x, \\ t(x) &= (v(x)+\delta)(w(x)+\epsilon) - (v(x)+\delta')(w(x)+\epsilon'), \\ u(x) &= (v(x)+\varsigma)(t(x)+\eta) + \kappa. \end{aligned}$$

The minimum number of *real* additions necessary, when the minimum number of (real) multiplications is achieved, remains unknown for $n \geq 9$.

41. $a(c+d) - (a+b)d + i(a(c+d) + (b-a)c)$. [Beware numerical instability. Three multiplications are necessary, since complex multiplication is a special case of (69) with $p(u) = u^2 + 1$. Without the restriction on additions there are other possibilities. For example, the symmetric formula $ac - bd + i((a+b)(c+d) - ac - bd)$ was suggested by Peter Ungar in 1963; cf. Eq. 4.3.3–2 with 2^n replaced by i. See I. Munro, *Proc. ACM Symp. Theory Comp.* **3** (1960), 40–44; S. Winograd, *Linear Alg. Appl.* **4** (1971), 381–388.]

Alternatively, if $a^2 + b^2 = 1$ and $t = (1-a)/b = b/(1+a)$, the algorithm "$w = c - td$, $v = d + bw$, $u = w - tv$" for calculating the product $(a+bi)(c+di) = u+iv$ has been suggested by Oscar Buneman [*J. Comp. Phys.* **12** (1973), 127–128]. In this method if $a = \cos\theta$ and $b = \sin\theta$, we have $t = \tan(\theta/2)$.

[Helmut Alt and Jan van Leeuwen have shown that four real multiplications or divisions are necessary for computing $1/(a+bi)$, and four are sufficient for computing $a/(b+ci)$. It is unknown whether $(a+bi)/(c+di)$ can be computed with only five multiplications or divisions.]

42. (a) Let $\pi_1, \ldots, \pi_m$ be the λ_i's that correspond to chain multiplications; then $\pi_i = P_{2i-1} \times P_{2i}$ and $u(x) = P_{2m+1}$, where each P_j has the form $\beta_j + \beta_{j0}x + \beta_{j1}\pi_1 + \cdots + \beta_{jr(j)}\pi_{r(j)}$, where $r(j) \leq \lceil j/2 \rceil - 1$ and each of the β_j and β_{jk} is a polynomial in the α's with integer coefficients. We can systematically modify the chain (cf. exercise 30) so that $\beta_j = 0$ and $\beta_{jr(j)} = 1$, for $1 \leq j \leq 2m$; furthermore we can assume that

$\beta_{30} = 0$. The result set now has at most $m + 1 + \sum_{1 \le j \le 2m}(\lceil j/2 \rceil - 1) = m^2 + 1$ degrees of freedom.

(b) Any such polynomial chain with at most m chain multiplications can be simulated by one with the form considered in (a), except that now we let $r(j) = \lceil j/2 \rceil - 1$ for $1 \le j \le 2m + 1$, and we do not assume that $\beta_{30} = 0$ or that $\beta_{jr(j)} = 1$ for $j \ge 3$. This single canonical form involves $m^2 + 2m$ parameters. As the α's run through all integers and as we run through all chains, the β's run through at most 2^{m^2+2m} sets of values mod 2, hence the result set does also. In order to obtain all 2^n polynomials of degree n with 0–1 coefficients, we need $m^2 + 2m \ge n$.

(c) Set $m \leftarrow \lfloor \sqrt{n} \rfloor$ and compute $x^2, x^3, \ldots, x^m$. Let $u(x) = u_{m+1}(x)x^{(m+1)m} + \cdots + u_1(x)x^m + u_0(x)$, where each $u_j(x)$ is a polynomial of degree $\le m$ with integer coefficients (hence it can be evaluated without any more multiplications). Now evaluate $u(x)$ by rule (2) as a polynomial in x^m with known coefficients. (The number of additions used is approximately the sum of the absolute values of the coefficients, so this algorithm is efficient on 0–1 polynomials. Paterson and Stockmeyer also gave another algorithm that uses about $\sqrt{2n}$ multiplications.)

Reference: *SIAM J. Computing* **2** (1973), 60–66; see also J. E. Savage, *SIAM J. Computing* **3** (1974), 150–158. For analogous results about additions, see Borodin and Cook, *SIAM J. Computing* **5** (1976), 146–157; Rivest and Van de Wiele, *Inf. Proc. Letters* **8** (1979), 178–180.]

43. When $a_i = a_j + a_k$ is a step in some optimal addition chain for $n + 1$, compute $x^i = x^j x^k$ and $p_i = p_k x^j + p_j$, where $p_i = x^{i-1} + \cdots + x + 1$; omit the final calculation of x^{n+1}. We save one multiplication whenever $a_k = 1$, in particular when $i = 1$. (Cf. exercise 4.6.3–31 with $\epsilon = \frac{1}{2}$.)

44. It suffices to show that (T_{ijk})'s rank is *at most* that of (t_{ijk}), since we can obtain (t_{ijk}) back from (T_{ijk}) by transforming it in the same way with F^{-1}, G^{-1}, H^{-1}. If $t_{ijk} = \sum_{1 \le l \le r} a_{il} b_{jl} c_{kl}$ then it follows immediately that

$$T_{ijk} = \sum_{1 \le l \le r}(\sum_{1 \le p \le m} F_{ip} a_{pl})(\sum_{1 \le q \le n} G_{jq} b_{ql})(\sum_{1 \le r \le s} H_{kr} c_{rl}).$$

[H. F. de Groote has proved that all normal schemes that yield 2×2 matrix products with seven chain multiplications are equivalent, in the sense that they can be obtained from each other by nonsingular matrix multiplication as in this exercise. In this sense Strassen's algorithm is unique.]

45. By exercise 44 we can add any multiple of a row, column, or plane to another one without changing the rank; we can also multiply a row, column, or plane by a nonzero constant, or transpose the tensor. A sequence of such operations can always be found to reduce a give $2 \times 2 \times 2$ tensor to one of the forms $\binom{0\,0}{0\,0}\binom{0\,0}{0\,0}$, $\binom{1\,0}{0\,0}\binom{0\,0}{0\,0}$, $\binom{1\,0}{0\,1}\binom{0\,0}{0\,0}$, $\binom{1\,0}{0\,0}\binom{0\,0}{0\,1}$, $\binom{1\,0}{0\,1}\binom{0\,1}{q\,r}$. The last tensor has rank 3 or 2 according as the polynomial $u^2 - ru - q$ has one or two irreducible factors in the field of interest, by Theorem W (cf. (72)).

46. A general $m \times n \times s$ tensor has mns degrees of freedom. By exercise 28 it is impossible to express all $m \times n \times s$ tensors in terms of the $(m + n + s)r$ elements of a realization A, B, C unless $(m + n + s)r \ge mns$. On the other hand, assume that $m \ge n \ge s$. The rank of an $m \times n$ matrix is at most n, so we can realize any tensor in ns chain multiplications by realizing each matrix plane separately. [Exercise 45 shows that this lower bound on the maximum tensor rank is not best possible, nor is the upper bound. Thomas D. Howell (Ph. D. thesis, Cornell Univ., 1976) has shown that there are tensors of rank $\ge \lceil mns/(m + n + s - 2) \rceil$ over the complex numbers.]

48. If A, B, C and A', B', C' are realizations of (t_{ijk}) and (t'_{ijk}) of respective lengths r and r', then $A'' = A \oplus A'$, $B'' = B \oplus B'$, $C'' = C \oplus C'$, and $A''' = A \otimes A'$, $B''' = B \otimes B'$, $C''' = C \otimes C'$, are realizations of (t''_{ijk}) and (t'''_{ijk}) of respective lengths $r + r'$ and $r \cdot r'$.

Note: Many people have made the natural conjecture that $\operatorname{rank}((t_{ijk}) \oplus (t'_{ijk})) = \operatorname{rank}(t_{ijk}) + \operatorname{rank}(t'_{ijk})$, but the construction in exercise 60(b) makes this seem much less plausible than it once was.

49. By Lemma T, $\operatorname{rank}(t_{ijk}) \geq \operatorname{rank}(t_{i(jk)})$. Conversely if M is a matrix of rank r we can transform it by row and column operations, finding nonsingular matrices F and G such that FMG has all entries 0 except for r diagonal elements that are 1; cf. Algorithm 4.6.2N. The tensor rank of FMG is therefore $\leq r$; and it is the same as the tensor rank of M, by exercise 44.

50. Let $i = \langle i', i'' \rangle$ where $1 \leq i' \leq m$ and $1 \leq i'' \leq n$; then $t_{\langle i',i'' \rangle jk} = \delta_{i''j}\delta_{i'k}$, and it is clear that $\operatorname{rank}(t_{i(jk)}) = mn$ since $(t_{i(jk)})$ is a permutation matrix. By Lemma L, $\operatorname{rank}(t_{ijk}) \geq mn$. Conversely, since (t_{ijk}) has only mn nonzero entries, its rank is clearly $\leq mn$. (There is consequently no normal scheme requiring fewer than the mn obvious multiplications. There is no such abnormal scheme either [*Comm. Pure and Appl. Math.* **3** (1970), 165–179]. But some savings can be achieved if the same matrix is used with $s > 1$ different column vectors, since this is equivalent to $(m \times n)$ times $(n \times s)$ matrix multiplication.)

51. (a) $s_1 = y_0 + y_1$, $s_2 = y_0 - y_1$; $m_1 = \frac{1}{2}(x_0 + x_1)s_1$, $m_2 = \frac{1}{2}(x_0 - x_1)s_2$; $w_0 = m_1 + m_2$, $w_1 = m_1 - m_2$. (b) Here are some intermediate steps, using the methodology in the text: $((x_0 - x_2) + (x_1 - x_2)u)((y_0 - y_2) + (y_1 - y_2)u) \bmod (u^2 + u + 1) = ((x_0 - x_2)(y_0 - y_2) - (x_1 - x_2)(y_1 - y_2)) + ((x_0 - x_2)(y_0 - y_2) - (x_1 - x_0)(y_1 - y_0))u$. The first realization is

$$\begin{pmatrix} 1 & 1 & \bar{1} & 0 \\ 1 & 0 & 1 & 1 \\ 1 & \bar{1} & 0 & \bar{1} \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 & \bar{1} & 0 \\ 1 & 0 & 1 & 1 \\ 1 & \bar{1} & 0 & \bar{1} \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 & 1 & \bar{2} \\ 1 & 1 & \bar{2} & 1 \\ 1 & \bar{2} & 1 & 1 \end{pmatrix} \times \frac{1}{3}.$$

The second realization is

$$\begin{pmatrix} 1 & 1 & 1 & \bar{2} \\ 1 & 1 & \bar{2} & 1 \\ 1 & \bar{2} & 1 & 1 \end{pmatrix} \times \frac{1}{3}, \quad \begin{pmatrix} 1 & 1 & \bar{1} & 0 \\ 1 & \bar{1} & 0 & \bar{1} \\ 1 & 0 & 1 & 1 \end{pmatrix}, \quad \begin{pmatrix} 1 & 1 & \bar{1} & 0 \\ 1 & 0 & 1 & 1 \\ 1 & \bar{1} & 0 & \bar{1} \end{pmatrix}.$$

The resulting algorithm computes $s_1 = y_0 + y_1$, $s_2 = y_0 - y_1$, $s_3 = y_2 - y_0$, $s_4 = y_2 - y_1$, $s_5 = s_1 + y_2$; $m_1 = \frac{1}{3}(x_0 + x_1 + x_2)s_5$, $m_2 = \frac{1}{3}(x_0 + x_1 - 2x_2)s_2$, $m_3 = \frac{1}{3}(x_0 - 2x_1 + x_2)s_3$, $m_4 = \frac{1}{3}(-2x_0 + x_1 + x_2)s_4$; $t_1 = m_1 + m_2$, $t_2 = m_1 - m_2$, $t_3 = m_1 + m_3$, $w_0 = t_1 - m_3$, $w_1 = t_3 + m_4$, $w_2 = t_2 - m_4$.

52. Let $i = \langle i', i'' \rangle$ when $i \bmod n' = i'$ and $i \bmod n'' = i''$. Then we wish to compute $w_{\langle k',k'' \rangle} = \sum x_{\langle i',i'' \rangle} y_{\langle j',j'' \rangle}$ summed for $i' + j' \equiv k'$ (modulo n') and $i'' + j'' \equiv k''$ (modulo n''). This can be done by applying the n' algorithm to the $2n'$ vectors $X_{i'}$ and $Y_{j'}$ of length n'', obtaining the n' vectors $W_{k'}$. Each vector addition becomes n'' additions, each parameter multiplication becomes n'' parameter multiplications, and each chain multiplication of vectors is replaced by a cyclic convolution of degree n''. [If the subalgorithms use the minimum number of chain multiplications, this algorithm uses $2(n' - d(n'))(n'' - d(n''))$ more than the minimum, where $d(n)$ is the number of divisors of n.]

53. (a) Let $n(k) = (p-1)p^{e-k-1} = \varphi(p^{e-k})$ for $0 \le k < e$, and $n(k) = 1$ for $k \ge e$. Represent the numbers $\{1, \ldots, m\}$ in the form $a^i p^k$ (modulo m), where $0 \le k \le e$ and $0 \le i < n(k)$, and a is a fixed primitive element modulo p^e. For example, when $m = 9$ we can let $a = 2$; the values are $\{2^0 3^0, 2^1 3^0, 2^0 3^1, 2^2 3^0, 2^5 3^0, 2^1 3^1, 2^4 3^0, 2^3 3^0, 2^0 3^2\}$. Then $f(a^i p^k) = \sum_{0 \le l \le e} \sum_{0 \le j < n(l)} \omega^{g(i,j,k,l)} F(a^j p^l)$ where $g(i,j,k,l) = a^{i+j} p^{k+l}$.

We shall compute $f_{ikl} = \sum_{0 \le j < n(l)} \omega^{g(i,j,k,l)} F(a^j p^l)$ for $0 \le i < n(k)$ and for each k, l. This is a cyclic convolution of degree $n(k+l)$ on the values $x_i = \omega^{a^i p^{k+l}}$ and $y_s = \sum_{0 \le j < n(l),\, s+j \equiv 0 \,(\text{modulo } n(k+l))} F(a^j p^l)$, since $f_{ikl} = \sum x_r y_s$ summed over $r+s \equiv i \,(\text{modulo } n(k+l))$. The Fourier transform is obtained by summing appropriate f_{ikl}'s. [*Note:* When linear combinations of the x_i are formed, e.g., as in (67), the result will be purely real or purely imaginary, when the cyclic convolution algorithm has been constructed by using rule (57) with $u^{n(k)} - 1 = (u^{n(k)/2} - 1)(u^{n(k)/2} + 1)$. The reason is that reduction mod $(u^{n(k)/2} - 1)$ produces a polynomial with real coefficients $\omega^j + \omega^{-j}$ while reduction mod $(u^{n(k)/2} + 1)$ produces a polynomial with imaginary coefficients $\omega^j - \omega^{-j}$.]

When $p = 2$ an analogous construction applies, using the representation $(-1)^i a^j 2^k$ (modulo m), where $0 \le k \le e$ and $0 \le i \le \min(e-k, 1)$ and $0 \le j < 2^{e-k-2}$. In this case we use the construction of exercise 52 with $n' = 2$ and $n'' = 2^{e-k-2}$; although these numbers are not relatively prime, the construction does yield the desired direct product of cyclic convolutions.

(b) Let $a'm' + a''m'' = 1$; and let $\omega' = \omega^{a''m''}$, $\omega'' = \omega^{a'm'}$. Define $s' = s \bmod m'$, $s'' = s \bmod m''$, $t' = t \bmod m'$, $t'' = t \bmod m''$, so that $\omega^{st} = (\omega')^{s't'} (\omega'')^{s''t''}$. It follows that $f(s', s'') = \sum_{0 \le t' < m',\, 0 \le t'' < m''} (\omega')^{s't'} (\omega'')^{s''t''} F(t', t'')$; in other words, the one-dimensional Fourier transform on m elements is actually a two-dimensional Fourier transform on $m' \times m''$ elements, in slight disguise.

We shall deal with "normal" algorithms consisting of (i) a number of sums s_i of the F's and s's; followed by (ii) a number of products m_j, each of which is obtained by multiplying one of the F's or S's by a real or imaginary number α_j; followed by (iii) a number of further sums t_k, each of which is formed from m's or t's (not F's or s's). The final values must be m's or t's. For example, the "normal" Fourier transform scheme for $m = 5$ constructed from (67) and the method of part (a) is as follows: $s_1 = F(1) + F(4)$, $s_2 = F(3) + F(2)$, $s_3 = s_1 + s_2$, $s_4 = s_1 - s_2$, $s_5 = F(1) - F(4)$, $s_6 = F(2) - F(3)$, $s_7 = s_5 - s_6$; $m_1 = \frac{1}{4}(\omega + \omega^2 + \omega^4 + \omega^3)s_3$, $m_2 = \frac{1}{4}(\omega - \omega^2 + \omega^4 - \omega^3)s_4$, $m_3 = \frac{1}{2}(\omega + \omega^2 - \omega^4 - \omega^3)s_5$, $m_4 = \frac{1}{2}(-\omega + \omega^2 + \omega^4 - \omega^3)s_6$, $m_5 = \frac{1}{2}(\omega^3 - \omega^2)s_7$, $m_6 = 1 \cdot F(5)$, $m_7 = 1 \cdot s_3$; $t_0 = m_1 + m_6$, $t_1 = t_0 + m_2$, $t_2 = m_3 + m_5$, $t_3 = t_0 - m_2$, $t_4 = m_4 - m_5$, $t_5 = t_1 + t_2$, $t_6 = t_3 + t_4$, $t_7 = t_1 - t_2$, $t_8 = t_3 - t_4$, $t_9 = m_6 + m_7$. Note the multiplication by 1 shown in m_6 and m_7; this is required by our conventions, and it is important to include such cases for use in recursive constructions (although the multiplications need not really be done). Here $m_6 = f_{001}$, $m_7 = f_{010}$, $t_5 = f_{000} + f_{001} = f(2^0)$, $t_6 = f_{100} + f_{101} = f(2^1)$, etc. We can improve the scheme by introducing $s_8 = s_3 + F(5)$, replacing m_1 by $(\frac{1}{4}(\omega + \omega^2 + \omega^4 + \omega^3) - 1)s_3$ [this is $-\frac{5}{4}s_3$], replacing m_6 by $1 \cdot s_8$, and deleting m_7 and t_9; this saves one of the trivial multiplications by 1, and it will be advantageous when the scheme is used to build larger ones. In the improved scheme, $f(5) = m_6$, $f(1) = t_5$, $f(2) = t_6$, $f(3) = t_8$, $f(4) = t_7$.

Now suppose we have normal one-dimensional schemes for m' and m'', using respectively (a', a'') complex additions, (t', t'') trivial multiplications by ± 1 or $\pm i$, and a total of (c', c'') complex multiplications including the trivial ones. (The nontrivial

complex multiplications are all "simple" since they involve only two real multiplications and no real additions.) We can construct a normal scheme for the two-dimensional $m' \times m''$ case by applying the m' scheme to vectors $F(t', *)$ of length m''. Each s_i step becomes m'' additions; each m_j becomes a Fourier transform on m'' elements, but with all of the α's in this algorithm multiplied by α_j; and each t_k becomes m'' additions. Thus the new algorithm has $(a'm'' + c'a'')$ complex additions, $t't''$ trivial multiplications, and a total of $c'c''$ complex multiplications.

Using these techniques, Winograd has found normal one-dimensional schemes for the following small values of m with the following costs (a, t, c):

$m = 2$	$(2, 2, 2)$	$m = 7$	$(36, 1, 9)$
$m = 3$	$(6, 1, 3)$	$m = 8$	$(26, 6, 8)$
$m = 4$	$(8, 4, 4)$	$m = 9$	$(46, 1, 12)$
$m = 5$	$(17, 1, 6)$	$m = 16$	$(74, 8, 18)$

By combining these schemes as described above, we obtain methods that use fewer arithmetic operations than the "fast Fourier transform" (FFT) discussed in exercise 14. For example, when $m = 1008 = 7 \cdot 9 \cdot 16$, the costs come to $(17946, 8, 1944)$, so we can do a Fourier transform on 1008 complex numbers with 3872 real multiplications and 35892 real additions. It is possible to improve on Winograd's method for combining relatively prime moduli by using multidimensional convolutions, as shown by Nussbaumer and Quandalle in *IBM J. Res. and Devel.* **22** (1978), 134–144; their ingenious approach reduces the amount of computation needed for 1008-point complex Fourier transforms to 3084 real multiplications and 34668 real additions. By contrast, the FFT on 1024 complex numbers involves 14344 real multiplications and 27652 real additions. If the two-passes-at-once improvement in the answer to exercise 14 is used, however, the FFT on 1024 complex numbers needs only 10936 real multiplications and 25948 additions, and it is not difficult to implement. Therefore the subtler methods are faster only on machines that take significantly longer to multiply than to add.

[References: *Proc. Nat. Acad. Sci. USA* **73** (1976), 1005–1006; ***Math. Comp.* 32 (1978), 175–199; *Advances in Math.* 32 (1979), 83–117.**]

54. $\max(2e_1\deg(p_1) - 1, \ldots, 2e_q\deg(p_q) - 1, q + 1)$.

55. $2n' - q'$, where n' is the degree of the minimum polynomial of P (i.e., the monic polynomial μ of least degree such that $\mu(P)$ is the zero matrix) and r' is the number of distinct irreducible factors it has. (Reduce P by similarity transformations.)

56. Let $t_{ijk} + t_{jik} = \tau_{ijk} + \tau_{jik}$, for all i, j, k. If A, B, C is a realization of (t_{ijk}) of rank r, then $\sum_{1 \le l \le r} c_{kl} (\sum a_{il} x_i)(\sum b_{jl} x_j) = \sum_{i,j} t_{ijk} x_i x_j = \sum_{i,j} \tau_{ijk} x_i x_j$ for all k. Conversely, let the lth chain multiplication of a polynomial chain, for $1 \le l \le r$, be the product $(\alpha_l + \sum \alpha_{il} x_i)(\beta_l + \sum \beta_{jl} x_j)$, where α_l and β_l denote possible constant terms and/or nonlinear terms. All terms of degree 2 appearing at any step of the chain can be expressed as a linear combination $\sum_{1 \le l \le r} c_l (\sum a_{il} x_i)(\sum b_{jl} x_j)$; hence the chain defines a tensor (t_{ijk}) of rank $\le r$ such that $t_{ijk} + t_{jik} = \tau_{ijk} + \tau_{jik}$. This establishes the hint. Now $\operatorname{rank}(\tau_{ijk} + \tau_{jik}) = \operatorname{rank}(t_{ijk} + t_{jik}) \le \operatorname{rank}(t_{ijk}) + \operatorname{rank}(t_{jik}) = 2 \operatorname{rank}(t_{ijk})$.

A bilinear form in $x_1, \ldots, x_m, y_1, \ldots, y_n$ is a quadratic form in $m + n$ variables, where $\tau_{ijk} = t_{i,j-m,k}$ for $i \le m$ and $j > m$, otherwise $\tau_{ijk} = 0$. Now $\operatorname{rank}(\tau_{ijk}) + \operatorname{rank}(\tau_{jik}) \ge \operatorname{rank}(t_{ijk})$, since we obtain a realization of (t_{ijk}) by suppressing the last n rows of A and the first m rows of B in a realization A, B, C of $(\tau_{ijk} + \tau_{jik})$.

57. Let N be the smallest power of 2 that exceeds $2n$, and let $u_{n+1} = \cdots = u_{N-1} = v_{n+1} = \cdots = v_{N-1} = 0$. If $U_s = \sum_{0\le t<N} \omega^{st} u_t$, $V_s = \sum_{0\le t<N} \omega^{st} v_t$, $0 \le s < N$, $\omega = e^{2\pi i/N}$, then $\sum_{0\le s<N} \omega^{-st} U_s V_s = N \sum u_{t_1} v_{t_2}$, where the latter sum is taken over all t_1 and t_2 with $0 \le t_1, t_2 < N$, $t_1 + t_2 \equiv t$ (modulo N). The terms vanish unless $t_1 \le n$ and $t_2 \le n$, so $t_1 + t_2 < N$; thus the sum is the coefficient of z^t in the product $u(z)v(z)$. If we use the method of exercise 14 to compute the Fourier transforms and the inverse transforms, the number of complex operations is $O(N \log N) + O(N \log N) + O(N) + O(N \log N)$; and $N < 4n$. [Cf. Section 4.3.3 and the paper by J. M. Pollard, *Math. Comp.* **25** (1971), 365–374.]

When multiplying integer polynomials, it is possible to use an *integer* number ω that is of order 2^t modulo a prime p, and to determine the results modulo sufficiently many primes. Useful primes in this regard, together with their least primitive roots r (from which we take $\omega = r^{(p-1)/2^t} \bmod p$ when $p \bmod 2^t = 1$), can be found as described in Section 4.5.4. For $t = 9$, the ten largest cases $< 2^{35}$ are $p = 2^{35} - 512a + 1$, where $(a, r) = (28, 7), (31, 10), (34, 13), (56, 3), (58, 10), (76, 5), (80, 3), (85, 11), (91, 5), (101, 3)$; the ten largest cases $< 2^{31}$ are $p = 2^{31} - 512a + 1$, where $(a, r) = (1, 10), (11, 3), (19, 11), (20, 3), (29, 3), (35, 3), (55, 19), (65, 6), (95, 3), (121, 10)$. For larger t, all primes p of the form $2^t q + 1$ where $q < 32$ is odd and $2^{24} < p < 2^{36}$ are given by $(p - 1, r) = (11 \cdot 2^{21}, 3), (25 \cdot 2^{20}, 3), (27 \cdot 2^{20}, 5), (25 \cdot 2^{22}, 3), (27 \cdot 2^{22}, 7), (5 \cdot 2^{25}, 3), (7 \cdot 2^{26}, 3), (27 \cdot 2^{26}, 13), (15 \cdot 2^{27}, 31), (17 \cdot 2^{27}, 3), (3 \cdot 2^{30}, 5), (13 \cdot 2^{28}, 3), (29 \cdot 2^{27}, 3), (23 \cdot 2^{29}, 5)$. Some of the latter primes can be used with $\omega = 2^e$ for appropriate small e. For a discussion of such primes, see R. M. Robinson, *Proc. Amer. Math. Soc.* **9** (1958), 673–681; S. W. Golomb, *Math. Comp.* **30** (1976), 657–663.

However, the method of exercise 59 will almost always be preferable in practice.

58. (a) In general if A, B, C realizes (t_{ijk}), then $(x_1, \ldots, x_m)A$, B, C is a realization of the $1 \times n \times s$ matrix whose entry in row j, column k is $\sum x_i t_{ijk}$. So there must be at least as many nonzero elements in $(x_1, \ldots, x_m)A$ as the rank of this matrix. In the case of the $m \times n \times (m + n - 1)$ tensor corresponding to polynomial multiplication of degree $m - 1$ by degree $n - 1$, the corresponding matrix has rank n whenever $(x_1, \ldots, x_m) \ne (0, \ldots, 0)$. A similar statement holds with $A \leftrightarrow B$ and $m \leftrightarrow n$. [In particular, if we work over the field of 2 elements, this says that the rows of A modulo 2 form a "linear code" of m vectors having distance at least n, whenever A, B, C is a realization consisting entirely of integers. This observation, due to R. W. Brockett and D. Dobkin [*Linear Algebra and its Applic.* **19** (1978), 207–235, Theorem 14], can be used to obtain nontrivial lower bounds on the rank over the integers. For example, M. R. Brown and D. Dobkin have used it to show that realizations of $n \times n$ polynomial multiplication over the integers must have $t \ge 3.52n$, for all sufficiently large n; see *IEEE Trans.* **C–29** (1980), 337–340.]

(b) $\begin{pmatrix} 1&0&0&0&0&1&1&1 \\ 0&1&0&0&1&1&0&1 \\ 0&0&1&1&0&0&1&1 \end{pmatrix}$, $\begin{pmatrix} 1&0&0&0&0&1&1&1 \\ 0&1&0&0&0&1&0&1 \\ 0&0&1&0&0&0&1&1 \\ 0&0&0&1&1&0&0&1 \end{pmatrix}$, $\begin{pmatrix} 1&0&0&0&0&0&0&0 \\ \bar{1}&\bar{1}&0&0&0&1&0&0 \\ \bar{1}&1&\bar{1}&0&0&0&1&0 \\ 1&0&0&\bar{1}&\bar{1}&\bar{1}&\bar{1}&1 \\ 0&0&1&0&1&0&0&0 \\ 0&0&0&1&0&0&0&0 \end{pmatrix}$.

59. [*IEEE Trans.* **ASSP–28** (1980), 205–215.] Note that cyclic convolution is polynomial multiplication mod $u^n - 1$, and negacyclic convolution is polynomial multiplication mod $u^n + 1$. Let us now change notation, replacing n by 2^n; we shall consider recursive algorithms for cyclic and negacyclic convolution $(z_0, \ldots, z_{2^n-1})$ of $(x_0, \ldots, x_{2^n-1})$ with

$(y_0, \ldots, y_{2^n-1})$. The algorithms are presented in unoptimized form, for brevity and ease in exposition; the reader who implements them will notice that many things can be streamlined.

C1. [Test for simple case.] If $n = 1$, set $z_0 \leftarrow x_0y_0 + x_1y_1$, $z_1 \leftarrow (x_0 + x_1)(y_0 + y_1) - z_0$, and terminate. Otherwise set $m \leftarrow 2^{n-1}$.

C2. [Remainderize.] For $0 \le k < m$, set $(x_k, x_{m+k}) \leftarrow (x_k + x_{m+k}, x_k - x_{m+k})$ and $(y_k, y_{m+k}) \leftarrow (y_k + y_{m+k}, y_k - y_{m+k})$. (Now we have $x(u) \bmod (u^m - 1) = x_0 + \cdots + x_{m-1}u^{m-1}$ and $x(u) \bmod (u^m + 1) = x_m + \cdots + x_{2m-1}$; we will compute $x(u)y(u) \bmod (u^m - 1)$ and $x(u)y(u) \bmod (u^m + 1)$, then we will combine the results by (57).)

C3. [Recurse.] Set $(z_0, \ldots, z_{m-1})$ to the cyclic convolution of $(x_0, \ldots, x_{m-1})$ with $(y_0, \ldots, y_{m-1})$. Also set $(z_m, \ldots, z_{2m-1})$ to the negacyclic convolution of $(x_m, \ldots, x_{2m-1})$ with $(y_m, \ldots, y_{2m-1})$.

C4. [Unremainderize.] For $0 \le k < m$, set $(z_k, z_{m+k}) \leftarrow \frac{1}{2}(z_k + z_{m+k}, z_k - z_{m+k})$. Now $(z_0, \ldots, z_{m-1})$ is the desired answer. ∎

N1. [Test for simple case.] If $n = 1$, set $t \leftarrow x_0(y_0 + y_1)$, $z_0 \leftarrow t - (x_0 + x_1)y_1$, $z_1 \leftarrow t + (x_1 - x_0)y_0$, and terminate. Otherwise set $m \leftarrow 2^{\lfloor n/2 \rfloor}$ and $r \leftarrow 2^{\lceil n/2 \rceil}$. (The following steps use 2^{n+1} auxiliary variables X_{ij} for $0 \le i < 2m$ and $0 \le j < r$, to represent $2m$ polynomials $X_i(w) = X_{i0} + X_{i1}w + \cdots + X_{i(r-1)}w^{r-1}$; similarly, there are 2^{n+1} auxiliary variables Y_{ij}.)

N2. [Initialize auxiliary polynomials.] Set $X_{ij} \leftarrow X_{(i+m)j} \leftarrow x_{mj+i}$ $Y_{ij} \leftarrow Y_{(i+m)j} \leftarrow y_{mj+i}$, for $0 \le i < m$ and $0 \le j < r$. (At this point we have $x(u) = X_0(u^m) + uX_1(u^m) + \cdots + u^{m-1}X_{m-1}(u^m)$, and a similar formula holds for $y(u)$. Our strategy will be to multiply these polynomials modulo $(u^{mr} + 1) = (u^n + 1)$, by operating modulo $(w^r + 1)$ on the polynomials $X(w)$ and $Y(w)$, finding their cyclic correlation of length $2m$ and thereby obtaining $x(u)y(u) = Z_0(u^m) + uZ_1(u^m) + \cdots + u^{2m-1}Z_{2m-1}(u^m)$.)

N3. [Transform.] (Now we will essentially do a fast Fourier transform on the polynomials $(X_0, \ldots, X_{m-1}, 0, \ldots, 0)$ and $(Y_0, \ldots, Y_{m-1}, 0, \ldots, 0)$, using $w^{r/m}$ as a $(2m)$th root of unity. This is efficient, because multiplication by a power of w is not really a multiplication at all.) For $j = \lfloor n/2 \rfloor - 1, \ldots, 1, 0$ (in this order), do the following for all m binary numbers $s + t = (s_{\lfloor n/2 \rfloor} \ldots s_{j+1}0 \ldots 0)_2 + (0 \ldots 0t_{j-1} \ldots t_0)_2$: Replace $(X_{s+t}(w), X_{s+t+2^j}(w))$ by the pair of polynomials $(X_{s+t}(w) + w^{(r/m)(s/2)}X_{s+t+2^j}(w), X_{s+t}(w) - w^{(r/m)(s/2)}X_{s+t+2^j}(w))$. (See Section 4.3.3 and Eq. 4.3.3–33. The operation $X_i(w) \leftarrow X_i(w) + w^kX_l(w)$ means, more precisely, that we set $X_{ij} \leftarrow X_{ij} + X_{l(j+k)}$ if $j + k < r$, otherwise $X_{ij} \leftarrow X_{ij} - X_{l(j+k-r)}$, for $0 \le j < r$.) Do the same transformation on the Y's.

N4. [Recurse.] For $0 \le i < 2m$, set $(Z_{i0}, \ldots, Z_{i(r-1)})$ to the negacyclic convolution of $(X_{i0}, \ldots, X_{i(r-1)})$ and $(Y_{i0}, \ldots, Y_{i(r-1)})$.

N5. [Untransform.] For $j = 0, 1, \ldots, \lfloor n/2 \rfloor$ (in this order), set $(Z_{s+t}(w), Z_{s+t+2^j}(w)) \leftarrow \frac{1}{2}(Z_{s+t}(w) + Z_{s+t+2^j}(w), w^{-(r/m)(s/2)}(Z_{s+t}(w) - Z_{s+t+2^j}(w)))$, for all m choices of s and t as in step N3.

N6. [Repack.] (Now we have accomplished the goal stated at the end of step N2, since it is easy to show that the transform of the Z's is the product of the transforms of the X's and the Y's.) Set $z_i \leftarrow Z_{i0} - Z_{(m+i)(r-1)}$ and $z_{mj+i} \leftarrow Z_{ij} + Z_{(m+i)(j-1)}$ for $0 < j < r$, for $0 \le i < m$. ∎

It is easy to verify that at most n extra bits of precision are needed for the intermediate variables in this calculation; i.e., if $|x_i| \le M$ for $0 \le i < 2^n$ at the beginning of the algorithm, then all of the x and X variables will be bounded by $2^n M$ throughout.

Algorithm N performs A_n addition-subtractions, D_n halvings, and M_n multiplications, where $A_1 = 5$, $D_1 = 0$, $M_1 = 3$; for $n > 1$ we have $A_n = \lfloor n/2 \rfloor 2^{n+2} + 2^{\lfloor n/2 \rfloor + 1} A_{\lfloor n/2 \rfloor} + (\lfloor n/2 \rfloor + 1)2^{n+1} + 2^n$, $D_n = 2^{\lfloor n/2 \rfloor + 1} D_{\lfloor n/2 \rfloor} + (\lfloor n/2 \rfloor + 1)2^{n+1}$, and $M_n = 2^{\lfloor n/2 \rfloor + 1} M_{\lfloor n/2 \rfloor}$. The solutions are $A_n = 11 \cdot 2^{n-1+\lceil \lg n \rceil} - 3 \cdot 2^n + 6 \cdot 2^n S_n$, $D_n = 4 \cdot 2^{n-1+\lceil \lg n \rceil} - 2 \cdot 2^n + 2 \cdot 2^n S_n$, $M_n = 3 \cdot 2^{n-1+\lceil \lg n \rceil}$; here S_n satisfies the recurrence $S_1 = 0$, $S_n = 2S_{\lceil n/2 \rceil} + \lfloor n/2 \rfloor$, and it is not difficult to prove the inequalities $\frac{1}{2} n \lceil \lg n \rceil \le S_n \le \frac{1}{2} n \lg n + n$. Algorithm C does approximately the same amount of work as Algorithm N.

It would be interesting to find a simpler way to carry out the additions and subtractions in step N3 (and the reverse operations in N5), perhaps analogous to Yates's method. The operation $X_i \leftarrow X_i + w^k X_l$ sketched above can be done with a procedure that generalizes the data-rotation algorithm of Fletcher and Silver in *CACM* **9** (1966), 326, but there might be a better way.

60. (a) In Σ_1, for example, we can group all terms having a common value of j and k into a single trilinear term; this gives ν^2 trilinear terms when $(j,k) \in E \times E$, plus ν^2 when $(j,k) \in E \times O$ and ν^2 when $(j,k) \in O \times E$. When $\tilde{j} = k$ we can also include $-x_{jj}\, y_{j\tilde{j}}\, z_{\tilde{j}j}$ in Σ_1, free of charge. [In the case $n = 10$, the method multiplies 10 by 10 matrices with 710 noncommutative multiplications; this is fewer than required by any other known method, although Winograd's scheme (35) uses only 600 when commutativity is allowed.]

(b) Here we simply let S be all the indices (i,j,k) of one problem, $\tilde{S}$ the indices of the other. [When $m = n = s = 10$, the result is quite surprising: We can multiply two separate 10×10 matrices with 1300 noncommutative multiplications, while no scheme is known that would multiply each of them with 650.]

(c) Corresponding to the left-hand side of the stated identity we get the terms

$$x_{i+\epsilon,j+\zeta}\, y_{j+\zeta,k+\eta}\, z_{k+\eta,i+\epsilon} + x_{j+\eta,k+\epsilon}\, y_{k+\epsilon,i+\zeta}\, z_{i+\zeta,j+\eta} + x_{k+\zeta,i+\eta}\, y_{i+\eta,j+\epsilon}\, z_{j+\epsilon,k+\zeta}$$

summed over $(i,j,k) \in S$ and $0 \le \epsilon, \zeta, \eta \le 1$, so we get all the trilinear terms of the form $x_{ij}\, y_{jk}\, z_{ki}$ except when $\lceil i/2 \rceil = \lceil j/2 \rceil = \lceil k/2 \rceil$; however, these missing terms can all be included in Σ_1, Σ_2, or Σ_3. The sum Σ_1 turns out to include terms of the form $x_{i+\epsilon,j+\zeta}\, y_{i+\eta,j+\epsilon}$ times some sum of z's, so it contributes $8\nu^2$ terms to the trilinear realization; and Σ_2, Σ_3 are similar. To verify that the aBC terms cancel out, note that they are $\sum (-1)^{\zeta+\eta} x_{i+\epsilon,j+\zeta}\, y_{k+\epsilon,i+\zeta}\, z_{j+\epsilon,k+\zeta}$, so $\eta = 1$ cancels with $\eta = 0$. [This technique leads to asymptotic improvements over Strassen's method whenever $\frac{1}{3} n^3 + 6n^2 - \frac{4}{3} n < n^{\lg 7}$, namely when $36 \le n \le 184$, and it was the first construction known to break the $\lg 7$ barrier. Reference: *SIAM J. Computing* **9** (1980), 321–342.]

61. (a) Replace $a_{ij}(u)$ by $u a_{il}(u)$. (b) Let $a_{il}(u) = a_{il\mu} u^\mu$, etc., in a polynomial realization of length $r = \operatorname{rank}_d(t_{ijk})$. Then $t_{ijk} = \sum_{\mu+\nu+\sigma=d} \sum_{1 \le l \le r} a_{il\mu} b_{jl\nu} c_{kl\sigma}$. [This result can be improved to $\operatorname{rank}(t_{ijk}) \le (2d+1) \operatorname{rank}_d(t_{ijk})$ in an infinite field, because the trilinear form $\sum_{\mu+\nu+\sigma} a_\mu b_\nu c_\sigma$ corresponds to multiplication of polynomials modulo u^{d+1}, as pointed out by Bini and Pan.] (c,d) This is clear from the realizations in exercise 48.

62. The rank is 3, by the method of proof in Theorem W with $P = \left(\begin{smallmatrix} 0 & 1 \\ 0 & 0 \end{smallmatrix}\right)$. The border rank cannot be 1, since we cannot have $a_1(u)b_1(u)c_1(u) \equiv a_1(u)b_2(u)c_2(u) \equiv u^d$ and $a_1(u)b_2(u)c_1(u) \equiv a_1(u)b_1(u)c_2(u) \equiv 0$ (modulo u^{d+1}). The border rank is 2 because of the realization $\left(\begin{smallmatrix} 1 & 1 \\ u & 0 \end{smallmatrix}\right)$, $\left(\begin{smallmatrix} u & 0 \\ 1 & 1 \end{smallmatrix}\right)$, $\left(\begin{smallmatrix} 1 & -1 \\ 0 & u \end{smallmatrix}\right)$.

63. (a) Let the elements of $T(m, n, s)$ and $T(M, N, S)$ be denoted by $t_{\langle i,j'\rangle\langle j,k'\rangle\langle k,i'\rangle}$ and $T_{\langle I,J'\rangle\langle J,K'\rangle\langle K,I'\rangle}$, respectively. Each element $\mathcal{T}_{\langle I,J'\rangle\langle J,K'\rangle\langle K,I'\rangle}$ of the direct product, where $I = \langle i, I\rangle$, $J = \langle j, J\rangle$, and $K = \langle k, K\rangle$, is equal to $t_{\langle i,j'\rangle\langle j,k'\rangle\langle k,i'\rangle}T_{\langle I,J'\rangle\langle J,K'\rangle\langle K,I'\rangle}$ by definition, so it is 1 iff $I' = I$ and $J' = J$ and $K' = K$.

(b) We have $M(mns) \le r^3$, since $T(mns, mns, mns) = T(m, n, s) \otimes T(n, s, m) \otimes T(s, m, n)$. If $M(P) \le R$ we have $M(P^h) \le R^h$ for all h, and it follows that $M(N) \le M(P^{\lceil \log_P N\rceil}) \le R^{\lceil \log_P N\rceil} \le RN^{\log R/\log P}$. [This result appears in Pan's 1972 paper.]

(c) We have $M_d(mns) \le r^3$ for some d, where $M_d(n) = \operatorname{rank}_d(T(n, n, n))$. If $M_d(P) \le R$ we have $M_{hd}(P^h) \le R^h$ for all h, and the stated formula follows since $M(P^h) \le \binom{hd+2}{2}R^h$ by exercise 61. In an infinite field we save a factor of $\log N$. [This result is due to Bini and Schönhage, 1979.]

(d) Let $P = mns$; we can perform p^{3h} independent $P^h \times P^h$ matrix multiplications with at most $\binom{hd+2}{2}p^{3h}r^{3h}$ noncommutative scalar multiplications. Reduce $M(P^{2h})$ to $M(P^h)$ matrix multiplications of size $P^h \times P^h$; thus we have $M(P^{2h}) \le \binom{hd+2}{2}r^{3h}M(P^h)(1 + O(p/r)^{3h})$. Iterating this recurrence gives

$$M(N) = O(N^{\beta(m,n,s,r)}) \cdot \exp(O(\log\log n)^2).$$

64. (a) Let $a = x_{ij}$, $A = uX_{ki}$, $b = y_{jk}$, $B = Y_{ij}$, $c = uz_{ki}$, $C = Z_{jk}$, so that $\Sigma_2 = O(u^2)$ can be eliminated. [When $m = s = 7$ and $n = 1$, this gives $M(N) = O(N^{2.66})$.] (b) Take $\alpha_1 = s - 1$, $\alpha_2 = -\alpha_1^{-2}$, $\alpha_3 = \alpha_4 = -1$, $\alpha_5 = 1$, $\alpha_6 = s - 1$, and $d \ge 4$. [We assume that α^{-1} exists in the field.] (c) Taking the direct product of $T(m, n, 2s) \oplus T(2n, s, m) \oplus T(s, 2m, n)$ with itself $3h$ times gives a tensor whose border rank is at most $\big(2(m+1)n(s+2)\big)^{3h}$. This tensor is the direct sum of 3^{3h} terms of the form $T(m^i(2n)^j s^k, n^i s^j (2m)^k, (2s)^i m^j n^k)$ where $i + j + k = 3h$, and $(3h)!/h!^3$ of these have $i = j = k = h$. Thus if we let $P = (2mns)^h$ and $p = (3h)!/h!^3$, the border rank of $pT(P, P, P)$ is at most pr, where

$$r = \big(2(m + 1)n(s + 2)\big)^{3h}/p.$$

Exercise 63(d) now implies that $M(N) = O(N^{\beta(P,P,P,r)+\epsilon})$ for all $\epsilon > 0$; here P and r are functions of h. We complete the proof by letting h be large in $\beta(P, P, P, r) = \log r/\log P = (3h\log 2(m + 1)n(s + 2) - 3h\log 3 + O(\log h))/(3h\log 2mns)$, which equals $\beta(m, n, 2s, \frac{2}{3}(m + 1)n(s + 2)) + O((\log h)/h)$. [The best value is obtained for $m = 5$, $n = 1$, $s = 11$, $\beta = 3\log_{110} 52 < 2.522$.]

SECTION 4.7

1. Find the first nonzero coefficient V_m, as in (4), and divide both $U(z)$ and $V(z)$ by z^m (shifting the coefficients m places to the left). The quotient will be a power series iff $U_0 = \cdots = U_{m-1} = 0$.

2. We have $V_0^{n+1}W_n = V_0^nU_n - (V_0^1W_0)(V_0^{n-1}V_n) - (V_0^2W_1)(V_0^{n-2}V_{n-1}) - \cdots - (V_0^nW_{n-1})(V_0^0V_1)$. Thus, we can start by replacing (U_j, V_j) by $(V_0^jU_j, V_0^{j-1}V_j)$ for $j \geq 1$, then set $W_n \leftarrow U_n - \sum_{0\leq k<n} W_kV_{n-k}$ for $n \geq 0$, finally replace W_j by W_j/V_0^{j+1} for $j \geq 0$. Similar techniques are possible in connection with other algorithms in this section.

3. Yes. When $\alpha = 0$, it is easy to prove by induction that $W_1 = W_2 = \cdots = 0$. When $\alpha = 1$, we find $W_n = V_n$, by the "cute" identity

$$\sum_{1\leq k\leq n} \left(\frac{k-(n-k)}{n}\right) V_kV_{n-k} = V_0V_n.$$

4. If $W(z) = e^{V(z)}$, then $W'(z) = V'(z)W(z)$; we find $W_0 = 1$, and

$$W_n = \sum_{1\leq k\leq n} \frac{k}{n} V_kW_{n-k}, \qquad \text{for } n \geq 1.$$

If $W(z) = \ln(1 + V(z))$, then $W'(z) + W'(z)V(z) = V'(z)$; the rule is $W_0 = 0$, and $W_1 + 2W_2z + \cdots = V'(z)/(1 + V(z))$.

[By exercise 6, the logarithm can be obtained to order n in $O(n \log n)$ operations. R. P. Brent observes that $\exp(V(z))$ can also be calculated with this asymptotic speed by applying Newton's method to $f(x) = \ln x - V(z)$; therefore general exponentiation $(1+V(z))^\alpha = \exp(\alpha \ln(1+V(z)))$ is $O(n \log n)$ too. Reference: *Analytic Computational Complexity*, ed. by J. F. Traub (New York: Academic Press, 1975), 172–176.]

5. We get the original series back again. This can be used to test a reversion algorithm.

6. $\varphi(x) = x + x(1 - xV(z))$, cf. Algorithm 4.3.3R. Thus after $W_0, \ldots, W_{N-1}$ are known, the idea is to input $V_N, \ldots, V_{2N-1}$, compute $(W_0 + \cdots + W_{N-1}z^{N-1}) \times (V_0 + \cdots + V_{2N-1}z^{2N-1}) = 1 + R_0z^N + \cdots + R_{N-1}z^{2N-1} + O(z^{2N})$, and let $W_N + \cdots + W_{2N-1}z^{N-1} = -(W_0 + \cdots + W_{N-1}z^{N-1})(R_0 + \cdots + R_{N-1}z^{N-1}) + O(z^N)$. [*Numer. Math.* **22** (1974), 341–348; this algorithm was, in essence, first published by M. Sieveking, *Computing* **10** (1972), 153–156.] Note that the total time for N coefficients is $O(N \log N)$ if we use "fast" polynomial multiplication (exercise 4.6.4–57).

7. $W_n = \binom{mk}{k}/n$ when $n = (m-1)k + 1$, otherwise 0. (Cf. exercise 2.3.4.4–11.)

8. Input G_1 in step L1, and G_n in step L2. In step L4, the output should be $(U_{n-1}G_1 + 2U_{n-2}G_2 + \cdots + nU_0G_n)/n$. (The running time of the order N^3 algorithm is hereby increased by only order N^2. The value $W_1 = G_1$ might be output in step L1.)

Note: Algorithms T and N determine $V^{-1}(U(z))$; the algorithm in this exercise determines $G(V^{-1}(z))$, which is somewhat different. Of course, the results can all be obtained by a sequence of operations of reversion and composition (exercise 11), but it is helpful to have more direct algorithms for each case.

9.

	$n=1$	$n=2$	$n=3$	$n=4$	$n=5$
T_{1n}	1	1	2	5	14
T_{2n}		1	2	5	14
T_{3n}			1	3	9
T_{4n}				1	4
T_{5n}					1

10. Form $y^{1/\alpha} = x(1 + a_1x + a_2x^2 + \cdots)^{1/\alpha} = x(1 + c_1x + c_2x^2 + \cdots)$ by means of Eq. (9); then revert the latter series. (See the remarks following Eq. 1.2.11.3–11.)

11. Set $W_0 \leftarrow U_0$, and set $(T_k, W_k) \leftarrow (V_k, 0)$ for $1 \le k \le N$. Then for $n = 1$, 2, ..., N, do the following: Set $W_j \leftarrow W_j + U_nT_j$ for $n \le j \le N$; and then set $T_j \leftarrow T_{j-1}V_1 + \cdots + T_nV_{j-n}$ for $j = N, N-1, \ldots, n+1$.

Here $T(z)$ represents $V(z)^n$. An *on-line* power series algorithm for this problem, analogous to Algorithm T, could be constructed, but it would require about $N^2/2$ storage locations. There is also an on-line algorithm that solves this exercise and needs only $O(N)$ storage locations: We may assume that $V_1 = 1$, if U_k is replaced by $U_kV_1^k$ and V_k is replaced by V_k/V_1 for all k. Then we may revert $V(z)$ by Algorithm L, using its output as input to the algorithm of exercise 8 with $G_1 = U_1$, $G_2 = U_2$, etc., thus computing $U((V^{-1})^{-1}(z)) - U_0$.

Brent and Kung have constructed several algorithms that are asymptotically faster. For example, we can evaluate $U(x)$ for $x = V(z)$ by a slight variant of exercise 4.6.4–42(c), doing about $2\sqrt{n}$ chain multiplications of cost $M(n)$ and about n parameter multiplications of cost n, where $M(n)$ is the number of operations needed to multiply power series to order n; the total time is therefore $O(\sqrt{n}M(n) + n^2) = O(n^2)$. A still faster method can be based on the identity $U(V_0(z) + z^mV_1(z)) = U(V_0(z)) + z^mU'(V_0(z))V_1(z) + z^{2m}U''(V_0(z))V_1(z)^2/2! + \cdots$, extending to about n/m terms, where we choose $m \approx \sqrt{n/\log n}$; the first term $U(V_0(z))$ is evaluated in $O(mn(\log n)^2)$ operations using a method somewhat like that in exercise 4.6.4–43. Since we can go from $U^{(k)}(V_0(z))$ to $U^{(k+1)}(V_0(z))$ in $O(n \log n)$ operations by differentiating and dividing by $V_0'(z)$, the entire procedure takes $O(mn(\log n)^2 + (n/m)n \log n) = O(n \log n)^{3/2}$ operations. [*JACM* **25** (1978), 581–595.]

12. Polynomial division is trivial unless $m \ge n \ge 1$. Assuming the latter, the equation $u(x) = q(x)v(x) + r(x)$ is equivalent to $U(z) = Q(z)V(z) + z^{m-n+1}R(z)$ where $U(x) = x^mu(x^{-1})$, $V(x) = x^nv(x^{-1})$, $Q(x) = x^{m-n}q(x^{-1})$, and $R(x) = x^{n-1}r(x^{-1})$ are the "reverse" polynomials of u, v, q, and r.

To find $q(x)$ and $r(x)$, compute the first $m - n + 1$ coefficients of the power series $U(z)/V(z) = W(z) + O(z^{m-n+1})$; then compute the power series $U(z) - V(z)W(z)$, which has the form $z^{m-n+1}T(z)$ where $T(z) = T_0 + T_1z + \cdots$. Note that $T_j = 0$ for all $j \ge n$; hence $Q(z) = W(z)$ and $R(z) = T(z)$ satisfy the requirements.

13. Apply exercise 4.6.1–3 with $u(z) = z^N$ and $v(z) = W_0 + \cdots + W_{N-1}z^{N-1}$; the desired approximations are the values of $v_3(z)/v_2(z)$ obtained during the course of the algorithm. Exercise 4.6.1–26 tells us that there are no further possibilities with relatively prime numerator and denominator. If each W_i is an integer, an all-integer extension of Algorithm 4.6.1C will have the desired properties.

Notes: The case $N = 2n+1$ and $\deg(w_1) = \deg(w_2) = n$ is of particular interest, since it is equivalent to a so-called Toeplitz system. The method of this exercise can be generalized to arbitrary rational interpolation of the form $W(z) \equiv p(z)/q(z)$ (modulo $(z - z_1)\ldots(z - z_N)$), where the z_i's need not be distinct; thus, we can specify the value of $W(z)$ and some of its derivatives at several points. See Fred G. Gustavson and David Y. Y. Yun, IBM Research Report RC7551 (1979).

14. If $U(z) = z + U_kz^k + \cdots$ and $V(z) = z^k + V_{k+1}z^{k+1} + \cdots$, we find that the difference $V(U(z)) - U'(z)V(z)$ is $\sum_{j\ge1} z^{2k+j-1}j(U_kV_{k+j} - U_{k+j} +$ (polynomial involving only $U_k, \ldots, U_{k+j-1}, V_{k+1}, \ldots, V_{k+j-1}$)); hence $V(z)$ is unique if $U(z)$ is given and $U(z)$ is unique if $V(z)$ and U_k are given.

The solution depends on two auxiliary algorithms, the first of which solves the equation $V(z+z^kU(z)) = (1+z^{k-1}W(z))V(z)+z^{k-1}S(z)+O(z^{k-1+n})$ for $V(z) = V_0 + V_1z + \cdots + V_{n-1}z^{n-1}$, given $U(z)$, $W(z)$, $S(z)$, and n. If $n = 1$, let $V_0 = -S(0)/W(0)$; or let $V_0 = 1$ when $S(0) = W(0) = 0$. To go from n to $2n$, let

$$V(z+z^kU(z)) = (1+z^{k-1}W(z))V(z)+z^{k-1}S(z)-z^{k-1+n}R(z)+O(z^{k-1+2n}),$$
$$1+z^{k-1}\hat{W}(z) = (z/(z+z^kU(z)))^n(1+z^{k-1}W(z)),$$
$$\hat{S}(z) = (z/(z+z^kU(z)))^nR(z),$$

and let $\hat{V}(z) = V_n + V_{n+1}z + \cdots + V_{2n-1}z^{n-1}$ satisfy

$$\hat{V}(z+z^kU(z)) = (1+z^{k-1}\hat{W}(z))\hat{V}(z)+z^{k-1}\hat{S}(z)+O(z^{k-1+n}).$$

The second algorithm solves $W(z)U(z)+zU'(z) = V(z)+O(z^n)$ for $U(z) = U_0+U_1z+\cdots+U_{n-1}z^{n-1}$, given $V(z)$, $W(z)$, and n. If $n = 1$, let $U_0 = V(0)/W(0)$, or let $U_0 = 1$ in case $V(0) = W(0) = 0$. To go from n to $2n$, let $W(z)U(z)+zU'(z) = V(z)-z^nR(z)+O(z^{2n})$, and let $\hat{U}(z) = U_n+\cdots+U_{2n-1}z^{n-1}$ be the solution to the equation $(n+W(z))\hat{U}(z)+z\hat{U}'(z) = R(z)+O(z^n)$.

Resuming the notation of (27), the first algorithm can be used to solve $\hat{V}(U(z)) = U'(z)(z/U(z))^k\hat{V}(z)$ to any desired accuracy, and we set $V(z) = z^k\hat{V}(z)$. To find $P(z)$, suppose we have $V(P(z)) = P'(z)V(z)+O(z^{2k-1+n})$, an equation that holds for $n = 1$ when $P(z) = z+\alpha z^k$ and α is arbitrary. We can go from n to $2n$ by letting $V(P(z)) = P'(z)V(z)+z^{2k-1+n}R(z)+O(z^{2k-1+2n})$ and replacing $P(z)$ by $P(z)+z^{k+n}\hat{P}(z)$, where the second algorithm is used to find the polynomial $\hat{P}(z)$ such that $(k+n-zV'(z)/V(z))\hat{P}(z)+z\hat{P}'(z) = (z^k/V(z))R(z)+O(z^n)$.

APPENDIX A

TABLES OF NUMERICAL QUANTITIES

Table 1

QUANTITIES THAT ARE FREQUENTLY USED IN STANDARD SUBROUTINES AND IN ANALYSIS OF COMPUTER PROGRAMS (40 DECIMAL PLACES)

$\sqrt{2} = 1.41421\ 35623\ 73095\ 04880\ 16887\ 24209\ 69807\ 85697-$
$\sqrt{3} = 1.73205\ 08075\ 68877\ 29352\ 74463\ 41505\ 87236\ 69428+$
$\sqrt{5} = 2.23606\ 79774\ 99789\ 69640\ 91736\ 68731\ 17623\ 54406+$
$\sqrt{10} = 3.16227\ 76601\ 68379\ 33199\ 88935\ 44432\ 71853\ 37196-$
$\sqrt[3]{2} = 1.25992\ 10498\ 94873\ 16476\ 72106\ 07278\ 22835\ 05703-$
$\sqrt[3]{3} = 1.44224\ 95703\ 07408\ 38232\ 16383\ 10780\ 10958\ 83919-$
$\sqrt[4]{2} = 1.18920\ 71150\ 02721\ 06671\ 74999\ 70560\ 47591\ 52930-$
$\ln 2 = 0.69314\ 71805\ 59945\ 30941\ 72321\ 21458\ 17656\ 80755+$
$\ln 3 = 1.09861\ 22886\ 68109\ 69139\ 52452\ 36922\ 52570\ 46475-$
$\ln 10 = 2.30258\ 50929\ 94045\ 68401\ 79914\ 54684\ 36420\ 76011+$
$1/\ln 2 = 1.44269\ 50408\ 88963\ 40735\ 99246\ 81001\ 89213\ 74266+$
$1/\ln 10 = 0.43429\ 44819\ 03251\ 82765\ 11289\ 18916\ 60508\ 22944-$
$\pi = 3.14159\ 26535\ 89793\ 23846\ 26433\ 83279\ 50288\ 41972-$
$1^\circ = \pi/180 = 0.01745\ 32925\ 19943\ 29576\ 92369\ 07684\ 88612\ 71344+$
$1/\pi = 0.31830\ 98861\ 83790\ 67153\ 77675\ 26745\ 02872\ 40689+$
$\pi^2 = 9.86960\ 44010\ 89358\ 61883\ 44909\ 99876\ 15113\ 53137-$
$\sqrt{\pi} = \Gamma(1/2) = 1.77245\ 38509\ 05516\ 02729\ 81674\ 83341\ 14518\ 27975+$
$\Gamma(1/3) = 2.67893\ 85347\ 07747\ 63365\ 56929\ 40974\ 67764\ 41287-$
$\Gamma(2/3) = 1.35411\ 79394\ 26400\ 41694\ 52880\ 28154\ 51378\ 55193+$
$e = 2.71828\ 18284\ 59045\ 23536\ 02874\ 71352\ 66249\ 77572+$
$1/e = 0.36787\ 94411\ 71442\ 32159\ 55237\ 70161\ 46086\ 74458+$
$e^2 = 7.38905\ 60989\ 30650\ 22723\ 04274\ 60575\ 00781\ 31803+$
$\gamma = 0.57721\ 56649\ 01532\ 86060\ 65120\ 90082\ 40243\ 10422-$
$\ln \pi = 1.14472\ 98858\ 49400\ 17414\ 34273\ 51353\ 05871\ 16473-$
$\phi = 1.61803\ 39887\ 49894\ 84820\ 45868\ 34365\ 63811\ 77203+$
$e^\gamma = 1.78107\ 24179\ 90197\ 98523\ 65041\ 03107\ 17954\ 91696+$
$e^{\pi/4} = 2.19328\ 00507\ 38015\ 45655\ 97696\ 59278\ 73822\ 34616+$
$\sin 1 = 0.84147\ 09848\ 07896\ 50665\ 25023\ 21630\ 29899\ 96226-$
$\cos 1 = 0.54030\ 23058\ 68139\ 71740\ 09366\ 07442\ 97660\ 37323+$
$-\zeta'(2) = 0.93754\ 82543\ 15843\ 75370\ 25740\ 94567\ 86497\ 78979-$
$\zeta(3) = 1.20205\ 69031\ 59594\ 28539\ 97381\ 61511\ 44999\ 07650-$
$\ln \phi = 0.48121\ 18250\ 59603\ 44749\ 77589\ 13424\ 36842\ 31352-$
$1/\ln \phi = 2.07808\ 69212\ 35027\ 53760\ 13226\ 06117\ 79576\ 77422-$
$-\ln \ln 2 = 0.36651\ 29205\ 81664\ 32701\ 24391\ 58232\ 66946\ 94543-$

Table 2

QUANTITIES THAT ARE FREQUENTLY USED IN STANDARD SUBROUTINES AND IN ANALYSIS OF COMPUTER PROGRAMS (44 OCTAL PLACES)

The names at the left of the equal signs are given in decimal notation.

$0.1 =$	*0.06314 63146 31463 14631 46314 63146 31463 14631 4632*
$0.01 =$	*0.00507 53412 17270 24365 60507 53412 17270 24365 6051*
$0.001 =$	*0.00040 61115 64570 65176 76355 44264 16254 02030 4467*
$0.0001 =$	*0.00003 21556 13530 70414 54512 75170 33021 15002 3522*
$0.00001 =$	*0.00000 24761 32610 70664 36041 06077 17401 56063 3442*
$0.000001 =$	*0.00000 02061 57364 05536 66151 55323 07746 44470 2603*
$0.0000001 =$	*0.00000 00153 27745 15274 53644 12741 72312 20354 0215*
$0.00000001 =$	*0.00000 00012 57143 56106 04303 47374 77341 01512 6333*
$0.000000001 =$	*0.00000 00001 04560 27640 46655 12262 71426 40124 2174*
$0.0000000001 =$	*0.00000 00000 06676 33766 35367 55653 37265 34642 0163*
$\sqrt{2} =$	*1.32404 74631 77167 46220 42627 66115 46725 12575 1744*
$\sqrt{3} =$	*1.56663 65641 30231 25163 54453 50265 60361 34073 4222*
$\sqrt{5} =$	*2.17067 36334 57722 47602 57471 63003 00563 55620 3202*
$\sqrt{10} =$	*3.12305 40726 64555 22444 02242 57101 41466 33775 2253*
$\sqrt[3]{2} =$	*1.20505 05746 15345 05342 10756 65334 25574 22415 0303*
$\sqrt[3]{3} =$	*1.34233 50444 22175 73134 67363 76133 05334 31147 6012*
$\sqrt[4]{2} =$	*1.14067 74050 61556 12455 72152 64430 60271 02755 7314*
$\ln 2 =$	*0.54271 02775 75071 73632 57117 07316 30007 71366 5364*
$\ln 3 =$	*1.06237 24752 55006 05227 32440 63065 25012 35574 5534*
$\ln 10 =$	*2.23273 06735 52524 25405 56512 66542 56026 46050 5071*
$1/\ln 2 =$	*1.34252 16624 53405 77027 35750 37766 40644 35175 0435*
$1/\ln 10 =$	*0.33626 75425 11562 41614 52325 33525 27655 14756 0622*
$\pi =$	*3.11037 55242 10264 30215 14230 63050 56006 70163 2112*
$1^\circ = \pi/180 =$	*0.01073 72152 11224 72344 25603 54276 63351 22056 1155*
$1/\pi =$	*0.24276 30155 62344 20251 23760 47257 50765 15156 7007*
$\pi^2 =$	*11.67517 14467 62135 71322 25561 15466 30021 40654 3410*
$\sqrt{\pi} = \Gamma(1/2) =$	*1.61337 61106 64736 65247 47035 40510 15273 34470 1776*
$\Gamma(1/3) =$	*2.53347 35234 51013 61316 73106 47644 54653 00106 6605*
$\Gamma(2/3) =$	*1.26523 57112 14154 74312 54572 37655 60126 23231 0245*
$e =$	*2.55760 52130 50535 51246 52773 42542 00471 72363 6166*
$1/e =$	*0.27426 53066 13167 46761 52726 75436 02440 52371 0336*
$e^2 =$	*7.30714 45615 23355 33460 63507 35040 32664 25356 5022*
$\gamma =$	*0.44742 14770 67666 06172 23215 74376 01002 51313 2552*
$\ln \pi =$	*1.11206 40443 47503 36413 65374 52661 52410 37511 4606*
$\phi =$	*1.47433 57156 27751 23701 27634 71401 40271 66710 1501*
$e^\gamma =$	*1.61772 13452 61152 65761 22477 36553 53327 17554 2126*
$e^{\pi/4} =$	*2.14275 31512 16162 52370 35530 11342 53525 44307 0217*
$\sin 1 =$	*0.65665 24436 04414 73402 03067 23644 11612 07474 1451*
$\cos 1 =$	*0.42450 50037 32406 42711 07022 14666 27320 70675 1232*
$-\zeta'(2) =$	*0.74001 45144 53253 42362 42107 23350 50074 46100 2771*
$\zeta(3) =$	*1.14735 00023 60014 20470 15613 42561 31715 10177 0662*
$\ln \phi =$	*0.36630 26256 61213 01145 13700 41004 52264 30700 4065*
$1/\ln \phi =$	*2.04776 60111 17144 41512 11436 16575 00355 43630 4065*
$-\ln \ln 2 =$	*0.27351 71233 67265 63650 17401 56637 26334 31455 5701*

Several of the values in Tables 1 and 2 were computed by John W. Wrench, Jr. For high-precision values of constants not found in this list, see J. Peters, *Ten Place Logarithms of the Numbers from 1 to 100000*, Appendix to Volume 1 (New York: F. Ungar Publ. Co., 1957); and *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun (Washington, D.C.: U.S. Gov't Printing Office, 1964), Chapter 1.

A table of Bernoulli numbers through B_{250} appears in a paper by D. E. Knuth and T. J. Buckholtz, *Math. Comp.* **21** (1967), 663–688.

Table 3

VALUES OF HARMONIC NUMBERS, BERNOULLI NUMBERS, AND FIBONACCI NUMBERS, FOR SMALL VALUES OF n

n	H_n	B_n	F_n	n
0	0	1	0	0
1	1	−1/2	1	1
2	3/2	1/6	1	2
3	11/6	0	2	3
4	25/12	−1/30	3	4
5	137/60	0	5	5
6	49/20	1/42	8	6
7	363/140	0	13	7
8	761/280	−1/30	21	8
9	7129/2520	0	34	9
10	7381/2520	5/66	55	10
11	83711/27720	0	89	11
12	86021/27720	−691/2730	144	12
13	1145993/360360	0	233	13
14	1171733/360360	7/6	377	14
15	1195757/360360	0	610	15
16	2436559/720720	−3617/510	987	16
17	42142223/12252240	0	1597	17
18	14274301/4084080	43867/798	2584	18
19	275295799/77597520	0	4181	19
20	55835135/15519504	−174611/330	6765	20
21	18858053/5173168	0	10946	21
22	19093197/5173168	854513/138	17711	22
23	444316699/118982864	0	28657	23
24	1347822955/356948592	−236364091/2730	46368	24
25	34052522467/8923714800	0	75025	25

For any x, let $H_x = \sum_{n\geq 1}\left(\frac{1}{n} - \frac{1}{n+x}\right)$. Then

$$\begin{aligned}
H_{1/2} &= 2 - 2\ln 2,\\
H_{1/3} &= 3 - \tfrac{1}{2}\pi/\sqrt{3} - \tfrac{3}{2}\ln 3,\\
H_{2/3} &= \tfrac{3}{2} + \tfrac{1}{2}\pi/\sqrt{3} - \tfrac{3}{2}\ln 3,\\
H_{1/4} &= 4 - \tfrac{1}{2}\pi - 3\ln 2,\\
H_{3/4} &= \tfrac{4}{3} + \tfrac{1}{2}\pi - 3\ln 2,\\
H_{1/5} &= 5 - \tfrac{1}{2}\pi\phi\sqrt{(2+\phi)/5} - \tfrac{1}{2}(3-\phi)\ln 5 - (\phi - \tfrac{1}{2})\ln(2+\phi),\\
H_{2/5} &= \tfrac{5}{2} - \tfrac{1}{2}\pi/\phi\sqrt{2+\phi} - \tfrac{1}{2}(2+\phi)\ln 5 + (\phi - \tfrac{1}{2})\ln(2+\phi),\\
H_{3/5} &= \tfrac{5}{3} + \tfrac{1}{2}\pi/\phi\sqrt{2+\phi} - \tfrac{1}{2}(2+\phi)\ln 5 + (\phi - \tfrac{1}{2})\ln(2+\phi),\\
H_{4/5} &= \tfrac{5}{4} + \tfrac{1}{2}\pi\phi\sqrt{(2+\phi)/5} - \tfrac{1}{2}(3-\phi)\ln 5 - (\phi - \tfrac{1}{2}\ln(2+\phi),\\
H_{1/6} &= 6 - \tfrac{1}{2}\pi\sqrt{3} - 2\ln 2 - \tfrac{3}{2}\ln 3,\\
H_{5/6} &= \tfrac{6}{5} + \tfrac{1}{2}\pi\sqrt{3} - 2\ln 2 - \tfrac{3}{2}\ln 3,
\end{aligned}$$

and, in general, when $0 < p < q$ (cf. exercise 1.2.9–19),

$$H_{p/q} = \frac{q}{p} - \frac{1}{2}\pi\cot\frac{p}{q}\pi - \ln 2q + 2\sum_{1\leq n<q/2}\cos\frac{2\pi np}{q}\ln\sin\frac{n}{q}\pi.$$

APPENDIX B

INDEX TO NOTATIONS

In the following formulas, letters that are not further qualified have the following significance:

j, k	integer-valued arithmetic expression
m, n	nonnegative integer-valued arithmetic expression
x, y, z	real-valued arithmetic expression
f	real-valued function

Formal symbolism	Meaning	Section reference
∎	end of algorithm, program, or proof	1.1
A_n	the nth element of linear array A	
A_{mn}	the element in row m and column n of rectangular array A	
$A[n]$	equivalent to A_n	1.1
$A[m, n]$	equivalent to A_{mn}	1.1
$V \leftarrow E$	give variable V the value of expression E	1.1
$U \leftrightarrow V$	interchange the values of variables U and V	1.1
$(B \Rightarrow E;\ E')$	conditional expression: denotes E if B is true, E' if B is false	8.1
δ_{kj}	Kronecker delta: $(j = k \Rightarrow 1;\ 0)$	1.2.6
$\sum_{R(k)} f(k)$	sum of all $f(k)$ such that the variable k is an integer and relation $R(k)$ is true	1.2.3
$\prod_{R(k)} f(k)$	product of all $f(k)$ such that the variable k is an integer and relation $R(k)$ is true	1.2.3
$\min_{R(k)} f(k)$	minimum value of all $f(k)$ such that the variable k is an integer and relation $R(k)$ is true	1.2.3
$\max_{R(k)} f(k)$	maximum value of all $f(k)$ such that the variable k is an integer and relation $R(k)$ is true	1.2.3

Formal symbolism	Meaning	Section reference
A^T	transpose of rectangular array A: $A^T[j,k] = A[k,j]$	1.2.3
x^y	x to the y power (when x is positive)	1.2.2
x^k	x to the kth power: $\left(k \geq 0 \Rightarrow \prod_{0 \leq j < k} x;\quad 1/x^{-k}\right)$	1.2.2
$x^{\overline{k}}$	x upper k: $\left(k \geq 0 \Rightarrow \prod_{0 \leq j < k} (x+j);\quad 1/(x+k)^{\overline{-k}}\right)$	1.2.6
$x^{\underline{k}}$	x lower k: $\left(k \geq 0 \Rightarrow \prod_{0 \leq j < k} (x-j);\quad 1/(x-k)^{\underline{-k}}\right)$	1.2.6
$n!$	n factorial: $n^{\underline{n}}$	1.2.5
$f'(x)$	derivative of f at x	1.2.9
$f''(x)$	second derivative of f at x	1.2.10
$f^{(n)}(x)$	nth derivative: $(n = 0 \Rightarrow f(x);\ g'(x))$, where $g(x) = f^{(n-1)}(x)$	1.2.11.2
$H_n^{(x)}$	harmonic number of order x: $\sum_{1 \leq k \leq n} 1/k^x$	1.2.7
H_n	harmonic number: $H_n^{(1)}$	1.2.7
F_n	Fibonacci number: $(n \leq 1 \Rightarrow n;\ F_{n-1} + F_{n-2})$	1.2.8
B_n	Bernoulli number	1.2.11.2
$X \cdot Y$	dot product of vectors $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_n)$: $x_1y_1 + \cdots + x_ny_n$	
$(\ldots a_1 a_0 . a_{-1} \ldots)_b$	radix-b positional notation: $\sum_k a_k b^k$	4.1
(min x_1, ave x_2, max x_3, dev x_4)	a random variable having minimum value x_1, average ("expected") value x_2, maximum value x_3, standard deviation x_4	1.2.10

Formal symbolism	Meaning	Section reference
$\binom{x}{k}$	binomial coefficient: $(k < 0 \Rightarrow 0;\ x^{\underline{k}}/k!)$	1.2.6
$\binom{n}{n_1, n_2, \ldots, n_m}$	multinomial coefficient (defined only when $n = n_1 + n_2 + \cdots + n_m$)	1.2.6
$\begin{bmatrix} n \\ m \end{bmatrix}$	Stirling number of the first kind: $\sum_{0<k_1<k_2<\cdots<k_{n-m}<n} k_1 k_2 \ldots k_{n-m}$	1.2.6
$\begin{Bmatrix} n \\ m \end{Bmatrix}$	Stirling number of the second kind: $\sum_{1\le k_1\le k_2\le\cdots\le k_{n-m}\le m} k_1 k_2 \ldots k_{n-m}$	1.2.6
$/x_1, x_2, \ldots, x_n/$	continued fraction: $1/(x_1 + 1/(x_2 + 1/(\cdots + 1/(x_n)\ldots)))$	4.5.3
$((x))$	sawtooth function	3.3.3
$\lvert x\rvert$	absolute value of x: $(x \ge 0 \Rightarrow x;\ -x)$	
$\lVert S\rVert$	cardinal: the number of elements in set S	
$\lfloor x\rfloor$	floor of x, greatest integer function: $\max_{k\le x} k$	1.2.4
$\lceil x\rceil$	ceiling of x, least integer function: $\min_{k\ge x} k$	1.2.4
$\{\, a \mid R(a) \,\}$	set of all a such that the relation $R(a)$ is true	
$\{a_1, \ldots, a_n\}$	the set $\{\, a_k \mid 1 \le k \le n \,\}$	
$\{x\}$	fractional part (used in contexts where a real value, not a set, is implied): $x - \lfloor x\rfloor$	3.3.3
$[y, z)$	half-open interval: $\{\, x \mid y \le x < z \,\}$	
$\langle X_n\rangle$	the infinite sequence $X_0, X_1, X_2, \ldots$ (here the letter n is part of the symbolism)	1.2.9
$\log_b x$	logarithm, base b, of x (real positive x and b, where $b \ne 1$): the y such that $x = b^y$	1.2.2
$\lg x$	binary logarithm: $\log_2 x$	1.2.2
$\ln x$	natural logarithm: $\log_e x$	1.2.2

Formal symbolism	Meaning	Section reference
$\exp x$	exponential of x: e^x	1.2.2
$x \bmod y$	mod function: $\big(y = 0 \Rightarrow x;\ x - y\lfloor x/y \rfloor\big)$	1.2.4
$u(x) \bmod v(x)$	remainder of polynomial u after division by polynomial v	4.6.1
$x \equiv y \ (\text{modulo } z)$	relation of congruence: $x \bmod z = y \bmod z$	1.2.4
$j \setminus k$	j divides k: $k \bmod j = 0$	1.2.4
$\mathrm{B}(x, y)$	beta function	1.2.6
γ	Euler's constant: $\lim_{n\to\infty}(H_n - \ln n)$	1.2.7
$\gamma(x, y)$	incomplete gamma function	1.2.11.3
$\Gamma(x)$	gamma function: $\lim_{y\to\infty} \gamma(x, y)$	1.2.5
$\delta(x)$	characteristic function of the integers	3.3.3
e	base of natural logarithms: $\sum_{n\geq 0} 1/n!$	1.2.2
$\zeta(x)$	zeta function: $\lim_{n\to\infty} H_n^{(x)}$ (when $x > 1$)	1.2.7
$\ell(u)$	leading coefficient of polynomial u	4.6
$l(n)$	length of shortest addition chain for n	4.6.3
$\Lambda(n)$	von Mangoldt's function	4.5.3
$\mu(n)$	Möbius function	4.5.2
$O\big(f(n)\big)$	big-oh of $f(n)$, as the variable $n \to \infty$	1.2.11.1
$O\big(f(x)\big)$	big-oh of $f(x)$, for small values of the variable x (or for x in some specified range)	1.2.11.1
$\varphi(n)$	Euler's totient function: $\sum_{\substack{0\leq k<n \\ \gcd(k,n)=1}} 1$	1.2.4
π	circle ratio: $\sum_{n\geq 0} (-1)^n/(2n+1)$	
ϕ	golden ratio: $\frac{1}{2}\big(1 + \sqrt{5}\big)$	1.2.8
$\emptyset$	empty set: $\{\, x \mid 0 = 1 \,\}$	
∞	infinity: larger than any number	

Formal symbolism	Meaning	Section reference
$\det(A)$	determinant of square matrix A	1.2.3
$\gcd(j,k)$	greatest common divisor of j and k: $\left(j = k = 0 \Rightarrow 0;\ \max_{d\backslash j,\, d\backslash k} d\right)$	4.5.2
$\operatorname{lcm}(j,k)$	least common multiple of j and k: $\left(j = k = 0 \Rightarrow 0;\ \min_{d>0,\, j\backslash d,\, k\backslash d} d\right)$	4.5.2
$\operatorname{sign}(x)$	sign of x: $(x = 0 \Rightarrow 0;\ x/\lvert x\rvert)$	
$\Pr(S(n))$	probability that statement $S(n)$ is true, for "random" integers n (here the letter n is part of the symbolism)	3.5, 4.2.4
$\operatorname{mean}(g)$	mean value of the probability distribution represented by generating function g: $g'(1)$	1.2.10
$\operatorname{var}(g)$	variance of the probability distribution represented by generating function g: $g''(1) + g'(1) - g'(1)^2$	1.2.10
$\deg(u)$	degree of polynomial u	4.6
$\operatorname{cont}(u)$	content of polynomial u	4.6.1
$\operatorname{pp}(u(x))$	primitive part of polynomial u	4.6.1
$\Re(w)$	real part of complex number w	
$\Im(w)$	imaginary part of complex number w	
$\overline{w}$	complex conjugate: $\Re(w) - \Im(w)\,i$	
$\sqcup$	one blank space	1.3.1
rA	register A (accumulator) of MIX	1.3.1
rX	register X (extension) of MIX	1.3.1
rI1, ..., rI6	(index) registers I1, ..., I6 of MIX	1.3.1
rJ	(jump) register J of MIX	1.3.1
(L:R)	partial field of MIX word, $0 \le L \le R \le 5$	1.3.1
OP ADDRESS,I(F)	notation for MIX instruction	1.3.1, 1.3.2
u	unit of time in MIX	1.3.1
*	"self" in MIXAL	1.3.2
0F, 1F, 2F, ..., 9F	"forward" local symbol in MIXAL	1.3.2
0B, 1B, 2B, ..., 9B	"backward" local symbol in MIXAL	1.3.2
0H, 1H, 2H, ..., 9H	"here" local symbol in MIXAL	1.3.2
$\oplus\ominus\otimes\oslash$	rounded or special operations	4.2.1

INDEX AND GLOSSARY

Seek and ye shall find.

—Matthew 7:7

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information; an answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.

Any accuracies or inaccuracies in this index may be explained by the fact that it was prepared with the help of a computer, but not blindly. For additional definitions of computer terminology, see Volume 1 and the *IFIP–ICC Vocabulary of Information Processing* (Amsterdam: North-Holland Publishing Co., 1966).

THIS BOOK was composed at the Stanford Artificial Intelligence Laboratory using the TEX system for technical text developed by the author. The letters and symbols belong to the Computer Modern family of typefaces developed by the author using the METAFONT system for alphabet design. The Chinese characters in the index were designed by Tung Yun Mei using his LCCD system. The paper is S. D. Warren Text, basis 50 lb. (pH 7.8), which has a life expectancy of several hundred years. The offset printing and sewn binding were done at the Maple Press in Manchester, Pennsylvania.

Simon
Plouffe
15/02/85

Knuth

SORTING AND SEARCHING

THE ART OF COMPUTER PROGRAMMING

VOL. 3
1973

THE ART OF COMPUTER PROGRAMMING

DONALD E. KNUTH *Stanford University*

ADDISON-WESLEY PUBLISHING COMPANY

Volume 3 / **Sorting and Searching**

THE ART OF COMPUTER PROGRAMMING

Reading, Massachusetts · Menlo Park, California · London · Don Mills, Ontario

This book is in the
ADDISON-WESLEY SERIES IN
COMPUTER SCIENCE AND INFORMATION PROCESSING

Consulting Editors
RICHARD S. VARGA and MICHAEL A. HARRISON

PREFACE

Cookery is become an art,
a noble science;
cooks are gentlemen.
—TITUS LIVIUS, *Ab Urbe Condita* XXXIX.vi
(Robert Burton, *Anatomy of Melancholy* 1.2.2.2)

This book forms a natural sequel to the material on Information Structures in Chapter 2, because it adds the concept of linearly-ordered data to the other basic structural ideas. The title "Sorting and Searching" may sound as if this book is only for those system programmers who are concerned with the preparation of general-purpose sorting routines or applications to information retrieval. But in fact the area of sorting and searching provides an ideal framework for discussing a wide variety of important general issues:

How are good algorithms discovered?
How can given algorithms and programs be improved?
How can the efficiency of algorithms be analyzed mathematically?
How can a person choose rationally between different algorithms for the same application?
In what senses can algorithms be proved "best possible"?
How does the theory of computing interact with practical considerations?
How can external memories like tapes, drums, or disks be used efficiently with large data bases?

Indeed, I believe that virtually *every* important aspect of programming arises somewhere in the context of sorting or searching!

This volume comprises Chapters 5 and 6 of the complete series. Chapter 5 is concerned with sorting into order; this is a rather large subject which has been divided chiefly into two parts, internal sorting and external sorting. There also are supplementary sections which develop auxiliary theories about permutations (Section 5.1) and about optimum sorting algorithms (Section 5.3). Chapter 6 deals with the problem of searching for specified items in tables or files; this is subdivided into methods which search sequentially, or by comparison of keys, or by digital properties, or by "hashing," and then the more difficult problem of secondary key retrieval is considered. There is a surprising amount

of interplay between both chapters, with strong analogies tying the topics together. Two important varieties of information structures are also discussed, in addition to those considered in Chapter 2, namely priority queues (Section 5.2.3) and linear lists represented as balanced trees (Section 6.2.3).

A reader who is unfamiliar with Volume 1 of this series is urged to consult the Index to Notations (Appendix B), since some of the notational conventions found here are not in general use.

I have been using this book as a text for a student's second course in Data Structures, at the junior-to-graduate level, omitting most of the mathematical material. The mathematical portions of this book would make an appropriate text for a senior or graduate-level course in Analysis of Algorithms, emphasizing especially Sections 5.1, 5.2.2, 6.3, and 6.4. A graduate-level course on Computational Complexity could also be based on Sections 4.3.3, 4.6.3, 4.6.4, 5.3, and 5.4.4.

The rapid rise of Computer Science has delayed the publication of this book by almost three years, since so many aspects of sorting and searching have become so highly developed. I am very grateful for the continuing research support that has been granted to me by the National Science Foundation, the Office of Naval Research, the Institute for Defense Analyses, the International Business Machines Corporation, and Norges Almenvitenskapelige Forskningsrad.

Many people helped me prepare this volume for publication, especially Edward A. Bender, Clark A. Crane, David E. Ferguson, Robert W. Floyd, Ronald L. Graham, Leonidas Guibas, John Hopcroft, Richard M. Karp, Gary D. Knott, Rudolph A. Krutar, Shen Lin, Vaughan R. Pratt, Stephan O. Rice, Richard P. Stanley, J. A. van der Pool, and John W. Wrench, Jr., in addition to the students at Stanford and Berkeley who had to debug the manuscript.

Oslo, Norway
September 1972

D. E. K.

There are certain common Privileges of a Writer,
the Benefit whereof, I hope, there will be no Reason to doubt;
Particularly, that where I am not understood, it shall be concluded,
that something very useful and profound is coucht underneath.
—JONATHAN SWIFT (*Tale of a Tub*, Preface, 1704)

Notes on the Exercises

The exercises in this set of books have been designed for self-study as well as classroom study. It is difficult, if not impossible, for anyone to learn a subject purely by reading about it, without applying the information to specific problems and thereby forcing himself to think about what has been read. Furthermore, we all learn best the things that we have discovered for ourselves. Therefore the exercises form a major part of this work; a definite attempt has been made to keep them as informative as possible and to select problems that are enjoyable to solve.

In many books, easy exercises are found mixed randomly among extremely difficult ones. This is sometimes unfortunate because the reader should have some idea about how much time it ought to take him to do a problem before he tackles it (otherwise he may just skip over all the problems). A classic example of this situation is the book *Dynamic Programming* by Richard Bellman; this is an important, pioneering book in which a group of problems is collected together at the end of some chapters under the heading "Exercises and Research Problems," with extremely trivial questions appearing in the midst of deep, unsolved problems. It is rumored that someone once asked Dr. Bellman how to tell the exercises apart from the research problems, and he replied, "If you can solve it, it is an exercise; otherwise it's a research problem."

Good arguments can be made for including both research problems and very easy exercises in a book of this kind; therefore, to save the reader from the possible dilemma of determining which are which, *rating numbers* have been provided to indicate the level of difficulty. These numbers have the following general significance:

Rating	*Interpretation*
00	An extremely easy exercise which can be answered immediately if the material of the text has been understood, and which can almost always be worked "in your head."
10	A simple problem, which makes a person think over the material just read, but which is by no means difficult. It should be possible to do this in one minute at most; pencil and paper may be useful in obtaining the solution.
20	An average problem which tests basic understanding of the text material but which may take about fifteen to twenty minutes to answer completely.

30 A problem of moderate difficulty and/or complexity which may involve over two hours' work to solve satisfactorily.

40 Quite a difficult or lengthy problem which is perhaps suitable for a term project in classroom situations. It is expected that a student will be able to solve the problem in a reasonable amount of time, but the solution is not trivial.

50 A research problem which (to the author's knowledge at the time of writing) has not yet been solved satisfactorily. If the reader has found an answer to this problem, he is urged to write it up for publication; furthermore, the author of this book would appreciate hearing about the solution as soon as possible (provided it is correct)!

By interpolation in this "logarithmic" scale, the significance of other rating numbers becomes clear. For example, a rating of *17* would indicate an exercise that is a bit simpler than average. Problems with a rating of *50* which are subsequently solved by some reader may appear with a *45* rating in later editions of the book.

The author has earnestly tried to assign accurate rating numbers, but it is difficult for the person who makes up a problem to know just how formidable it will be for someone else; and everyone has more aptitude for certain types of problems than for others. It is hoped that the rating numbers represent a good guess as to the level of difficulty, but they should be taken as general guidelines, not as absolute indicators.

This book has been written for readers with varying degrees of mathematical training and sophistication; and, as a result, some of the exercises are intended only for the use of more mathematically inclined readers. Therefore the rating is preceded by an *M* if the exercise involves mathematical concepts or motivation to a greater extent than necessary for someone who is primarily interested in only the programming algorithms themselves. An exercise is marked with the letters "*HM*" if its solution necessarily involves a knowledge of calculus or other higher mathematics not developed in this book. An "*HM*" designation does *not* necessarily imply difficulty.

Some exercises are preceded by an arrowhead, "▸"; this designates problems which are especially instructive and which are especially recommended. Of course, no reader/student is expected to work *all* of the exercises, and so those which are perhaps the most valuable have been singled out. This is not meant to detract from the other exercises! Each reader should at least make an attempt to solve all of the problems whose rating is *10* or less; and the arrows may help in deciding which of the problems with a higher rating should be given priority.

Solutions to most of the exercises appear in the answer section. Please use them wisely; do not turn to the answer until you have made a genuine effort to solve the problem by yourself, or unless you do not have time to work this particular problem. *After* getting your own solution or giving the problem a

decent try, you may find the answer instructive and helpful. The solution given will often be quite short, and it will sketch the details under the assumption that you have earnestly tried to solve it by your own means first. Sometimes the solution gives less information than was asked; often it gives more. It is quite possible that you may have a better answer than the one published here, or you may have found an error in the published solution; in such a case, the author will be pleased to know the details as soon as possible. Later editions of this book will give the improved solutions together with the solver's name where appropriate.

When working an exercise you may generally use the answer to previous exercises, unless this is specifically forbidden. The rating numbers have been assigned with this in mind; thus it is possible for exercise $n + 1$ to have a lower rating than exercise n, even though it includes the result of exercise n as a special case.

Summary of codes:

▸	Recommended	*00*	Immediate
M	Mathematically oriented	*10*	Simple (one minute)
HM	Requiring "higher math"	*20*	Medium (quarter hour)
		30	Moderately hard
		40	Term project
		50	Research problem

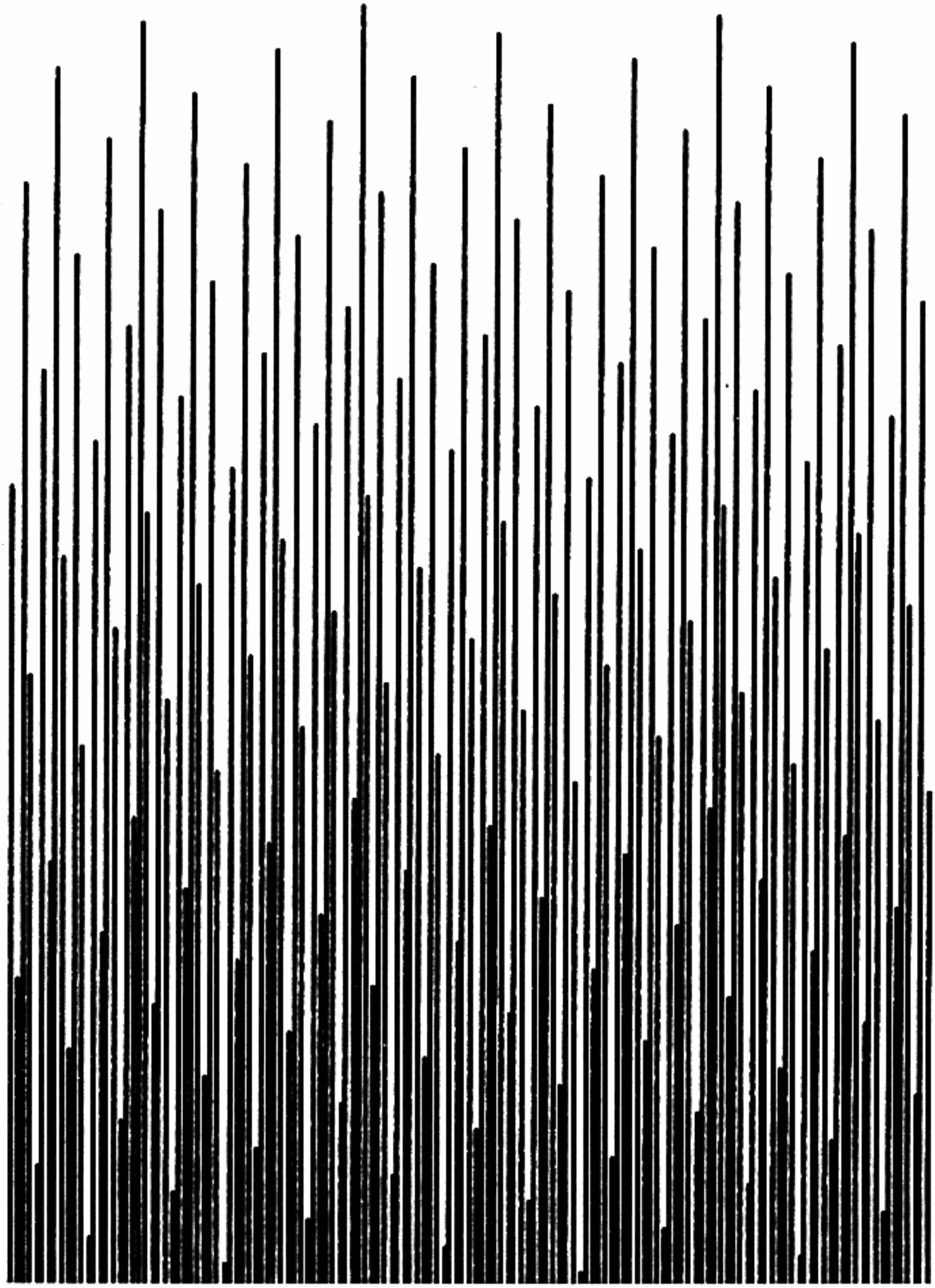
EXERCISES

▸ **1.** [*00*] What does the rating "*M20*" mean?

2. [*10*] Of what value can the exercises in a textbook be to the reader?

3. [*M50*] Prove that when n is an integer, $n > 2$, the equation $x^n + y^n = z^n$ has no solution in positive integers x, y, z.

CONTENTS



CHAPTER FIVE

SORTING

There is nothing more difficult to take in hand,
more perilous to conduct, or more uncertain in its success,
than to take the lead in the introduction of
a new order of things.

— NICCOLÒ MACHIAVELLI (The Prince, 1513)

"But you can't look up all those license
numbers in time," Drake objected.
"We don't have to, Paul. We merely arrange a list
and look for duplications."

— PERRY MASON (The Case of the Angry Mourner, 1951)

"Treesort" Computer — With this new 'computer-approach'
to nature study you can quickly identify over 260
different trees of U.S., Alaska, and Canada,
even palms, desert trees, and other exotics.
To sort, you simply insert the needle.

— Catalog of Edmund Scientific Company (1964)

In this chapter we shall study a topic which arises frequently in programming: the rearrangement of items into ascending or descending order. Imagine how hard it would be to use a dictionary if its words were not listed in alphabetical order; in a similar way, the order in which items are stored in computer memory often has a profound influence on the speed and simplicity of algorithms which manipulate them.

Although dictionaries of the English language define "sorting" as the process of separating or arranging things according to class or kind, it is traditional for computer programmers to use the word in the much more special sense of sorting things into ascending or descending order. The process should perhaps be called *ordering*, not sorting; but anyone who tries to call it "ordering" is soon led into confusion because of the many different meanings attached to this word. Consider the following sentence, for example: "Since only two of our tape drives were in working order, I was ordered to order more tape units in short order, in order to order the data several orders of magnitude faster." Mathematical terminology abounds with still more senses of order (the order

of a group, the order of a permutation, the order of a branch point, relations of order, etc., etc.). So we find that the word "order" can lead to chaos.

Some people have suggested the word "sequencing" as an appropriate name for an ordering process; but this word often seems to lack the right connotation, especially when equal elements are present, and it occasionally conflicts with other terminology. It is quite true that "sorting" is itself an overused word ("He was sort of out of sorts after sorting that sort of data."), but it has become firmly established in computing parlance. Therefore we shall use the word "sorting" chiefly in the strict sense of "sorting into order," without further apologies.

Some of the most important applications of sorting are

a) Solving the "togetherness" problem, in which all items with the same identification are brought together. Suppose that we have 10,000 items in random order, many of which have equal values; and suppose that we want to rearrange this file so that all items with equal values appear in consecutive positions. This is essentially the problem of "sorting" in the older sense of the word; and it can be solved easily by sorting the file in the new sense of the word, so that the values are in ascending order, $v_1 \le v_2 \le \cdots \le v_{10,000}$. (The efficiency which is possible in this procedure explains why the original meaning of "sorting" has changed.)

b) If two or more files have been sorted into the same order, it is possible to find all of the matching entries in one sequential pass through them, without backing up. This is the principle which Perry Mason used to help solve a murder case (see the quotation at the beginning of this chapter). It is usually much more economical to access a list of information in sequence from beginning to end, instead of skipping around at random in the list, unless the list is small enough to be stored in a high-speed random-access memory. Sorting makes it possible to use sequential accessing on large files, as a feasible substitute for direct addressing.

c) Sorting also is an aid to searching, as we shall see in Chapter 6, and therefore it helps to make computer output more suitable for human consumption. In fact, a listing that has been sorted into alphabetic order often looks quite authoritative even when the associated numerical information has been incorrectly computed.

Although sorting has traditionally been used mostly for business data processing, it is actually a basic tool which a programmer should keep in mind for use in a wide variety of situations. We have discussed its use in simplifying algebraic formulas, in exercise 2.3.2–17. The exercises below illustrate the diversity of typical applications.

One of the first large-scale software systems to demonstrate the versatility of sorting was the Larc Scientific Compiler developed by Computer Sciences Corporation in 1960. This optimizing compiler for an extended FORTRAN

language made heavy use of sorting so that the various compilation algorithms were presented with relevant parts of the source program in a convenient sequence. The first pass was a lexical scan which divided the FORTRAN source code into individual tokens, each representing an identifier (e.g., the name of a variable), or a constant, or an operator, etc. Each token was assigned several sequence numbers; when sorted on the name and an appropriate sequence number, all the uses of a given identifier were brought together. The "defining entries," which specify for example whether the identifier is a function name, parameter, or a dimensioned variable, were given a low sequence number, so that they would appear first among the tokens having a given identifier; this made it easy to check for conflicting usage of an identifier, and to allocate storage with respect to `EQUIVALENCE` declarations, etc. The information thus gathered about each identifier was now attached to each token; in this way no "symbol table" of identifiers needed to be maintained in the high-speed memory. The updated tokens were then sorted on another sequence number, which essentially brought the source program back into its original order except that the sequence was specially designed to put arithmetic expressions into a more convenient "Polish prefix" form. Sorting was also used in later phases of compilation, to facilitate loop optimization, to merge error messages into the listing, etc. In short, the compiler was designed so that virtually all the processing could be done sequentially from files that were stored in an auxiliary drum memory, since appropriate sequence numbers were attached to the data in such a way that it could be sorted into various convenient arrangements.

Another, more obvious, application of sorting occurs in file editing routines, where each line is identified by a key number. While a user is typing additions and changes, it is not necessary to have the entire file in memory. The change lines can subsequently be sorted (they are usually pretty much in order anyway), then merged with the original file. This leads to a reasonably efficient way to use memory in a multiprogramming situation. [Cf. C. C. Foster, *Comp. J.* **11** (1968), 134–137.]

Computer manufacturers estimate that over 25 percent of the running time on their computers is currently being spent on sorting, when all their customers are taken into account. There are many installations in which sorting uses more than half of the computing time. From these statistics we may conclude either that (i) there are many important applications of sorting, or (ii) many people sort when they shouldn't, or (iii) inefficient sorting algorithms are in common use. The real truth probably involves some of all three alternatives. In any event we can see that sorting is worthy of serious study, as a practical matter.

But even if sorting were almost useless, there would be plenty of rewarding reasons for studying it anyway! The ingenious algorithms which have been discovered show that sorting is an interesting topic to explore in its own right. There are many fascinating unsolved problems in this area (as well as quite a few solved ones).

From a broader perspective we will find also that sorting algorithms make an interesting *case study* of how to attack computer programming problems in general. Many important principles of data structure manipulation will be illustrated. We will be examining the evolution of various sorting techniques in an attempt to indicate how the reader might have discovered the same ideas for himself (if he had been confronted with the problem before anyone else). By extrapolation of this case study we may learn a good deal about that state of mind which helps to design good algorithms for other computer problems.

Sorting techniques also provide excellent illustrations of the general ideas involved in the *analysis of algorithms*, i.e., the ideas used to determine performance characteristics of algorithms so that an intelligent choice can be made between competing methods. Readers who are mathematically inclined will find quite a few instructive techniques in this chapter for estimating the speed of computer algorithms and for solving complicated recurrence relations. On the other hand, the material has been arranged so that readers without a mathematical bent can safely skip over these calculations.

Before going on, we ought to define our problem a little more clearly, and introduce some terminology. We are given N items

$$R_1, R_2, \ldots, R_N$$

to be sorted; we shall call them *records*, and the entire collection of N records will be called a *file*. Each record R_j has a *key*, K_j, which governs the sorting process. There may be additional information, besides the key, in a record; this extra "satellite information" has no effect on the sorting process except that it remains with the same record.

An ordering relation "$<$" is specified on the keys so that, for any three key values a, b, c, the following conditions are satisfied:

i) Exactly one of the possibilities $a < b$, $a = b$, $b < a$ is true. (This is the law of trichotomy.)

ii) If $a < b$ and $b < c$, then $a < c$. (The law of transitivity.)

These two properties characterize the mathematical concept of *linear ordering*, also called *total ordering*. Any relationship "$<$" satisfying (i) and (ii) can be sorted by most of the methods to be mentioned in this chapter, although some sorting techniques require numerical or alphabetic keys with the usual ordering.

The goal of sorting is to determine a permutation $p(1)p(2) \ldots p(N)$ of the records, which puts the keys in nondecreasing order:

$$K_{p(1)} \le K_{p(2)} \le \cdots \le K_{p(N)}. \tag{1}$$

The sorting is called *stable* if we make the further requirement that records with equal keys retain their original relative order, i.e., if

$$p(i) < p(j) \quad \text{whenever} \quad K_{p(i)} = K_{p(j)} \quad \text{and} \quad i < j. \tag{2}$$

In some cases we will want the records to be physically rearranged in storage so that their keys are in order, while in other cases it will be sufficient merely to have an auxiliary table which specifies the permutation in some way, so that the records can be accessed in order of their keys.

A few of the sorting methods assume the existence of either or both of the values "∞" and "$-\infty$" which are defined to be greater than or less than all keys, respectively:

$$-\infty < K_j < \infty, \qquad 1 \le j \le N. \tag{3}$$

These values are substituted for artificial keys and they also are used as sentinel indicators. The case of equality is excluded in (3); if equality can occur, the algorithms can be modified so that they will still work, but usually at the expense of some elegance and efficiency.

Sorting can be classified generally into *internal sorting*, in which the records are kept in the computer's high-speed random-access memory; and *external sorting*, when there are more records than can be held in memory at once. Internal sorting allows more flexibility in the structuring and accessing of the data, while external sorting shows us how to live with rather stringent accessing constraints.

The time required to sort N records, using a decent general-purpose sorting algorithm, is roughly proportional to $N \log N$; we make about $\log N$ "passes" over the data. This is the minimum possible time, as we shall see in Section 5.3.1. Thus if we double the number of records, it will take a little more than twice as long to sort them, all other things being equal. (Actually, as N approaches infinity, a better indication of the time needed to sort is $N(\log N)^2$, if the keys are distinct, since the size of the keys grows with N; but for practical purposes, N never really approaches infinity.)

EXERCISES—First Set

1. [*M20*] Prove, from the laws of trichotomy and transitivity, that the permutation $p(1)\ p(2)\ \ldots\ p(N)$ is *uniquely* determined when the sorting is assumed to be stable.

2. [*21*] Assume that each record R_j in a certain file contains *two* keys, a "major key" K_j and a "minor key" k_j, with a linear ordering $<$ defined on each of the sets of keys. Then we can define "lexicographic order" between pairs of keys (K, k) in the usual way:

$$(K_i, k_i) < (K_j, k_j) \quad \text{if} \quad K_i < K_j \quad \text{or if} \quad K_i = K_j \quad \text{and} \quad k_i < k_j.$$

A man named Mr. *A* took this file and sorted it first on the major keys, obtaining n groups of records with equal major keys in each group,

$$K_{p(1)} = \cdots = K_{p(i_1)} < K_{p(i_1+1)} = \cdots = K_{p(i_2)} < \cdots < K_{p(i_{n-1}+1)} = \cdots = K_{p(i_n)},$$

where $i_n = N$. Then he sorted each of the n groups $R_{p(i_{j-1})+1}, \ldots, R_{p(i_j)}$ on their minor keys.

Mr. B took the same original file and sorted it first on the minor keys; then he took the resulting file, and sorted it on the major keys.

Mr. C took the same original file and did a single sorting operation on it, using lexicographic order on the major and minor keys (K_j, k_j).

Did all three men obtain the same result?

3. [*M25*] Let $<$ be a relation on $K_1, \ldots, K_N$ which satisfies the law of trichotomy but *not* the transitive law. Prove that even without the transitive law it is possible to sort the records in a stable manner, meeting conditions (1) and (2); in fact, there are at least three arrangements which satisfy the conditions!

4. [*15*] Mr. B. C. Dull (a MIX programmer) wanted to know if the number stored in location A is greater than, less than, or equal to the number stored in location B. So he wrote

```
LDA  A
SUB  B
```

and tested whether register A was positive, negative, or zero. What serious mistake did he make, and what should he have done?

5. [*17*] Write a MIX subroutine for multiprecision comparison of keys, having the following specifications:

Calling sequence: `JMP COMPARE`

Entry conditions: rI1 $= n$; CONTENTS(A $+ k$) $= a_k$, CONTENTS(B $+ k$) $= b_k$, for $1 \leq k \leq n$; assume that $n \geq 1$.

Exit conditions:
CI $= +1$, if $(a_n, \ldots, a_1) > (b_n, \ldots, b_1)$;
CI $= 0$, if $(a_n, \ldots, a_1) = (b_n, \ldots, b_1)$;
CI $= -1$, if $(a_n, \ldots, a_1) < (b_n, \ldots, b_1)$;
rX and rI1 possibly affected.

Here the relation $(a_n, \ldots, a_1) < (b_n, \ldots, b_1)$ denotes lexicographic ordering from left to right; i.e., there is an index j such that $a_k = b_k$ for $n \geq k > j$, but $a_j < b_j$.

▸ **6.** [*30*] Locations A and B contain two numbers a and b, respectively. Show that it is possible to write a MIX program which computes and stores min (a, b) in location C, *without using any jump operators.* (*Caution:* Since you will not be able to test whether or not arithmetic overflow has occurred, it is wise to guarantee that overflow is impossible regardless of the values of a and b.)

7. [*M27*] After N independent, uniformly distributed random variables between 0 and 1 have been sorted into nondecreasing order, what is the probability that the rth smallest of these is $\leq x$?

EXERCISES—Second Set

Each of the following exercises states a problem which a computer programmer might face. Suggest a "good" way to solve the problem, assuming that only a relatively

small amount of internal memory is available, supplemented by about half a dozen tape units (enough tape units for sorting).

8. [*15*] You are given a tape containing one million words of data. How do you determine how many distinct words are present on the tape?

9. [*18*] You are the U. S. Internal Revenue Service; you receive millions of "information" forms from organizations telling how much income they have paid to people, and millions of "tax" forms from people telling how much income they have been paid. How do you catch people who don't report all of their income?

10. [*M25*] (*Transposing a matrix.*) You are given a magnetic tape containing one million words, representing the elements of a 1000×1000 matrix stored in order by rows: $a_{1,1}\ a_{1,2} \ldots a_{1,1000}\ a_{2,1} \ldots a_{2,1000} \ldots a_{1000,1000}$. How do you create a tape in which the elements are stored by columns $a_{1,1}\ a_{2,1} \ldots a_{1000,1}\ a_{1,2} \ldots a_{1000,2} \ldots a_{1000,1000}$ instead? (Try to make at most ten passes over the data.)

11. [*M26*] Given a large file of N words, how would you "shuffle" it into a random rearrangement?

▶ **12.** [*24*] A certain university has about 1000 faculty members and 500 committees; every faculty member is expected to be a member of at least two committees. Your job is to prepare a nicely-printed master list of all committees. For this purpose you are given a deck of about 1500 cards, in random order, punched as follows:

Faculty cards. Column 1, blank; columns 2–18, last name followed by blanks; columns 19–20, initials; columns 21–23, first committee number; columns 24–26, second committee number; . . . ; columns 78–80, twentieth committee number (if necessary), or blank.

Committee cards. Column 1, "*"; columns 2–77, name of committee; columns 78–80, number of committee.

How should you proceed? (Spell out your method in some detail.)

13. [*20*] You are working with two computer systems which have different conventions for the "collating sequence" (i.e., ordering relation) of alphameric characters. How do you make one computer sort alphameric files in the order used by the other computer?

14. [*18*] You are given a list of the names of a fairly large number of people born in the U.S.A., together with the name of the state where they were born. How do you count the number of people born in each state? (Assume that nobody appears in the list more than once.)

15. [*20*] In order to make it easier to make changes to large FORTRAN programs, you want to design a "cross-reference" routine; such a routine takes FORTRAN programs as input and prints them together with an index which shows each use of each identifier (i.e., each name) in the program. How should such a routine be designed?

16. [*33*] (*Library card sorting.*) The manner of alphabetizing catalog cards varies somewhat from one library to another; the following "alphabetical" listing indicates many of the procedures recommended in the *American Library Association Rules for Filing Catalog Cards* (Chicago, 1942):

Text of card	*Remarks*
R. Accademia nazionale dei Lincei, Rome	Ignore foreign royalty (except British)
1812; ein historischer roman.	Achtzehnhundert zwölf
Bibliothèque d'histoire révolutionnaire.	Treat apostrophe as space in French
Bibliothèque des curiosités.	Ignore accents on letters
Brown, Mrs. J. Crosby	Ignore designation of rank
Brown, John	Names with dates follow those without
Brown, John, mathematician	. . . the latter are subarranged by descriptive words
Brown, John, of Boston	
Brown, John, 1715–1766	Arrange identical names by birthdate
BROWN, JOHN, 1715–1766	Works "about" follow works "by"
Brown, John, d. 1811	Sometimes birthdate must be estimated
Brown, Dr. John, 1810–1882	Ignore designation of rank
Brown-Williams, Reginald Makepeace	Hyphen treated as space
Brown America.	Book titles follow compound names
Brown & Dallison's Nevada directory.	& in English becomes "and"
Brownjohn, Alan	
Den', Vladimir Éduardovich, 1867–	Ignore apostrophe in names
The den.	Ignore an initial article
Den lieben süssen mädeln.	. . . provided it's in nominative case
Dix, Morgan, 1827–1908	Names before words
1812 ouverture.	Dix-huit cent douze
Le XIXe sicèle français.	Dix-neuvième
The 1847 issue of U. S. stamps.	Eighteen forty-seven
1812 overture.	Eighteen twelve
I am a mathematician.	(by Norbert Weiner)
IBM journal of research and development.	Initials are like one-letter words
ha-I ha-eḥad.	Ignore initial article
Ia; a love story.	Ignore punctuation in titles
International Business Machines Corporation	
al-Khuwārizmī, Muḥammad ibn Mūsā, *fl.* 813–846	Ignore initial "al-" in Arabic names
Labour; a magazine for all workers.	Respell it "Labor"
Labor research association	
Labour, *see* Labor	Cross-reference card
McCall's cookbook	Ignore apostrophe in English
McCarthy, John, 1927–	Mc = Mac
Machine-independent computer programming.	Treat hyphen as space
MacMahon, Maj. Percy Alexander, 1854–1929	Ignore designation of rank
Mrs. Dalloway.	"Mrs." = "Mistress"
Mistress of mistresses.	
Royal society of London	
St. Petersburger Zeitung.	"St." = "Saint", even in German
Saint-Saëns, Camille, 1835–1921	Hyphen treated as space
Ste. Anne des Monts, Quebec	Sainte
Seminumerical algorithms.	

Text of card	*Remarks*
Uncle Tom's cabin.	
U. S. Bureau of the census.	"U. S." = "United States"
Vandermonde, Alexander Théophile, 1735–1796	
Van Valkenburg, Mac Elwyn, 1921–	Ignore space after prefix in surnames
Von Neumann, John, 1903–1957	
The whole art of legerdemain.	
Who's afraid of Virginia Woolf?	Ignore apostrophe in English
Wijngaarden, Adriaan van, 1916–	Surname doesn't begin with lower case letter

(Most of these rules are subject to certain exceptions, and there are many other rules not illustrated here.)

If you were given the job of sorting large numbers of library cards by computer, and eventually maintaining a very large file of such cards, and if you had no chance to change these long-standing policies of card filing, how would you arrange the data in such a way that the sorting and merging operations are facilitated?

17. [*M21*] (*Discrete logarithms.*) You know that p is a (rather large) prime number, and that a is a primitive root modulo p. Therefore, for all b in the range $1 \le b < p$, there is a unique n such that $a^n \bmod p = b$, $1 \le n < p$. How do you find n, given b, in less than $O(n)$ steps?

[*Hint:* Let $m = \lceil \sqrt{p} \rceil$ and try to solve $a^{mn_1} \equiv ba^{-n_2}$ (modulo p) for $0 \le n_1, n_2 < m$.]

18. [*M25*] (E. T. Parker.) Euler conjectured that there are no solutions to the equation

$$u^6 + v^6 + w^6 + x^6 + y^6 = z^6$$

in nonnegative integers u, v, w, x, y, z, except for the trivial solutions when at least four of the variables are zero. At the same time he conjectured that

$$x_1^n + \cdots + x_{n-1}^n = x_n^n$$

had no nontrivial solutions, for all $n \ge 3$, but this conjecture has been disproved by the computer-discovered identity $27^5 + 84^5 + 110^5 + 133^5 = 144^5$; see L. J. Lander, T. R. Parkin, and J. L. Selfridge, *Math. Comp.* **21** (1967), 446–459. Can you think of a way in which sorting would help in the search for counterexamples to Euler's conjecture when $n = 6$?

▸ **19.** [*24*] Given a file containing a large number of distinct 30-bit binary words $x_1, \ldots, x_N$, what is a good way to find all *complementary* pairs (x_i, x_j) that are present? (Two words are complementary when one has 0 wherever the other has 1, and conversely; thus they are complementary if and only if their sum is $(11\ldots1)_2$, when they are treated as binary numbers.)

▸ **20.** [*25*] Given a file containing 1000 30-bit binary words $x_1, \ldots, x_{1000}$, how would you prepare a list of all pairs (x_i, x_j) such that $x_i = x_j$ except in at most two bit positions?

▶ **21.** [*22*] How would you go about looking for five-letter anagrams such as CARET, CARTE, CATER, CRATE, REACT, TRACE; CRUEL, LUCRE, ULCER; DOWRY, ROWDY, WORDY? [One might wish to know whether there are any sets of ten or more five-letter English anagrams besides the remarkable set

APERS, ASPER, PARES, PARSE, PEARS, PRASE, RAPES, REAPS, SPARE, SPEAR,

to which we might add the French word APRÈS.]

22. [*M28*] Given the specifications of a fairly large number of directed graphs, what approach will be useful for grouping the *isomorphic* ones together? (Directed graphs are isomorphic if there is a one-to-one correspondence between their vertices and a one-to-one correspondence between their arcs, where the correspondences preserve incidence between vertices and arcs.)

23. [*30*] In a certain group of 4096 people, everyone has about 100 acquaintances. A file has been prepared listing all pairs of people who are acquaintances. (The relation is symmetric; i.e., if x is acquainted with y, then y is acquainted with x. Therefore the file contains roughly 200,000 entries.) How would you design an algorithm to list all the k-person *cliques* in this group of people, given k? (A clique is an instance of mutual acquaintances: Everyone in the clique is acquainted with everyone else.) Assume that no really large cliques are present.

24. [*30*] Three million people with distinct names were laid end-to-end, reaching from New York to California. Each participant was given a slip of paper on which he wrote down his own name and the name of the person immediately west of him in the line. The man at the extreme western end didn't understand what to do, so he threw his paper away; the remaining 2,999,999 slips of paper were put into a huge basket and taken to the National Archives in Washington, D. C. Here the contents of the basket were shuffled completely and transferred to magnetic tapes.

At this point an information scientist observed that there was enough information on the tapes to reconstruct the list of people in their original order. And a computer scientist discovered a way to do the reconstruction with less than 1000 passes through the data tapes, using only sequential accessing of tape files and a small amount of random-access memory. How did he do it?

[In other words, given the pairs (x_i, x_{i+1}), for $1 \le i < N$, in random order, where the x_i are distinct, how can the sequence $x_1, x_2, \ldots, x_N$ be obtained, restricting all operations to serial techniques suitable for use with magnetic tapes? This is the problem of sorting into order when there is no easy way to tell which of two given keys precedes the other; we have already raised this question as part of exercise 2.2.3–25.]

*5.1. COMBINATORIAL PROPERTIES OF PERMUTATIONS

A permutation of a finite set is an arrangement of its elements into a row. Permutations are of special importance in the study of sorting algorithms, since they represent the unsorted input data. In order to study the efficiency of different sorting methods, we will want to be able to count the number of permutations which cause a certain step of a sorting procedure to be executed a certain number of times.

We have, of course, met with permutations already many times in Chapters 1, 2, and 3. For example in Section 1.2.5, we discussed two basic theoretical methods of constructing the $n!$ permutations of n objects; in Section 1.3.3, we analyzed some algorithms dealing with the cycle structure and multiplicative properties of permutations; in Section 3.3.2 we studied their "runs up" and "runs down." The purpose of the present section is to study some other properties of permutations, and to consider the general case where equal elements are allowed to appear. In the course of this study we will learn a good deal about combinatorial mathematics.

The properties of permutations are sufficiently pleasing to be interesting in their own right, and it is convenient to develop them systematically in one place instead of scattering the material throughout this chapter. But readers who are not mathematically inclined and readers who are anxious to dive right into sorting techniques are advised to go on to Section 5.2 immediately, since the present section actually has little *direct* connection to sorting.

*5.1.1. Inversions

Let $a_1\, a_2 \ldots a_n$ be a permutation of the set $\{1, 2, \ldots, n\}$. If $i < j$ and $a_i > a_j$, the pair (a_i, a_j) is called an "inversion" of the permutation; for example, the permutation 3 1 4 2 has three inversions: (3, 1), (3, 2), and (4, 2). Each inversion is a pair of elements that is "out of sort," so the only permutation with no inversions is the sorted permutation $1\ 2 \ldots n$. This connection with sorting is the chief reason why we will be so interested in inversions, although we have already used the concept to analyze a dynamic storage allocation algorithm (see exercise 2.2.2–9).

The concept of inversions was introduced by G. Cramer in 1750 [*Intr. à l'Analyse des Lignes Courbes algébriques* (Geneva, 1750), 657–659; cf. Thomas Muir, *Theory of Determinants* **1** (1906), 11–14], in connection with his famous rule for solving linear equations. In essence, he defined the determinant of an $n \times n$ matrix in the following way:

$$\det \begin{pmatrix} x_{11} & x_{12} & \ldots & x_{1n} \\ \vdots & \vdots & & \vdots \\ x_{n1} & x_{n2} & \ldots & x_{nn} \end{pmatrix} = \sum (-1)^{I(a_1 a_2 \ldots a_n)} x_{1a_1} x_{2a_2} \ldots x_{na_n},$$

summed over all permutations $a_1\, a_2 \ldots a_n$, where $I(a_1\, a_2 \ldots a_n)$ is the number of inversions of the permutation.

The *inversion table* $b_1\ b_2 \ldots b_n$ of the permutation $a_1\ a_2 \ldots a_n$ is obtained by letting b_j be the number of elements to the *left* of j that are *greater* than j. (In other words, b_j is the number of inversions whose second component is j.) For example, the permutation

$$5\ 9\ 1\ 8\ 2\ 6\ 4\ 7\ 3 \tag{1}$$

has the inversion table

$$2\ 3\ 6\ 4\ 0\ 2\ 2\ 1\ 0, \tag{2}$$

since 5 and 9 are to the left of 1; 5, 9, 8 are to the left of 2; etc., 20 inversions in all. By definition we will always have

$$0 \le b_1 \le n-1, \quad 0 \le b_2 \le n-2, \quad \ldots, \quad 0 \le b_{n-1} \le 1, \quad b_n = 0. \tag{3}$$

Perhaps the most important fact about inversions is Marshall Hall's observation that *an inversion table uniquely determines the corresponding permutation.* [See *Proc. Symp. Applied Math.* **6** (American Math. Society, 1956), 203.] We can go back from any inversion table $b_1\ b_2 \ldots b_n$ satisfying (3) to the unique permutation which produces it, by successively determining the relative placement of the elements $n, n-1, \ldots, 1$ (in this order). For example, we can construct the permutation corresponding to (2) as follows: Write down the number 9; then since $b_8 = 1$, 8 is to follow 9. Since $b_7 = 2$, 7 comes after both 8 and 9. Since $b_6 = 2$, 6 follows two of the numbers already written down; so we have

$$9\ 8\ 6\ 7$$

so far. Continue by placing 5 at the left, since $b_5 = 0$; put 4 after 4 of the numbers; put 3 after 6 numbers (i.e. at the extreme right); and we have

$$5\ 9\ 8\ 6\ 4\ 7\ 3.$$

Inserting 2 and 1 in an analogous way yields (1).

This correspondence is important because we can often translate a problem stated in terms of permutations into an equivalent problem stated in terms of inversion tables, and the latter problem may be easier to solve. For example, consider the simplest question of all: How many permutations of $\{1, 2, \ldots, n\}$ are possible? The answer must be the number of possible inversion tables, and they are easily enumerated since there are n choices for b_1, independently $n-1$ choices for $b_2, \ldots, 1$ choice for b_n, making $n(n-1)\ldots 1 = n!$ choices in all. Inversions are easier to count, because the b's are completely independent of each other, while the a's must be mutually distinct.

In Section 1.2.10 we analyzed a problem dealing with the number of local maxima of a permutation, when it is read from right to left; in other words, we wanted to count how many elements are larger than any of their successors. (For example, the right-to-left maxima in (1) are 9, 8, 7, and 3.) This is the

number of j such that $b_j = n - j$. Since b_1 will equal $n - 1$ with probability $1/n$, and (independently) b_2 will be equal to $n - 2$ with probability $1/(n - 1)$, etc., it is clear by consideration of the inversions that the average number of right-to-left maxima is

$$\frac{1}{n} + \frac{1}{n-1} + \cdots + 1 = H_n.$$

The corresponding generating function is also easily derived in a similar way.

Other applications of inversion tables appear later in this chapter, in connection with specific sorting algorithms.

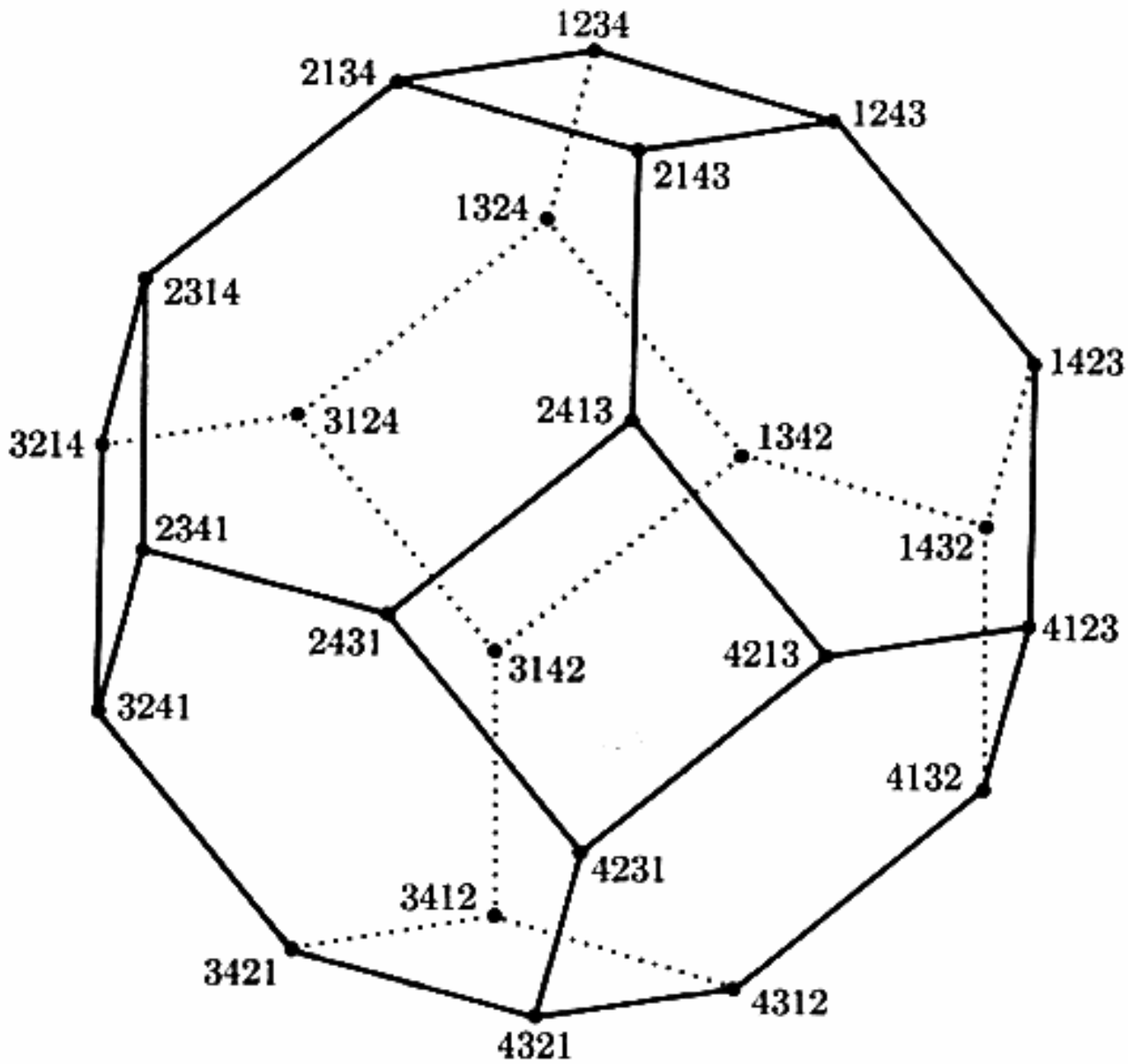


Fig. 1. The truncated octahedron, which shows the change in inversions when adjacent elements of a permutation are interchanged.

If we interchange two *adjacent* elements of a permutation, it is easy to see that the total number of inversions will increase or decrease by unity. Figure 1 shows the 24 permutations of $\{1, 2, 3, 4\}$, with lines joining permutations that differ by an interchange of adjacent elements; following a line downward increases the number of inversions by unity. Hence the number of inversions of a permutation π is the length of a downward path from 1 2 3 4 to π in Fig. 1; all such paths must have the same length.

Note that the diagram may be viewed as a three-dimensional solid, the "truncated octahedron," which has 8 hexagonal faces and 6 square faces. This is one of the uniform polyhedra discussed by Archimedes (see exercise 10).

The reader should not confuse "inversions" of a permutation with the "inverse" of a permutation. Recall that we can write a permutation in two-line form

$$\begin{pmatrix} 1 & 2 & 3 & \dots & n \\ a_1 & a_2 & a_3 & \dots & a_n \end{pmatrix}; \tag{4}$$

the *inverse* $a'_1\, a'_2\, a'_3 \dots a'_n$ of this permutation is the permutation obtained by interchanging the two rows and then sorting the columns into increasing order of the new top row:

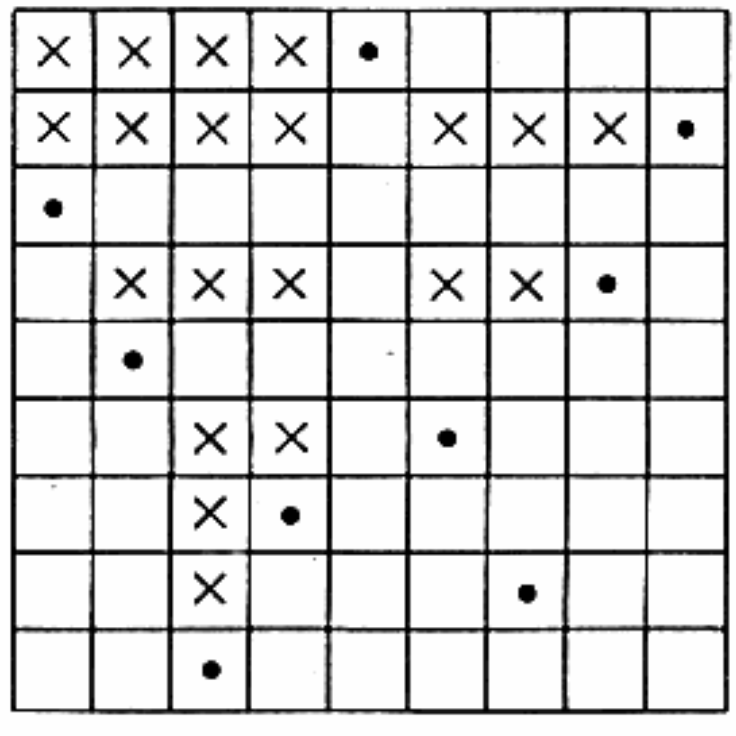
$$\begin{pmatrix} a_1 & a_2 & a_3 & \dots & a_n \\ 1 & 2 & 3 & \dots & n \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & \dots & n \\ a'_1 & a'_2 & a'_3 & \dots & a'_n \end{pmatrix}. \tag{5}$$

For example, the inverse of 5 9 1 8 2 6 4 7 3 is 3 5 9 7 1 6 8 4 2, since

$$\begin{pmatrix} 5 & 9 & 1 & 8 & 2 & 6 & 4 & 7 & 3 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 5 & 9 & 7 & 1 & 6 & 8 & 4 & 2 \end{pmatrix}.$$

Another way to define the inverse is to say that $a'_j = k$ if and only if $a_k = j$.

The inverse of a permutation was first defined by H. A. Rothe [in K. F. Hindenburg (ed.), *Sammlung combinatorisch-analytischer Abhandlungen* **2** (Leipzig, 1800), 263–305], who noticed an interesting connection between inverses and inversions: *The inverse of a permutation has exactly as many inversions as the permutation itself.* Rothe's proof of this fact was not the simplest possible one, but it is instructive and rather pretty nevertheless. We construct an $n \times n$ chessboard having a dot in column j of row i whenever $a_i = j$. Then we put X's in all squares which have dots lying both below (in the same column) and to their right (in the same row). For example, the diagram for 5 9 1 8 2 6 4 7 3 is



The number of X's is the number of inversions, since it is easy to see that b_j is the number of X's in column j. Now if we transpose the diagram (interchanging rows and columns), we get the diagram corresponding to the inverse of the original permutation; hence the number of X's (the number of inversions) is the same in both cases. Rothe used this fact to prove that the determinant of a matrix is unchanged when the matrix is transposed.

The analysis of some sorting algorithms involves the knowledge of how many permutations of n elements have exactly k inversions. Let us denote that number by $I_n(k)$; Table 1 lists the first few values of this function.

Table 1

PERMUTATIONS WITH k INVERSIONS

n	$I_n(0)$	$I_n(1)$	$I_n(2)$	$I_n(3)$	$I_n(4)$	$I_n(5)$	$I_n(6)$	$I_n(7)$	$I_n(8)$	$I_n(9)$	$I_n(10)$	$I_n(11)$
1	1	0	0	0	0	0	0	0	0	0	0	0
2	1	1	0	0	0	0	0	0	0	0	0	0
3	1	2	2	1	0	0	0	0	0	0	0	0
4	1	3	5	6	5	3	1	0	0	0	0	0
5	1	4	9	15	20	22	20	15	9	4	1	0
6	1	5	14	29	49	71	90	101	101	90	71	49

By considering the inversion table $b_1\, b_2 \ldots b_n$, it is obvious that $I_n(0) = 1$, $I_n(1) = n - 1$, and there is a symmetry property

$$I_n\left(\binom{n}{2} - k\right) = I_n(k). \tag{6}$$

Furthermore, since each of the b's can be chosen independently of the others, it is not difficult to see that the generating function

$$G_n(z) = I_n(0) + I_n(1)z + I_n(2)z^2 + \cdots \tag{7}$$

satisfies $G_n(z) = (1 + z + \cdots + z^{n-1})G_{n-1}(z)$; hence it has the comparatively simple form

$$(1 + z + \cdots + z^{n-1}) \ldots (1 + z)(1) = (1 - z^n) \ldots (1 - z^2)(1 - z)/(1 - z)^n. \tag{8}$$

From this generating function, we can easily extend Table 1, and we can verify that the numbers below the zigzag line in that table satisfy

$$I_n(k) = I_n(k - 1) + I_{n-1}(k), \qquad \text{for} \qquad k < n. \tag{9}$$

(This relation does *not* hold *above* the zigzag line.) A more complicated argument (see exercise 14) shows that, in fact, we have the formulas

$$I_n(2) = \binom{n}{2} - 1, \qquad n \geq 2;$$

$$I_n(3) = \binom{n+1}{3} - \binom{n}{1}, \qquad n \geq 3;$$

$$I_n(4) = \binom{n+2}{4} - \binom{n+1}{2}, \qquad n \geq 4;$$

$$I_n(5) = \binom{n+3}{5} - \binom{n+2}{3} + 1, \qquad n \geq 5;$$

in general, the formula for $I_n(k)$ contains about $1.6\sqrt{k}$ terms:

$$I_n(k) = \binom{n+k-2}{k} - \binom{n+k-3}{k-2} + \binom{n+k-6}{k-5} + \binom{n+k-8}{k-7} - \cdots$$
$$+ (-1)^j \left(\binom{n+k-u_j-1}{k-u_j} + \binom{n+k-u_j-j-1}{k-u_j-j} \right) + \cdots, \qquad n \geq k, \tag{10}$$

where $u_j = (3j^2 - j)/2$ is a so-called "pentagonal number."

If we divide $G_n(z)$ by $n!$ we get the generating function $g_n(z)$ for the probability distribution of the number of inversions in a random permutation of n elements. This is the product

$$g_n(z) = h_1(z)h_2(z) \ldots h_n(z), \tag{11}$$

where $h_k(z) = (1 + z + \cdots + z^{k-1})/k$ is the generating function for the uniform distribution of a random nonnegative integer less than k. It follows that

$$\begin{aligned} \text{mean}\,(g_n) &= \text{mean}\,(h_1) + \text{mean}\,(h_2) + \cdots + \text{mean}\,(h_n) \\ &= 0 + \frac{1}{2} + \cdots + \frac{n-1}{2} = \frac{n(n-1)}{4}; \end{aligned} \tag{12}$$

$$\begin{aligned} \text{var}\,(g_n) &= \text{var}\,(h_1) + \text{var}\,(h_2) + \cdots + \text{var}\,(h_n) \\ &= 0 + \frac{1}{4} + \cdots + \frac{n^2-1}{12} = \frac{n(2n+5)(n-1)}{72}. \end{aligned} \tag{13}$$

So the average number of inversions is rather large, about $\frac{1}{4}n^2$; the standard deviation is also rather large, about $\frac{1}{6}n^{3/2}$.

To bring our study of inversions to an interesting conclusion, we will discuss a remarkable discovery made by P. A. MacMahon [*Amer. J. Math.* **35** (1913), 281–322]. Let us define the *index* of the permutation $a_1\, a_2 \ldots a_n$ as the sum of all subscripts j such that $a_j > a_{j+1}$, $1 \leq j < n$. For example, the index of

5 9 1 8 2 6 4 7 3 is $2+4+6+8=20$. By coincidence the index is the same as the number of inversions in this case. If we list the 24 permutations of $\{1, 2, 3, 4\}$, namely

Permutation	Index	Inversions	Permutation	Index	Inversions
1 2 3 4	0	0	3|1 2 4	1	2
1 2 4|3	3	1	3|1 4|2	4	3
1 3|2 4	2	1	3|2|1 4	3	3
1 3 4|2	3	2	3|2 4|1	4	4
1 4|2 3	2	2	3 4|1 2	2	4
1 4|3|2	5	3	3 4|2|1	5	5
2|1 3 4	1	1	4|1 2 3	1	3
2|1 4|3	4	2	4|1 3|2	4	4
2 3|1 4	2	2	4|2|1 3	3	4
2 3 4|1	3	3	4|2 3|1	4	5
2 4|1 3	2	3	4|3|1 2	3	5
2 4|3|1	5	4	4|3|2|1	6	6

we see that *the number of permutations having a given index, k, is the same as the number having k inversions.*

At first this fact might appear to be almost obvious, but further scrutiny makes it very mysterious, and no simple direct proof is evident. MacMahon gave an ingenious indirect proof, as follows: Let $J(a_1\, a_2 \ldots a_n)$ be the index of the permutation $a_1\, a_2 \ldots a_n$, and let

$$H_n(z) = \sum z^{J(a_1\, a_2 \ldots a_n)}, \tag{14}$$

summed over all permutations of $\{1, 2, \ldots, n\}$, be the corresponding generating function. We wish to show that $H_n(z) = G_n(z)$. For this purpose we will define a one-to-one correspondence between n-tuples $(q_1, q_2, \ldots, q_n)$ of nonnegative integers, on the one hand, and ordered pairs of n-tuples,

$$((a_1, a_2, \ldots, a_n), (p_1, p_2, \ldots, p_n))$$

on the other hand, where $a_1\, a_2 \ldots a_n$ is a permutation of $\{1, 2, \ldots, n\}$ and $p_1 \geq p_2 \geq \cdots \geq p_n \geq 0$. This correspondence will satisfy the condition

$$q_1 + q_2 + \cdots + q_n = J(a_1\, a_2 \ldots a_n) + (p_1 + p_2 + \cdots + p_n). \tag{15}$$

The generating function $\sum z^{q_1+q_2+\cdots+q_n}$, summed over all n-tuples of nonnegative integers $(q_1, q_2, \ldots, q_n)$, is $Q_n(z) = 1/(1-z)^n$; and the generating function $\sum z^{p_1+p_2+\cdots+p_n}$, summed over all n-tuples of integers $(p_1, p_2, \ldots, p_n)$ such that $p_1 \geq p_2 \geq \cdots \geq p_n \geq 0$, is

$$P_n(z) = 1/(1-z)(1-z^2)\ldots(1-z^n), \tag{16}$$

as shown in exercise 15. In view of (15), the one-to-one correspondence we are about to establish will prove that $Q_n(z) = H_n(z)P_n(z)$, i.e.,

$$H_n(z) = Q_n(z)/P_n(z) = G_n(z).$$

The desired correspondence is defined by means of a "sorting" algorithm. Start with an empty list, and for $k = 1, 2, \ldots, n$ (in this order) insert q_k into the list as follows: Let the list after $k - 1$ steps contain $p_1, p_2, \ldots, p_{k-1}$, where $p_1 \geq p_2 \geq \cdots \geq p_{k-1}$; and assume that a permutation $a_1\, a_2 \ldots a_{k-1}$ on $\{n, n-1, \ldots, n-k+2\}$ has been defined. Let j be the unique integer such that $p_j > q_k \geq p_{j+1}$; if $q_k \geq p_1$, we let $j = 0$, and if $p_{k-1} > q_k$, we let $j = k - 1$. Now insert q_k into the list between p_j and p_{j+1} and insert the integer $(n - k + 1)$ into the permutation between a_j and a_{j+1}. When this has been done for all k, we will have a permutation $a_1\, a_2 \ldots a_n$ of $\{1, 2, \ldots, n\}$ and an n-tuple $(p_1, p_2, \ldots, p_n)$ such that $p_1 \geq p_2 \geq \cdots \geq p_n \geq 0$; and

$$p_j > p_{j+1} \qquad \text{whenever} \qquad a_j > a_{j+1}.$$

Finally, for $1 \leq j < n$, subtract one from each of $p_1, \ldots, p_j$ for each j such that $a_j > a_{j+1}$. The resulting pair $((a_1, a_2, \ldots, a_n), (p_1, p_2, \ldots, p_n))$ satisfies (15).

For example, suppose that $n = 6$ and $(q_1, \ldots, q_6) = (3, 1, 4, 0, 0, 1)$. The construction proceeds as follows:

k	$p_1 \ldots p_k$	$a_1 \ldots a_k$
1	3	6
2	3 1	6 5
3	4 3 1	4 6 5
4	4 3 1 0	4 6 5 3
5	4 3 1 0 0	4 6 5 2 3
6	4 3 1 1 0 0	4 6 1 5 2 3

The final adjustment gives $(p_1, \ldots, p_6) = (2, 1, 0, 0, 0, 0)$.

It is not hard to verify that this process is reversible; so the desired correspondence has been established, and MacMahon's theorem has been proved. We will meet with similar one-to-one correspondence algorithms in Section 5.1.4.

EXERCISES

1. [*10*] What is the inversion table for the permutation 2 7 1 8 4 5 9 3 6? What permutation has the inversion table 5 0 1 2 1 2 0 0?

2. [*M15*] The solution to Josephus' problem, as stated in exercise 1.3.2–22, is a permutation of $\{1, 2, \ldots, n\}$; the example given there ($n = 8$, $m = 4$) has the solution 5 4 6 1 3 8 7 2. The inversion table corresponding to this permutation is 3 6 3 1 0 0 1 0. Give a simple recurrence relation for the elements $b_1\, b_2 \ldots b_n$ of

the inversion table in the general Josephus problem for n men, with every mth man executed.

3. [*18*] If the permutation $a_1\, a_2 \ldots a_n$ corresponds to the inversion table $b_1\, b_2 \ldots b_n$, what is the permutation $\bar{a}_1\, \bar{a}_2 \ldots \bar{a}_n$ corresponding to the inversion table

$$(n - 1 - b_1)\, (n - 2 - b_2) \ldots (0 - b_n)?$$

▸ **4.** [*20*] Design an algorithm suitable for computer implementation which constructs the permutation $a_1\, a_2 \ldots a_n$ corresponding to a given inversion table $b_1\, b_2 \ldots b_n$ satisfying (3). [*Hint:* Consider a linked-memory technique.]

5. [*35*] The algorithm of exercise 4 requires an execution time roughly proportional to n^2 on typical computers; is it possible to design an algorithm whose running time is substantially better than order n^2?

▸ **6.** [*26*] Design an algorithm which computes the inversion table $b_1\, b_2 \ldots b_n$ corresponding to a given permutation $a_1\, a_2 \ldots a_n$ of $\{1, 2, \ldots, n\}$, where the running time is essentially proportional to $n \log n$ on typical computers.

7. [*20*] Several other kinds of inversion tables can be defined, corresponding to a given permutation $a_1\, a_2 \ldots a_n$ of $\{1, 2, \ldots, n\}$, besides the particular table $b_1\, b_2 \ldots b_n$ defined in the text; in this exercise we will consider three other types of inversion tables that arise in applications.

Let c_j be the number of inversions whose *first* component is j, that is, the number of elements to the *right* of j that are less than j. [Corresponding to (1) we have the table 0 0 0 1 4 2 1 5 7; clearly $0 \le c_j < j$.] Let $B_j = b_{a_j}$ and $C_j = c_{a_j}$.

Show that $0 \le B_j < j$ and $0 \le C_j \le n - j$, for $1 \le j \le n$; furthermore show that the permutation $a_1\, a_2 \ldots a_n$ can be determined uniquely when either $c_1\, c_2 \ldots c_n$ or $B_1\, B_2 \ldots B_n$ or $C_1\, C_2 \ldots C_n$ is given.

8. [*M24*] Continuing the notation of exercise 7, let $a'_1\, a'_2 \ldots a'_n$ be the inverse of $a_1\, a_2 \ldots a_n$, and let the corresponding inversion tables be $b'_1\, b'_2 \ldots b'_n$, $c'_1\, c'_2 \ldots c'_n$, $B'_1\, B'_2 \ldots B'_n$, and $C'_1\, C'_2 \ldots C'_n$. Find as many interesting relations between the a_j, b_j, c_j, B_j, C_j, a'_j, b'_j, c'_j, B'_j, C'_j as you can.

▸ **9.** [*M21*] Prove that, in the notation of exercise 7, the permutation $a_1\, a_2 \ldots a_n$ is its own inverse if and only if $b_j = C_j$ for $1 \le j \le n$.

10. [*HM20*] Consider Fig. 1 as a polyhedron in three dimensions. What is the diameter of the truncated octahedron (the distance between vertex 1234 and vertex 4321), if all of its edges have unit length?

11. [*M25*] (a) If $\pi = a_1\, a_2 \ldots a_n$ is a permutation of $\{1, 2, \ldots, n\}$, let $E(\pi) = \{(a_i, a_j) \mid i < j, a_i > a_j\}$ be the set of its inversions, and let

$$\overline{E}(\pi) = \{(a_i, a_j) \mid i > j, a_i > a_j\}$$

be the "non-inversions." Prove that $E(\pi)$ and $\overline{E}(\pi)$ are transitive. (A set S of ordered pairs is called *transitive* if (a, c) is in S whenever both (a, b) and (b, c) are in S.) (b) Conversely, let E be any transitive subset of $T = \{(x, y) \mid 1 \le y < x \le n\}$ whose complement $T \backslash E$ is also transitive. Prove that there exists a permutation π such that $E(\pi) = E$.

12. [*M28*] Continuing the notation of the previous exercise, prove that if π_1 and π_2 are permutations and if E is the smallest transitive set containing $E(\pi_1) \cup E(\pi_2)$,

then $\overline{E}$ is transitive. [Hence, if we say π_1 is "above" π_2 whenever $E(\pi_1) \subseteq E(\pi_2)$, a *lattice* of permutations is defined; there is a unique "lowest" permutation "above" two given permutations. Figure 1 is the lattice diagram when $n = 4$.]

13. [*M23*] It is well known that half of the terms in the expansion of a determinant have a $+$ sign, and half have a $-$ sign. In other words, there are just as many permutations with an *even* number of inversions as with an *odd* number, when $n \geq 2$. Show that, in general, the number of permutations having a number of inversions congruent to t modulo m is $n!/m$, regardless of the integer t, whenever $n \geq m$.

14. [*M24*] (F. Franklin.) A partition of n into k distinct parts is a representation $n = p_1 + p_2 + \cdots + p_k$, where $p_1 > p_2 > \cdots > p_k > 0$. For example, the partitions of 7 into distinct parts are 7, $6+1$, $5+2$, $4+3$, $4+2+1$. Let $f_k(n)$ be the number of partitions of n into k distinct parts; prove that $\sum_k (-1)^k f_k(n) = 0$, unless n has the form $(3j^2 \pm j)/2$, for some nonnegative integer j; in the latter case the sum is $(-1)^j$. For example, when $n = 7$ the sum is $-1+3-1 = 1$, since $7 = (3 \cdot 2^2 + 2)/2$. [*Hint:* Represent a partition as an array of dots, putting p_i dots in the ith row, for $1 \leq i \leq k$. Find the smallest j such that $p_{j+1} < p_j - 1$, and encircle the rightmost dots in the first j rows. If $j < p_k$, these j dots can usually be removed, tilted 45°, and placed as a new $(k+1)$ st row. On the other hand if $j \geq p_k$, the kth row of dots can usually be removed, tilted 45°, and placed to the right of the circled

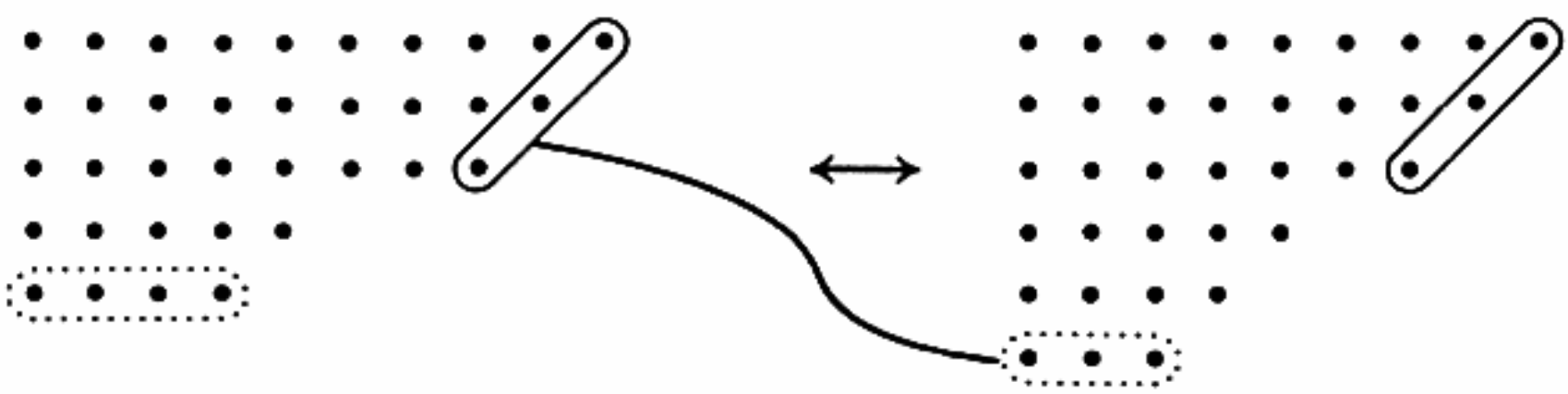


Fig. 2. Franklin's correspondence between partitions with distinct parts.

dots. (See Fig. 2.) This process pairs off partitions having an odd number of rows with partitions having an even number of rows, in most cases, so only unpaired partitions must be considered in the sum.]

Note: As a consequence, we obtain Euler's formula

$$(1-z)(1-z^2)(1-z^3)\ldots = 1 - z - z^2 + z^5 + z^7 - z^{12} - z^{15} + \cdots$$
$$= \sum_{-\infty < j < \infty} (-1)^j z^{(3j^2+j)/2}$$

Since the generating function for ordinary partitions (whose parts are not necessarily distinct) is $\sum p(n)z^n = 1/(1-z)(1-z^2)(1-z^3)\ldots$, we obtain a nonobvious recurrence relation for the partition numbers, $p(n) = p(n-1) + p(n-2) - p(n-5) - p(n-7) + p(n+12) + p(n+15) - \cdots$.

15. [*M23*] Prove that (16) is the generating function for partitions into at most n parts, i.e., prove that the coefficient of z^m in $1/(1-z)(1-z^2)\ldots(1-z^n)$ is the number of ways to write $m = p_1 + p_2 + \cdots + p_n$ with $p_1 \geq p_2 \geq \cdots \geq p_n \geq 0$.

[*Hint:* Drawing dots as in exercise 14, show that there is a one-to-one correspondence between n-tuples $(p_1, p_2, \ldots, p_n)$ such that $p_1 \geq p_2 \geq \cdots \geq p_n \geq 0$ and sequences $(P_1, P_2, P_3, \ldots)$ such that $n \geq P_1 \geq P_2 \geq P_3 \geq \cdots \geq 0$, with the property that $p_1 + p_2 + \cdots + p_n = P_1 + P_2 + P_3 + \cdots$. In other words, partitions into at most n parts correspond to partitions into parts not exceeding n.]

16. [*M25*] (L. Euler.) Prove the following identities by interpreting both sides of the equations in terms of partitions:

$$\prod_{k\geq 0} \frac{1}{(1-q^k z)} = \frac{1}{(1-z)(1-qz)(1-q^2z)\ldots}$$

$$= 1 + \frac{z}{1-q} + \frac{z^2}{(1-q)(1-q^2)} + \cdots = \sum_{n\geq 0} z^n \Big/ \prod_{1\leq k\leq n} (1-q^k).$$

$$\prod_{k\geq 0} (1+q^k z) = (1+z)(1+qz)(1+q^2z)\cdots$$

$$= 1 + \frac{z}{1-q} + \frac{z^2 q}{(1-q)(1-q^2)} + \cdots = \sum_{n\geq 0} z^n q^{n(n-1)/2} \Big/ \prod_{1\leq k\leq n} (1-q^k).$$

17. [*20*] In MacMahon's correspondence defined at the end of this section, what are the 24 quadruples (q_1, q_2, q_3, q_4) for which $(p_1, p_2, p_3, p_4) = (0, 0, 0, 0)$?

18. [*M30*] (T. Hibbard, *CACM* **6** (1963), 210.) Let $n > 0$, and assume that a sequence of 2^n n-bit integers $X_0, \ldots, X_{2^n-1}$ has been generated at random, where each bit of each number is independently equal to 1 with probability p. Consider the sequence $X_0 \oplus 0, X_1 \oplus 1, \ldots, X_{2^n-1} \oplus (2^n - 1)$, where $\oplus$ denotes the operation of taking "exclusive or" on the binary representations. Thus if $p = 0$, the sequence is $0, 1, \ldots, 2^n - 1$, and if $p = 1$ it is $2^n - 1, \ldots, 1, 0$; and when $p = \frac{1}{2}$, each element of the sequence is a random integer between 0 and $2^n - 1$. For general p this is a useful way to generate a sequence of random integers with a biased number of inversions, although the distribution of the elements of the sequence taken as a whole is uniform.

Determine the average number of inversions in such a sequence, as a function of the probability p.

19. [*M36*] (D. Foata.) Give a direct proof of MacMahon's index theorem: Find an explicit one-to-one correspondence which takes a permutation on n elements, having index k, into one having k inversions and having the same rightmost element.

20. [*M43*] The following famous identity due to Jacobi [*Fundamenta Nova Theoriæ Functionum Ellipticorum* (1829), §64] is the basis of many remarkable relationships involving elliptic functions:

$$\prod_{k\geq 1} (1-u^k v^{k-1})(1-u^{k-1}v^k)(1-u^k v^k)$$

$$= (1-u)(1-v)(1-uv)(1-u^2v)(1-uv^2)(1-u^2v^2)\cdots$$

$$= 1 - (u+v) + (u^3v + uv^3) - (u^6v^3 + u^3v^6) + \cdots$$

$$= 1 + \sum_{n\geq 1} (-1)^n (u^{(n+1)n/2} v^{(n-1)n/2} + u^{(n-1)n/2} v^{(n+1)n/2}).$$

For example, if we set $u = z$, $v = z^2$, we obtain Euler's formula of exercise 14. If we set $z = \sqrt{u/v}$, $q = \sqrt{uv}$, we obtain

$$\prod_{k\geq 1}(1 - q^{2k-1}z)(1 - q^{2k-1}z^{-1})(1 - q^{2k}) = \sum_{-\infty<n<\infty}(-1)^n z^n q^{n^2}.$$

Is there a combinatorial proof of Jacobi's identity, analogous to Franklin's proof of the special case in exercise 14? (Thus we want to consider "complex partitions"

$$m + ni = (p_1 + q_1 i) + (p_2 + q_2 i) + \cdots + (p_k + q_k i)$$

where the $p_j + q_j i$ are distinct nonzero complex numbers; p_j and q_j are nonnegative integers with $|p_j - q_j| \leq 1$. Jacobi's identity says that the number of such representations with k even is the same as the number with k odd, except when m and n are consecutive triangular numbers!) What other remarkable properties do complex partitions have?

*5.1.2. Permutations of a Multiset

So far we have been discussing permutations of a *set* of elements; this is just a special case of the concept of permutations of a *multiset*. (A multiset is like a set except that it can have repetitions of identical elements. Some basic properties of multisets have been discussed in Section 4.6.3.)

For example, consider the multiset

$$M = \{a, a, a, b, b, c, d, d, d, d\} \tag{1}$$

which contains 3 a's, 2 b's, 1 c, and 4 d's. We may also indicate the multiplicities of elements in another way, namely

$$M = \{3 \cdot a, 2 \cdot b, c, 4 \cdot d\}. \tag{2}$$

A permutation of M is an arrangement of its elements into a row, e.g.,

$$c\ a\ b\ d\ d\ a\ b\ d\ a\ d.$$

From another point of view we would call this a string of letters, containing 3 a's, 2 b's, 1 c, and 4 d's.

How many permutations of M are possible? If we regarded the elements of M as distinct, by subscripting them $a_1, a_2, a_3, b_1, b_2, c_1, d_1, d_2, d_3, d_4$, we would have $10! = 3{,}628{,}800$ permutations, but many of these would actually be the same when we removed the subscripts. In fact, each permutation of M would occur exactly $3!2!1!4! = 288$ times, since we can start with any permutation of M and put subscripts on the a's in 3! ways, on the b's (independently) in 2! ways, on the c in 1 way, and on the d's in 4! ways. Therefore the true number of permutations of M is

$$\frac{10!}{3!2!1!4!} = 12{,}600.$$

In general, we can see by this same argument that the number of permutations of any multiset is the multinomial coefficient

$$\binom{n}{n_1, n_2, \ldots} = \frac{n!}{n_1! n_2! \ldots}, \tag{3}$$

where n_1 is the number of elements of one kind, n_2 is the number of another kind, etc., and $n = n_1 + n_2 + \cdots$ is the total number of elements.

The number of permutations of a set was known in ancient times. The Hebrew Book of Creation (c. 100 A.D.), which was the earliest literary product of Jewish philosophical mysticism, gives the correct values of the first seven factorials, after which it says "Go on and obtain numbers which the mouth cannot express and the ear cannot hear." [*Sefer Yezirah*, ed. by R. Mordecai Atia (Jerusalem: Sh. Monson, 1962), verse 52 (pp. 107–108); cf. also Solomon Gandz, *Studies in Hebrew Astronomy and Mathematics* (New York: Ktav, 1970), 494–496. The Book of Creation was based on supposedly significant relations between the seven planets, the seven consonants with a double pronunciation, the seven apertures in a human head, and the seven days of creation.] This is the first known enumeration of permutations in history. The second occurs in the Indian classic *Anuyogadvarā-sutra* (c. 500 A.D.), rule 97, which gives the formula

$$6 \times 5 \times 4 \times 3 \times 2 \times 1 - 2$$

for the number of permutations of six elements which are neither in ascending nor descending order. [See G. Chakravarti, *Bull. Calcutta Math. Soc.* **24** (1932), 79–88. The *Anuyogadvarā-sutra* is one of the books in the canon of Jainism, a religious sect which flourishes in India.]

The corresponding rule for multisets seems to have appeared first in the *Líláváti* of Bháscara Áchárya (c. 1150), sections 270–271. Bháscara stated the rule rather tersely, and illustrated it only with two simple examples {2, 2, 1, 1} and {4, 8, 5, 5, 5}. Consequently the English translations of his work do not all state the rule correctly, although there is little doubt that Bháscara knew what he was talking about. He went on to give the interesting formula

$$\frac{(4 + 8 + 5 + 5 + 5) \times 120 \times 11111}{5 \times 6}$$

for the sum of the 20 numbers $48555 + 45885 + \cdots$.

The correct rule for counting permutations when only one element is repeated was found independently by the German Jesuit scholar Athanasius Kircher in his voluminous treatise on music [*Musurgia Universalis* **2** (Rome, 1650), 5–7]. Kircher was interested in the number of tunes that could be made from a given collection of notes, so he devised what he called "musarithmetic." On pp. 18–21 of his treatise he correctly gave the number of permutations of the multiset $\{m \cdot C, n \cdot D\}$ for several values of m and n, although he didn't reveal his method of calculation except when $n = 1$.

The general rule (3) then appeared in Jean Prestet's *Elémens de Mathématiques* (Paris, 1675), 351–352, one of the very first expositions of combinatorial mathematics to be written in the Western world. Prestet stated the rule correctly for a general multiset, but illustrated it only in the simple case $\{a, a, b, b, c, c\}$. He specifically pointed out that division by the *sum* of the factorials, which he considered to be the natural generalization of Kircher's rule, did not work properly. A few years later, John Wallis's *Treatise of Algebra* **2** (Oxford, 1685), 117–118 gave a clearer and somewhat more detailed discussion of the rule.

In 1965, Dominique Foata introduced an ingenious idea called the "intercalation product" which makes it possible to extend many of the known results about ordinary permutations to the general case of multiset permutations. [See *Publ. Inst. Statistique*, Univ. Paris, **14** (1965, 81–241; also *Lecture Notes in Math.* **85** (Springer, 1969).] Assuming that the elements of the multiset are linearly ordered in some way, we may consider the *two-line notation*, e.g.,

$$\begin{pmatrix} a & a & a & b & b & c & d & d & d & d \\ c & a & b & d & d & a & b & d & a & d \end{pmatrix}, \tag{4}$$

where the top line contains the elements of M sorted into nondecreasing order and the bottom line is the permutation itself. The *intercalation product* $\alpha \intercal \beta$ of two multiset permutations α and β is obtained by (a) expressing α and β in the two-line notation, (b) juxtaposing these two-line representations, and (c) sorting the columns into nondecreasing order of the top line. The sorting is to be "stable" in the sense that left-to-right order of elements in the bottom line is preserved when the corresponding top line elements are equal. For example, $c\ a\ d\ a\ b \intercal b\ d\ d\ a\ d = c\ a\ b\ d\ d\ a\ b\ d\ a\ d$, since

$$\begin{pmatrix} a & a & b & c & d \\ c & a & d & a & b \end{pmatrix} \intercal \begin{pmatrix} a & b & d & d & d \\ b & d & d & a & d \end{pmatrix} = \begin{pmatrix} a & a & a & b & b & c & d & d & d & d \\ c & a & b & d & d & a & b & d & a & d \end{pmatrix}. \tag{5}$$

It is easy to see that the intercalation product operation is associative, i.e.,

$$\alpha \intercal (\beta \intercal \gamma) = \alpha \intercal (\beta \intercal \gamma), \tag{6}$$

and that it also satisfies the cancellation laws

$$\begin{aligned} \pi \intercal \alpha = \pi \intercal \beta \quad &\text{implies} \quad \alpha = \beta, \\ \alpha \intercal \pi = \beta \intercal \pi \quad &\text{implies} \quad \alpha = \beta. \end{aligned} \tag{7}$$

There is an "identity element"

$$\alpha \intercal \epsilon = \epsilon \intercal \alpha = \alpha, \tag{8}$$

where ϵ is the null permutation, the "arrangement" of the empty set. The commutative law is not valid in general (see exercise 2), but we do have

$$\alpha \intercal \beta = \beta \intercal \alpha \qquad \text{if } \alpha \text{ and } \beta \text{ have no letters in common.} \tag{9}$$

In an analogous fashion we can extend the concept of *cycles* to cases where elements are repeated; we let

$$(x_1 \quad x_2 \quad \ldots \quad x_n) \tag{10}$$

stand for the permutation obtained in two-line form by sorting the columns of

$$\begin{pmatrix} x_1 & x_2 & \ldots & x_n \\ x_2 & x_3 & \ldots & x_1 \end{pmatrix} \tag{11}$$

by their top elements in a stable manner. For example,

$$(d\ b\ d\ d\ a\ c\ a\ a\ b\ d) = \begin{pmatrix} d & b & d & d & a & c & a & a & b & d \\ b & d & d & a & c & a & a & b & d & d \end{pmatrix} = \begin{pmatrix} a & a & a & b & b & c & d & d & d & d \\ c & a & b & d & d & a & b & d & a & d \end{pmatrix},$$

so the permutation (4) is actually a cycle. We might render this cycle in words by saying something like "d goes to b goes to d goes to d goes . . . goes to d goes back." Note that these general cycles do not share all of the properties of ordinary cycles; $(x_1\ x_2\ \ldots\ x_n)$ is not always the same as $(x_2\ \ldots\ x_n\ x_1)$.

We observed in Section 1.3.3 that every permutation of a set has a unique representation (up to order) as a product of disjoint cycles, where the "product" of permutations is defined by a law of composition. It is easy to see that *the product of disjoint cycles is exactly the same as their intercalation;* this suggests that we might be able to generalize the previous results, obtaining a unique representation (in some sense) for any permutation of a multiset, as the intercalation of cycles. In fact there are at least two natural ways to do this, each of which has important applications.

Equation (5) shows one way to factor $c\ a\ b\ d\ d\ a\ b\ d\ a\ d$ as the intercalation of shorter permutations; let us consider the general problem of finding all factorizations $\pi = \alpha \mathbin{⫲} \beta$ of a given permutation π. It will be helpful to consider a particular permutation, such as

$$\pi = \begin{pmatrix} a & a & b & b & b & b & b & c & c & c & d & d & d & d & d \\ d & b & c & b & c & a & c & d & a & d & d & b & b & b & d \end{pmatrix}, \tag{12}$$

as we investigate this factorization problem.

If we can write this permutation π in the form $\alpha \mathbin{⫲} \beta$, where α contains the letter a at least once, then the leftmost a in the top line of the two-line notation for α must appear over the letter d, so α must also contain at least one occurrence of the letter d. If we now look at the leftmost d in the top line of α, we see in the same way that it must appear over the letter d, so α must contain at least *two* d's. Looking at the second d, we see that α also contains a b. We have the partial result

$$\alpha = \begin{pmatrix} a & \ldots & b & \ldots & d\ d & \ldots \\ d & & & & d\ b & \end{pmatrix} \tag{13}$$

on the sole assumption that α is a left factor of π which contains the letter a. Proceeding in the same manner, we find that the b in the top line of (13) must

appear over the letter c, etc. Eventually this process will reach the letter a again, and we can identify this a with the first a if we choose to do so. The argument we have just made essentially proves that any left factor α of (12) which contains the letter a has the form $(d\ d\ b\ c\ d\ b\ b\ c\ a) \intercal \alpha'$, for some permutation α'. (It is convenient to write the a last in the cycle, instead of first; this is permissible since there is only one a.) Similarly, if we had assumed that α contains the letter b, we would have deduced that $\alpha = (c\ d\ d\ b) \intercal \alpha''$ for some α''.

In general, this argument shows that, *if we have any factorization* $\alpha \intercal \beta = \pi$, *where* α *contains a given letter* y, *there is a unique cycle of the form*

$$(x_1 \ldots x_n\ y), \qquad n \geq 0, \qquad x_1, \ldots, x_n \neq y, \tag{14}$$

which is a left factor of α. This cycle is easily determined when π and y are given; it is the shortest left factor of π which contains the letter y. One of the consequences of this observation is the following theorem:

Theorem A. *Let the elements of the multiset* M *be linearly ordered by the relation "*$<$*". Every permutation* π *of* M *has a unique representation as the intercalation*

$$\pi = (x_{11} \ldots x_{1n_1}y_1) \intercal (x_{21} \ldots x_{2n_2}y_2) \intercal \cdots \intercal (x_{t1} \ldots x_{tn_t}y_t), \qquad t \geq 0, \tag{15}$$

where the following two conditions are satisfied:

$$y_1 \leq y_2 \leq \cdots \leq y_t;$$

$$y_i < x_{ij} \qquad \text{for} \qquad 1 \leq j \leq n_i, \qquad 1 \leq i \leq t. \tag{16}$$

(In other words the last element in each cycle is smaller than every other element, and the sequence of last elements is in nondecreasing order.)

Proof. If $\pi = \epsilon$, we obtain such a factorization by letting $t = 0$. Otherwise we let y_1 be the smallest element permuted; and we determine $(x_{11} \ldots x_{1n_1}y_1)$, the shortest left factor of π containing y_1, as in the example above. Now $\pi = (x_{11} \ldots x_{1n_1}y_1) \intercal \rho$ for some permutation ρ; by induction on the length, we can write

$$\rho = (x_{21} \ldots x_{2n_2}y_2) \intercal \cdots \intercal (x_{t1} \ldots x_{tn_t}y_t), \qquad t \geq 1,$$

where (16) is satisfied. This proves the existence of such a factorization.

Conversely, to prove that the representation (15) satisfying (16) is unique, clearly $t = 0$ if and only if π is the null permutation ϵ. When $t > 0$, (16) implies that y_1 is the smallest element permuted, and that $(x_{11} \ldots x_{1n_1}y_1)$ is the shortest left factor containing y_1. Therefore $(x_{11} \ldots x_{1n_1}y_1)$ is uniquely determined; by the cancellation law (7) and induction, the representation is unique. ∎

For example, the "canonical" factorization of (12), satisfying the given conditions, is

$$(d\ d\ b\ c\ d\ b\ b\ c\ a) \intercal (b\ a) \intercal (c\ d\ b) \intercal (d), \tag{17}$$

if $a < b < c < d$.

It is important to note that we can actually drop the parentheses and the $\top$'s in this representation, without ambiguity! Each cycle ends just after the first appearance of the smallest remaining element. So this construction associates the permutation

$$\pi' = d\ d\ b\ c\ d\ b\ b\ c\ a\ b\ a\ c\ d\ b\ d$$

with the original permutation

$$\pi = d\ b\ c\ b\ c\ a\ c\ d\ a\ d\ d\ b\ b\ b\ d.$$

Whenever the two-line representation of π had a column of the form $\binom{y}{x}$, where $x < y$, the associated permutation has a corresponding pair of adjacent elements $\ldots yx \ldots$. Thus our example permutation π has three columns of the form $\binom{d}{b}$, and π' has three occurrences of the pair db. In general, this construction establishes the following remarkable theorem:

Theorem B. *Let M be a multiset. There is a one-to-one correspondence between the permutations of M such that, if π corresponds to π', the following conditions hold:*

a) *The leftmost element of π' equals the leftmost element of π;*

b) *For all pairs of permuted elements (x, y) with $x < y$, the number of occurrences of the column $\binom{y}{x}$ in the two-line notation of π is equal to the number of times x is immediately preceded by y in π'.* ∎

When M is a set, this is essentially the same as the "unusual correspondence" we discussed near the end of Section 1.3.3, with unimportant changes. The more general result in Theorem B is quite useful for enumerating permutations of special types, since we can often solve a problem based on a two-line constraint more easily than the equivalent problem based on an adjacent-pair constraint.

P. A. MacMahon considered problems of this type in his extraordinary book *Combinatory Analysis* **1** (Cambridge Univ. Press, 1915), 168–186. He gave a constructive proof of Theorem B in the special case that M contains only two different kinds of elements, say a and b; his construction for this case is essentially the same as that given here, although he expressed it quite differently. For the case of three different elements a, b, c, MacMahon gave a complicated nonconstructive proof of Theorem B; the general case was first proved by Foata in 1965.

As a nontrivial example of Theorem B, let us find the number of strings of letters a, b, c containing exactly

$$\begin{array}{ll} A & \text{occurrences of the letter } a; \\ B & \text{occurrences of the letter } b; \\ C & \text{occurrences of the letter } c; \\ k & \text{occurrences of the adjacent pair of letters } ca; \\ l & \text{occurrences of the adjacent pair of letters } cb; \\ m & \text{occurrences of the adjacent pair of letters } ba. \end{array} \tag{18}$$

The theorem tells us this is the same as the number of two-line arrays of the form

$$\begin{array}{c} \overbrace{}^{A} \quad \overbrace{}^{B} \quad \overbrace{}^{C} \\ \begin{pmatrix} a & \ldots & a & b & \ldots & b & c & \ldots & c \\ \sqcup & \ldots & \sqcup & \sqcup & \ldots & \sqcup & \sqcup & \ldots & \sqcup \end{pmatrix} \\ \underbrace{}_{A\text{-}k\text{-}m\ a\text{'s}} \quad \underbrace{}_{m\ a\text{'s}} \quad \underbrace{}_{k\ a\text{'s}} \\ \underbrace{}_{B\text{-}l\ b\text{'s}} \quad \underbrace{}_{l\ b\text{'s}} \\ \underbrace{}_{C\ c\text{'s}} \end{array} \tag{19}$$

The a's can be placed in the second line in

$$\binom{A}{A-k-m}\binom{B}{m}\binom{C}{k} \quad \text{ways;}$$

then the b's can be placed in the remaining positions in

$$\binom{B+k}{B-l}\binom{C-k}{l} \quad \text{ways.}$$

The positions which are still vacant must be filled by c's; hence the desired number is

$$\binom{A}{A-k-m}\binom{B}{m}\binom{C}{k}\binom{B+k}{B-l}\binom{C-k}{l}. \tag{20}$$

Let us return to the question of finding all factorizations of a given permutation. Is there such a thing as a "prime" permutation, one which has no intercalation factors except itself and ϵ? The discussion preceding Theorem A leads us quickly to conclude that *a permutation is prime if and only if it is a cycle with no repeated elements.* For if it is such a cycle, our argument proves that there are no left factors except ϵ and the cycle itself. And if a permutation contains a repeated element y, it has a nontrivial cyclic left factor in which y appears only once.

If a permutation is not prime, we can factor it further and further into smaller and smaller pieces until it has been expressed as a product of primes. Furthermore we can show that the factorization is unique, if we neglect the order of factors that commute:

Theorem C. *Every permutation of a multiset can be written as a product*

$$\sigma_1 \intercal \sigma_2 \intercal \cdots \intercal \sigma_t, \qquad t \geq 0, \tag{21}$$

where each σ_j is a cycle having no repeated elements. This representation is unique, in the sense that any two such representations of the same permutation may be transformed into each other by successively interchanging pairs of adjacent disjoint cycles.

The term "disjoint cycles" means cycles having no elements in common. As an example of this theorem, we can verify that the permutation

$$\begin{pmatrix} a & a & b & b & c & c & d \\ b & a & a & c & d & b & c \end{pmatrix}$$

has exactly five factorizations into primes, namely

$$\begin{aligned}(a\ b)_{\mathsf{T}}(a)_{\mathsf{T}}(c\ d)_{\mathsf{T}}(b\ c) &= (a\ b)_{\mathsf{T}}(c\ d)_{\mathsf{T}}(a)_{\mathsf{T}}(b\ c)\\ &= (a\ b)_{\mathsf{T}}(c\ d)_{\mathsf{T}}(b\ c)_{\mathsf{T}}(a)\\ &= (c\ d)_{\mathsf{T}}(a\ b)_{\mathsf{T}}(a)_{\mathsf{T}}(b\ c)\\ &= (c\ d)_{\mathsf{T}}(a\ b)_{\mathsf{T}}(b\ c)_{\mathsf{T}}(a). \qquad (22)\end{aligned}$$

Proof. We must show that the stated uniqueness property holds. By induction on the length of the permutation, it suffices to prove that if ρ and σ are unequal cycles having no repeated elements, and if

$$\rho_{\mathsf{T}}\alpha = \sigma_{\mathsf{T}}\beta,$$

then ρ and σ are disjoint, and

$$\alpha = \sigma_{\mathsf{T}}\theta, \qquad \beta = \rho_{\mathsf{T}}\theta,$$

for some permutation θ.

If y is any element of the cycle ρ, then any left factor of $\sigma_{\mathsf{T}}\beta$ containing the element y must have ρ as a left factor. So if ρ and σ have an element in common, σ is a multiple of ρ; hence $\sigma = \rho$ (since they are primes), contradicting our assumption. Therefore the cycle containing y, having no elements in common with σ, must be a left factor of β. The proof is completed by using the cancellation law (7). ∎

As an example of Theorem C, let us consider permutations of the multiset $M = \{A \cdot a, B \cdot b, C \cdot c\}$ consisting of A a's, B b's, and C c's. Let $N(A, B, C, m)$ be the number of permutations of M whose two-line representation contains *no* columns of the forms $\binom{a}{a}, \binom{b}{b}, \binom{c}{c}$, and exactly m columns of the form $\binom{a}{b}$. It follows that there are exactly $A - m$ columns of the form $\binom{a}{c}$, $B - m$ of the form $\binom{c}{b}$, $C - B + m$ of the form $\binom{c}{a}$, $C - A + m$ of the form $\binom{b}{c}$, and $A + B - C - m$ of the form $\binom{b}{a}$, hence

$$N(A, B, C, m) = \binom{A}{m}\binom{B}{C - A + m}\binom{C}{B - m}. \qquad (23)$$

Theorem C tells us that we can count these permutations in another way: Since columns of the forms $\binom{a}{a}, \binom{b}{b}, \binom{c}{c}$ are excluded, the only possible prime factors of the permutation are

$$(a\ b), \qquad (a\ c), \qquad (b\ c), \qquad (a\ b\ c), \qquad (a\ c\ b). \qquad (24)$$

Each pair of these cycles has at least one letter in common, so the factorization

into primes is completely unique. If the cycle $(a\ b\ c)$ occurs k times in the factorization, our previous assumptions imply that $(a\ b)$ occurs $m - k$ times, $(b\ c)$ occurs $C - A + m - k$ times, $(a\ c)$ occurs $C - B + m - k$ times, and $(a\ c\ b)$ occurs $A + B - C - 2m + k$ times. Hence $N(A, B, C, m)$ is the number of permutations of these cycles (a multinomial coefficient), summed over k:

$$\begin{aligned} &N(A, B, C, m) \\ &\quad = \sum_k \frac{(C+m-k)!}{(m-k)!(C-A+m-k)!(C-B+m-k)!k!(A+B-C-2m+k)!} \\ &\quad = \sum_k \binom{m}{k}\binom{A}{m}\binom{A-m}{C-B+m-k}\binom{C+m-k}{A}. \end{aligned} \tag{25}$$

Comparing this with (23), we find that the following identity must be valid:

$$\sum_k \binom{m}{k}\binom{A-m}{C-B+m-k}\binom{C+m-k}{A} = \binom{B}{C-A+m}\binom{C}{B-m}. \tag{26}$$

This turns out to be the identity we met in exercise 1.2.6–31, namely

$$\sum_j \binom{M-R+S}{j}\binom{N+R-S}{N-j}\binom{R+j}{M+N} = \binom{R}{M}\binom{S}{N}, \tag{27}$$

with $M = A + B - C - m$, $N = C - B + m$, $R = B$, $S = C$, and $j = C - B + m - k$.

Similarly we can count the number of permutations of $\{A \cdot a, B \cdot b, C \cdot c, D \cdot d\}$ such that the number of columns of various types is specified as follows:

Column type:	a d	a b	b a	b c	c b	c d	d a	d c
Number of times:	r	$A-r$	q	$B-q$	$B-A+r$	$D-r$	$A-q$	$D-A+q$

(28)

(Here $A + C = B + D$.) The possible cycles occurring in a prime factorization of such permutations are then

Cycle:	$(a\ b)$	$(b\ c)$	$(c\ d)$	$(d\ a)$	$(a\ b\ c\ d)$	$(d\ c\ b\ a)$
Number of times:	$A-r-s$	$B-q-s$	$D-r-s$	$A-q-s$	s	$q-A+r+s$

(29)

for some s (see exercise 12). In this case the cycles $(a\ b)$ and $(c\ d)$ commute with each other, and so do $(b\ c)$ and $(d\ a)$, so we must count the number of distinct prime factorizations. It turns out (see exercise 10) that there is always a unique factorization such that no $(c\ d)$ is immediately followed by $(a\ b)$, and

no $(d\ a)$ is immediately followed by $(b\ c)$. Hence by the result of exercise 13, we have the identity

$$\sum_{s,t}\binom{B}{t}\binom{A-q-s}{A-r-s-t}\binom{B+D-r-s-t}{B-q-s}\times\frac{D!}{(D-r-s)!(A-q-s)!s!(q-A+r+s)!}=\binom{A}{r}\binom{B+D-A}{D-r}\binom{B}{q}\binom{D}{A-q}$$

Taking out the factor $\binom{D}{A-q}$ from both sides and simplifying the factorials slightly leaves us with the complicated-looking five-parameter binomial coefficient identity

$$\sum_{s,t}\binom{B}{t}\binom{A-r-t}{s}\binom{B+D-r-s-t}{D+q-r-t}\times\binom{D-A+q}{D-r-s}\binom{A-q}{r+t-q}=\binom{A}{r}\binom{B+D-A}{D-r}\binom{B}{q}. \quad (30)$$

The sum on s can be performed using (27), and the resulting sum on t is easily evaluated; so, after all this work, we were not fortunate enough to discover any identities that we didn't already know how to derive. But at least we have learned how to count certain kinds of permutations, in two different ways, and these counting techniques are good training for the problems that lie ahead.

EXERCISES

1. [*M05*] *True or false:* Let M_1 and M_2 be multisets. If α is a permutation of M_1 and β is a permutation of M_2, then $\alpha \intercal \beta$ is a permutation of $M_1 \cup M_2$.

2. [*10*] The intercalation of $c\ a\ d\ a\ b$ and $b\ d\ d\ a\ d$ is computed in (5); find the intercalation $b\ d\ d\ a\ d \intercal c\ a\ d\ a\ b$ which is obtained when the factors are interchanged.

3. [*M13*] Is the converse of (9) valid? In other words, if α and β commute under intercalation, must they have no letters in common?

4. [*M11*] The canonical factorization of (12), in the sense of Theorem A, is given in (17) when $a < b < c < d$. Find the corresponding canonical factorization when $d < c < b < a$.

5. [*M23*] Condition (b) of Theorem B requires $x < y$; what would happen if we weakened the relation to $x \leq y$?

6. [*M15*] How many strings are there which contain exactly m a's, n b's, and no other letters, with exactly k of the a's preceded immediately by a b?

7. [*M21*] How many strings on the letters a, b, c satisfying conditions (18) begin with the letter a? with the letter b? with c?

▸ **8.** [*20*] Find all factorizations of (12) into two factors $\alpha \intercal \beta$.

9. [*33*] Write computer programs which perform the factorizations of a given multiset permutation into the forms mentioned in Theorems A and C.

▸ **10.** [*M30*] *True or false:* Although the factorization into primes isn't quite unique, according to Theorem C, we can insure uniqueness in the following way: "There is a linear ordering $\prec$ of the set of primes such that every permutation of a multiset has a unique factorization $\sigma_1 \mathbin{⫿} \sigma_2 \mathbin{⫿} \cdots \mathbin{⫿} \sigma_n$ into primes subject to the condition that $\sigma_i \preceq \sigma_{i+1}$ whenever σ_i commutes with σ_{i+1}, for $1 \le i < n$."

11. [*M26*] Let $\sigma_1, \sigma_2, \ldots, \sigma_t$ be cycles without repeated elements. Define a partial ordering $\prec$ on the t objects $\{x_1, \ldots, x_t\}$ by saying that $x_i \prec x_j$ if $i < j$ and σ_i has at least one letter in common with σ_j. Prove the following connection between Theorem C and the notion of "topological sorting" (Section 2.2.3): *The number of distinct prime factorizations of* $\sigma_1 \mathbin{⫿} \sigma_2 \mathbin{⫿} \cdots \mathbin{⫿} \sigma_t$ *is the number of ways to sort the given partial ordering topologically.* (For example, corresponding to (22) we find that there are five ways to sort the ordering $x_1 \prec x_3$, $x_2 \prec x_4$, $x_1 \prec x_4$ topologically.) Conversely, given any partial ordering on t elements, there is a set of cycles

$$\{\sigma_1, \sigma_2, \ldots, \sigma_t\}$$

which defines that partial ordering in the stated way.

12. [*M16*] Show that (29) is a consequence of the assumptions of (28).

13. [*M21*] Prove that the number of permutations of $\{A \cdot a, B \cdot b, C \cdot c, D \cdot d, E \cdot e, F \cdot f\}$ containing no occurrences of the adjacent pairs of letters ca and db is

$$\sum_t \binom{D}{A-t}\binom{A+B+E+F}{t}\binom{A+B+C+E+F-t}{B}\binom{C+D+E+F}{C,D,E,F}.$$

14. [*M30*] One way to define the "inverse" π^{-1} of a general permutation π, suggested by other definitions in this section, is to interchange the lines of the two-line representation of π and then to do a stable sort of the columns in order to bring the top row into nondecreasing order. For example, if $a < b < c < d$, this definition implies that $c\ a\ b\ d\ d\ a\ b\ d\ a\ d^{-1} = a\ c\ d\ a\ d\ a\ b\ b\ d\ d$.

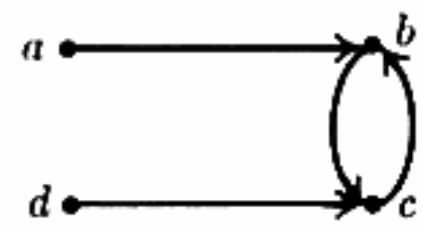
Explore properties of this inversion operation; for example, does it have any simple relation with intercalation products? Can we count the number of permutations such that $\pi = \pi^{-1}$?

▸ **15.** [*M25*] Prove that the permutation $a_1 \ldots a_n$ of the multiset

$$\{n_1 \cdot x_1, n_2 \cdot x_2, \ldots, n_m \cdot x_m\},$$

where $x_1 < x_2 < \cdots < x_n$ and $n_1 + n_2 + \cdots + n_m = m$, is a cycle if and only if the directed graph with vertices $\{x_1, x_2, \ldots, x_m\}$ and arcs from x_j to $a_{n_1+\cdots+n_j}$ contains precisely one oriented cycle. In the latter case, the number of ways to represent the permutation in cycle form is the length of the oriented cycle. For example, the directed graph corresponding to

$$\begin{pmatrix} a & a & a & b & b & c & c & c & d & d \\ d & c & b & a & c & a & a & b & d & c \end{pmatrix}$$ is



and the two ways to represent the permutation as a cycle are $(b\ a\ d\ d\ c\ a\ c\ a\ b\ c)$ and $(c\ a\ d\ d\ c\ a\ c\ b\ a\ b)$.

16. [*M35*] We found the generating function for *inversions* of permutations in the previous section, Eq. (8), in the special case that a set was being permuted. Show that, in general, if a *multiset* is permuted, the generating function for inversions is the "z-multinomial coefficient"

$$\binom{n}{n_1, n_2, \ldots}_z = \frac{n!_z}{n_1!_z n_2!_z \ldots}, \qquad \text{where} \qquad m!_z = (1 - z)(1 - z^2) \ldots (1 - z^m).$$

[Cf. (3); z-binomial coefficients were defined in Eq. 1.2.6–37.]

17. [*M24*] Find the average and standard deviation of the number of inversions in a random permutation of a given multiset, using the generating function found in exercise 16.

18. [*M30*] (P. A MacMahon.) The *index* of a permutation $a_1 a_2 \ldots a_n$ was defined in the previous section; and we proved that the number of permutations of a given set which have a given index k is the same as the number of permutations which have k inversions. Does the same result hold for permutations of a given multiset?

19. [*HM28*] Define the *Möbius function* $\mu(\pi)$ of a permutation π to be 0 if π contains repeated elements, otherwise $(-1)^k$ if π is the product of k primes. (Compare with the definition of the ordinary Möbius function, exercise 4.5.2–10.) (a) Prove that if $\pi \neq \epsilon$, we have

$$\sum \mu(\lambda) = 0,$$

summed over all permutations λ which are left factors of π (i.e., $\lambda \tau \rho = \pi$ for some ρ). (b) Given that $x_1 < x_2 < \cdots < x_m$ and $\pi = x_{i_1} x_{i_2} \cdots x_{i_n}$, where $1 \leq i_j \leq m$ for $1 \leq j \leq n$, prove that

$$\mu(\pi) = (-1)^n \epsilon(i_1\, i_2 \ldots i_n), \qquad \text{where} \qquad \epsilon(i_1\, i_2 \ldots i_n) = \operatorname{sign} \prod_{1 \leq j < k \leq n} (i_k - i_j).$$

20. [*HM33*] (D. Foata.) Let (a_{ij}) be any matrix of real numbers. In the notation of the previous exercise, part (b), define $\nu(\pi) = a_{i_1 j_1} \ldots a_{i_n j_n}$, where the two-line notation for π is

$$\begin{pmatrix} x_{i_1} & x_{i_2} & \ldots & x_{i_n} \\ x_{j_1} & x_{j_2} & \ldots & x_{j_n} \end{pmatrix}.$$

This function is useful in the computation of generating functions for permutations of a multiset, because $\sum \nu(\pi)$, summed over all permutations π of the multiset

$$\{n_1 \cdot x_1, \ldots, n_m \cdot x_m\},$$

will be the generating function for the number of permutations satisfying certain restrictions. For example, if we take $a_{ij} = z$ for $i = j$, and $a_{ij} = 1$ for $i \neq j$, then $\sum \nu(\pi)$ is the generating function for the number of "fixed points" (columns in which the top and bottom entries are equal). In order to study $\sum \nu(\pi)$ for all multisets simultaneously, we consider the function

$$G = \sum \pi \nu(\pi)$$

summed over all π in the set $\{x_1, \ldots, x_m\}^*$ of all permutations of multisets involving the elements $x_1, \ldots, x_m$, and we look at the coefficient of $x_1^{n_1} \ldots x_m^{n_m}$ in G.

In this formula for G we are treating π as the product of the x's. For example, when $m = 2$ we have

$$\begin{aligned} G &= 1 + x_1\nu(x_1) + x_2\nu(x_2) + x_1x_1\nu(x_1x_1) + x_1x_2\nu(x_1x_2) + x_2x_1\nu(x_2x_1) \\ &\quad + x_2x_2\nu(x_2x_2) + \cdots \\ &= 1 + x_1a_{11} + x_2a_{22} + x_1^2a_{11}^2 + x_1x_2a_{11}a_{22} + x_1x_2a_{21}a_{12} + x_2^2a_{22}^2 + \cdots . \end{aligned}$$

Thus the coefficient of $x_1^{n_1} \ldots x_m^{n_m}$ in G is $\sum\nu(\pi)$ summed over all permutations π of $\{n_1 \cdot x_1, \ldots, n_m \cdot x_m\}$. It is not hard to see that this coefficient is also the coefficient of $x_1^{n_1} \ldots x_m^{n_m}$ in the expression

$$(a_{11}x_1 + \cdots + a_{1m}x_m)^{n_1}(a_{21}x_1 + \cdots + a_{2m}x_m)^{n_2} \cdots (a_{m1}x_1 + \cdots + a_{mm}x_m)^{n_m}.$$

The purpose of this exercise is to prove what P. A. MacMahon called a "Master Theorem" in his *Combinatory Analysis* 1 (1915), Section 3, namely the formula

$$G = 1/D, \qquad \text{where} \qquad D = \det\begin{pmatrix} 1 - a_{11}x_1 & -a_{12}x_2 & \ldots & -a_{1m}x_m \\ -a_{21}x_1 & 1 - a_{22}x_2 & & -a_{2m}x_m \\ \vdots & & & \vdots \\ -a_{m1}x_1 & -a_{m2}x_2 & \ldots & 1 - a_{mn}x_m \end{pmatrix}.$$

For example, if $a_{ij} = 1$ for all i and j, this formula gives

$$G = 1/(1 - (x_1 + x_2 + \cdots + x_m)),$$

and the coefficient of $x_1^{n_1} \ldots x_m^{n_m}$ turns out to be $(n_1 + \cdots + n_m)!/n_1! \ldots n_m!$ as it should.

To prove the Master Theorem, show that (a) $\nu(\pi \top \rho) = \nu(\pi)\nu(\rho)$; (b) $D = \sum\pi\mu(\pi)\nu(\pi)$, in the notation of exercise 19, summed over all permutations π in

$$\{x_1, \ldots, x_m\}^*;$$

and (c) therefore $D \cdot G = 1$.

*5.1.3. Runs

In Section 3.3.2, we considered "runs up" in a permutation, as one method of testing the randomness of a sequence. If we place a vertical line at both ends of a permutation $a_1\, a_2 \ldots a_n$ and also between a_j and a_{j+1} whenever $a_j > a_{j+1}$, the *runs* are the segments between pairs of lines. For example, the permutation

$$|3\ 5\ 7|1\ 6\ 8\ 9|4|2|$$

has four runs. We determined the average number of runs of length k in a random permutation of $\{1, 2, \ldots, n\}$, as well as the covariance of the numbers of runs of lengths j and k. Runs are important in the study of sorting algorithms, because they represent sorted segments of the data, so we will now take up the subject of runs once again.

Let us use the notation

$$\left\langle {n \atop k} \right\rangle \tag{1}$$

to stand for the number of permutations of $\{1, 2, \ldots, n\}$ which have exactly k ascending runs. These numbers $\left\langle {n \atop k} \right\rangle$ arise in several contexts, and they are usually called *Eulerian numbers* since Euler discussed them in his famous book *Institutiones calculi differentialis* (St. Petersburg, 1755), 485–487 [Euler, *Opera Omnia* (1) **10** (1913), 373–375]; they should be distinguished from the so-called "Euler numbers" which are discussed in exercise 5.1.4–22.

We can use any given permutation on $\{1, \ldots, n-1\}$ to form n new permutations, by inserting the element n in all possible places. If the original permutation has k runs, exactly k of these new permutations will have k runs; the remaining $n-k$ will have $k+1$, since we increase the number of runs unless we place the element n at the end of an existing run. For example, the six permutations formed from 3 1 2 4 5 are

$$\begin{array}{lll} 6\ 3\ 1\ 2\ 4\ 5, & 3\ 6\ 1\ 2\ 4\ 5, & 3\ 1\ 6\ 2\ 4\ 5, \\ 3\ 1\ 2\ 6\ 4\ 5, & 3\ 1\ 2\ 4\ 6\ 5, & 3\ 1\ 2\ 4\ 5\ 6; \end{array}$$

all but the second and last of these have three runs instead of two. Therefore we have the recurrence relation

$$\left\langle {n \atop k} \right\rangle = k\left\langle {n-1 \atop k} \right\rangle + (n-k+1)\left\langle {n-1 \atop k-1} \right\rangle, \quad \text{integer } n \geq 1; \text{ integer } k. \tag{2}$$

By convention we will set

$$\left\langle {0 \atop k} \right\rangle = \delta_{1k}, \tag{3}$$

saying that the null permutation has one run. The reader may find it interesting to compare (2) with recurrence relations for Stirling numbers (Eqs. 1.2.6–42). Table 1 on page 37 lists the Eulerian numbers for small n.

Several patterns can be observed in Table 1. By definition, we have

$$\left\langle {n \atop 0} \right\rangle + \left\langle {n \atop 1} \right\rangle + \cdots + \left\langle {n \atop n} \right\rangle = n!; \tag{4}$$

$$\left\langle {n \atop 0} \right\rangle = 0, \qquad \left\langle {n \atop 1} \right\rangle = 1; \tag{5}$$

$$\left\langle {n \atop n} \right\rangle = 1, \qquad \text{for} \qquad n \geq 1. \tag{6}$$

There is also a rule of symmetry,

$$\left\langle {n \atop k} \right\rangle = \left\langle {n \atop n+1-k} \right\rangle, \qquad n \geq 1, \tag{7}$$

which comes from the fact that each permutation $a_1\, a_2 \ldots a_n$ having k runs corresponds to a permutation $a_n \ldots a_2 a_1$ having $n+1-k$ runs.

Another important property of the Eulerian numbers is the formula

$$\sum_k \left\langle {n \atop k} \right\rangle \binom{m+k-1}{n} = m^n, \qquad n \geq 0, \tag{8}$$

which we can prove by using the notion of sorting: Consider the m^n sequences $a_1\, a_2 \ldots a_n$ such that $1 \leq a_i \leq m$. We can sort any such sequence into non-decreasing order in a stable manner, obtaining the relation

$$a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}, \tag{9}$$

where $i_1\, i_2 \ldots i_n$ is a uniquely determined permutation of $\{1, 2, \ldots, n\}$ such that $a_{i_j} = a_{i_{j+1}}$ implies $i_j < i_{j+1}$; in other words, $i_j > i_{j+1}$ implies that $a_{i_j} < a_{i_{j+1}}$. If the permutation $i_1\, i_2 \ldots i_n$ has k runs, we will show that the number of corresponding sequences $a_1\, a_2 \ldots a_n$ is $\binom{m+n-k}{n}$, and this proves (8) if we replace k by $n + 1 - k$.

For example, if $n = 9$ and $i_1\, i_2 \ldots i_n = 3\ 5\ 7\ 1\ 6\ 8\ 9\ 4\ 2$, we want to count the number of sequences $a_1\, a_2 \ldots a_n$ such that

$$1 \leq a_3 \leq a_5 \leq a_7 < a_1 \leq a_6 \leq a_8 \leq a_9 < a_4 < a_2 \leq m; \tag{10}$$

this is the number of sequences $b_1\, b_2 \ldots b_9$ such that

$$1 \leq b_1 < b_2 < b_3 < b_4 < b_5 < b_6 < b_7 < b_8 < b_9 \leq m + 5,$$

since we can let $b_1 = a_3, b_2 = a_5 + 1, b_3 = a_7 + 2, b_4 = a_1 + 2, b_5 = a_6 + 3$, etc. The number of choices of the b's is simply the number of ways of choosing 9 things out of $m + 5$, namely $\binom{m+5}{9}$; a similar proof works for general n and k, and for any permutation $i_1\, i_2 \ldots i_n$ with k runs.

Since both sides of (8) are polynomials in m, we may replace m by any real number x, and we obtain an interesting representation of powers in terms of consecutive binomial coefficients:

$$x^n = \left\langle {n \atop 1} \right\rangle \binom{x}{n} + \left\langle {n \atop 2} \right\rangle \binom{x+1}{n} + \cdots + \left\langle {n \atop n} \right\rangle \binom{x+n-1}{n}, \qquad n \geq 1. \tag{11}$$

For example,

$$x^3 = \binom{x}{3} + 4\binom{x+1}{3} + \binom{x+2}{3}.$$

This is the key property of Eulerian numbers which makes them useful in the study of discrete mathematics.

Setting $x = 1$ in (11) proves again that

$$\left\langle {n \atop n} \right\rangle = 1,$$

since the binomial coefficients vanish in all but the last term. Setting $x = 2$ yields

$$\left\langle {n \atop n-1} \right\rangle = 2^n - n - 1, \qquad n \geq 1. \tag{12}$$

Setting $x = 3, 4, \ldots$ shows that relation (11) completely defines the numbers $\left\langle {n \atop k} \right\rangle$, and leads to a formula originally given by Euler:

$$\begin{aligned} \left\langle {n \atop k} \right\rangle &= k^n - (k-1)^n \binom{n+1}{1} + (k-2)^n \binom{n+1}{2} - \cdots + (-1)^k 0^n \binom{n+1}{k} \\ &= \sum_{0 \leq j \leq k} (-1)^j (k-j)^n \binom{n+1}{j}, \qquad n \geq 0,\ k \geq 0. \end{aligned} \tag{13}$$

Now let us study the generating function for runs; if we set

$$g_n(z) = \sum_k \left\langle {n \atop k} \right\rangle z^k / n!, \tag{14}$$

the coefficient of z^k is the probability that a random permutation of $\{1, 2, \ldots, n\}$ has exactly k runs. Since k runs are just as likely as $n + 1 - k$, the average

Table 1

EULERIAN NUMBERS

n	$\left\langle {n \atop 0} \right\rangle$	$\left\langle {n \atop 1} \right\rangle$	$\left\langle {n \atop 2} \right\rangle$	$\left\langle {n \atop 3} \right\rangle$	$\left\langle {n \atop 4} \right\rangle$	$\left\langle {n \atop 5} \right\rangle$	$\left\langle {n \atop 6} \right\rangle$	$\left\langle {n \atop 7} \right\rangle$
0	0	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
2	0	1	1	0	0	0	0	0
3	0	1	4	1	0	0	0	0
4	0	1	11	11	1	0	0	0
5	0	1	26	66	26	1	0	0
6	0	1	57	302	302	57	1	0
7	0	1	120	1191	2416	1191	120	1

number of runs must be $\frac{1}{2}(n+1)$, hence $g_n'(1) = \frac{1}{2}(n+1)$. Exercise 2(b) shows that there is a simple formula for *all* the derivatives of $g_n(z)$ at the point $z = 1$:

$$g_n^{(m)}(1) = \left\{ {n+1 \atop n+1-m} \right\} \Big/ \binom{n}{m}, \qquad n \geq m. \tag{15}$$

Thus in particular the variance $g_n''(1) + g_n'(1) - g_n'(1)^2$ comes to $\frac{1}{12}(n+1)$, for $n \geq 2$, indicating a rather stable distribution about the mean. (We found

this same quantity in Eq. 3.3.2–18, where it was called covar (R_1', R_1').) Since $g_n(z)$ is a polynomial, we can use formula (15) to deduce the Taylor series expansions

$$\begin{aligned} g_n(z) &= \frac{1}{n!} \sum_{0\le k\le n} (z-1)^{n-k} k! \left\{ {n+1 \atop k+1} \right\} \\ &= \frac{1}{n!} \sum_{0\le k\le n} z^{k+1}(1-z)^{n-k} k! \left\{ {n+1 \atop k+1} \right\}. \end{aligned} \tag{16}$$

The second of these equations follows from the first, since

$$g_n(z) = z^{n+1} g_n(1/z), \qquad n \ge 1, \tag{17}$$

by the symmetry condition (7). The Stirling number recurrence

$$\left\{ {n+1 \atop k+1} \right\} = (k+1) \left\{ {n \atop k+1} \right\} + \left\{ {n \atop k} \right\}$$

gives two slightly simpler representations,

$$g_n(z) = \frac{1}{n!} \sum_{0\le k\le n} z(z-1)^{n-k} k! \left\{ {n \atop k} \right\} = \frac{1}{n!} \sum_{0\le k\le n} z^k (1-z)^{n-k} k! \left\{ {n \atop k} \right\}. \tag{18}$$

The super generating function

$$g(z, x) = \sum_{n\ge 0} g_n(z) x^n = \sum_{k,n\ge 0} \left\langle {n \atop k} \right\rangle z^k x^n / n! \tag{19}$$

is therefore equal to

$$z \sum_{k,n\ge 0} \frac{((z-1)x)^n}{(z-1)^k} \left\{ {n \atop k} \right\} \frac{k!}{n!} = z \sum_{k\ge 0} \left(\frac{e^{(z-1)x} - 1}{z-1} \right)^k = \frac{z(1-z)}{e^{(z-1)x} - z}; \tag{20}$$

this is another relation discussed by Euler.

Further properties of the Eulerian numbers may be found in a survey paper by L. Carlitz [*Math. Magazine* **33** (1959), 247–260]; see also J. Riordan, *Introduction to Combinatorial Analysis* (New York: Wiley, 1958), 38–39, 214–219, 234–237.

Let us now consider the length of runs; how long will a run be, on the average? We have already studied the expected number of runs of given length, in Section 3.3.2; the average run length is approximately 2, in agreement with the fact that about $\frac{1}{2}(n+1)$ runs appear in a random permutation of length n. For applications to sorting algorithms, a slightly different viewpoint is useful; we will consider the length of the kth run of the permutation from left to right, for $k = 1, 2, \ldots$.

For example, how long is the first (leftmost) run of a random permutation $a_1\, a_2 \ldots a_n$? Its length is always ≥ 1, and its length is ≥ 2 exactly one-half the time (namely when $a_1 < a_2$). Its length is ≥ 3 exactly one-sixth the time (when $a_1 < a_2 < a_3$), and, in general, its length is $\geq m$ with probability $q_m = 1/m!$, for $1 \leq m \leq n$. The probability that its length is exactly equal to m is therefore

$$\begin{aligned} &p_m = q_m - q_{m+1} = 1/m! - 1/(m+1)!, \qquad \text{for} \qquad 1 \leq m < n; \\ &p_n = 1/n!. \end{aligned} \tag{21}$$

The average length is therefore

$$\begin{aligned} p_1 + 2p_2 + \cdots + np_n &= (q_1 - q_2) + 2(q_2 - q_3) + \cdots + (n-1)(q_{n-1} - q_n) + nq_n \\ &= q_1 + q_2 + \cdots + q_n \\ &= \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}. \end{aligned} \tag{22}$$

If we let $n \to \infty$, the limit is $e - 1 = 1.71828\ldots$, and for finite n the value is $e - 1 - \delta_n$ where δ_n is quite small;

$$\delta_n = \frac{1}{(n+1)!}\left(1 + \frac{1}{n+2} + \frac{1}{(n+2)(n+3)} + \cdots\right) < \frac{e-1}{(n+1)!}.$$

For practical purposes it is therefore convenient to study runs in a random *infinite* sequence of distinct numbers

$$a_1, a_2, a_3, \ldots;$$

by "random" we mean in this case that each of the $n!$ possible relative orderings of the first n elements in the sequence is equally likely. The average length of the first run in a random infinite sequence is therefore $e - 1$.

By slightly sharpening this method, we can ascertain the average length of the kth run in a random sequence. Let q_{km} be the probability that the first k runs have total length $\geq m$; then q_{km} is $1/m!$ times the number of permutations of $\{1, 2, \ldots, m\}$ that have $\leq k$ runs,

$$q_{km} = \left(\left\langle {m \atop 1} \right\rangle + \cdots + \left\langle {m \atop k} \right\rangle\right) \Big/ m!. \tag{23}$$

The probability that the first k runs have total length m is $q_{km} - q_{k(m+1)}$. Therefore if L_k denotes the average length of the kth run, we find that

$$\begin{aligned} L_1 + \cdots + L_k &= \text{average total length of first } k \text{ runs} \\ &= (q_{k1} - q_{k2}) + 2(q_{k2} - q_{k3}) + 3(q_{k3} - q_{k4}) + \cdots \\ &= q_{k1} + q_{k2} + q_{k3} + \cdots. \end{aligned}$$

Subtracting $L_1 + \cdots + L_{k-1}$ and using the value of q_{km} in (23) yields the desired formula

$$L_k = \frac{1}{1!}\left\langle {1 \atop k} \right\rangle + \frac{1}{2!}\left\langle {2 \atop k} \right\rangle + \frac{1}{3!}\left\langle {3 \atop k} \right\rangle + \cdots = \sum_{m \geq 1} \left\langle {m \atop k} \right\rangle \frac{1}{m!}. \tag{24}$$

Since $\left\langle {0 \atop k} \right\rangle = 0$ except when $k = 1$, L_k turns out to be the coefficient of z^k in the generating function $g(z, 1) - z$ (see Eq. 19), so we have

$$L(z) = \sum_{k \geq 0} L_k z^k = \frac{z(1 - z)}{e^{z-1} - z} - z. \tag{25}$$

From Euler's formula (13) we obtain a representation of L_k as a polynomial in e:

$$\begin{aligned} L_k &= \sum_{m \geq 0} \sum_{0 \leq j \leq k} (-1)^{k-j} \binom{m+1}{k-j} \frac{j^m}{m!} \\ &= \sum_{0 \leq j \leq k} (-1)^{k-j} \sum_{m \geq 0} \binom{m}{k-j} \frac{j^m}{m!} + \sum_{0 \leq j \leq k} (-1)^{k-j} \sum_{m \geq 0} \binom{m}{k-j-1} \frac{j^m}{m!} \\ &= \sum_{0 \leq j \leq k} \frac{(-1)^{k-j} j^{k-j}}{(k-j)!} \sum_{n \geq 0} \frac{j^n}{n!} + \sum_{0 \leq j \leq k} \frac{(-1)^{k-j} j^{k-j-1}}{(k-j-1)!} \sum_{n \geq 0} \frac{j^n}{n!} \\ &= k \sum_{0 \leq j \leq k} e^j \frac{(-1)^{k-j} j^{k-j-1}}{(k-j)!}. \end{aligned} \tag{26}$$

This formula for L_k was first obtained by B. J. Gassner [see *CACM* **10** (1967), 89–93]. In particular, we have

$$\begin{aligned} L_1 &= e - 1 && \approx 1.71828\ldots; \\ L_2 &= e^2 - 2e && \approx 1.95249\ldots; \\ L_3 &= e^3 - 3e^2 + \tfrac{3}{2}e && \approx 1.99579\ldots. \end{aligned}$$

So the second run is expected to be longer than the first, and the third run will be longer yet, on the average! This may seem surprising at first glance, but a moment's reflection shows that, since the first element of the second run tends to be small (it caused the first run to terminate), there is a better chance for the second run to go on longer. The first element of the third run will tend to be even smaller than that of the second.

The numbers L_k are of importance in the theory of replacement-selection sorting (Section 5.4.1), so it is interesting to study their values in detail. Table 2, which has been computed by J. W. Wrench, Jr., shows the first 18 values of L_k to 15 decimal places. Our discussion in the preceding paragraph might lead us to suspect at first that $L_{k+1} > L_k$, but in fact the values oscillate back and forth. Note that L_k rapidly approaches the limiting value 2; it is quite remarkable to see these monic polynomials in the transcendental number e con-

Table 2

AVERAGE LENGTH OF THE kTH RUN

k	L_k	k	L_k
1	1.71828 18284 59045+	10	2.00000 00012 05997+
2	1.95249 24420 12560−	11	2.00000 00001 93672+
3	1.99579 13690 84285−	12	1.99999 99999 99909+
4	2.00003 88504 76806−	13	1.99999 99999 97022−
5	2.00005 75785 89716+	14	1.99999 99999 99719+
6	2.00000 50727 55710−	15	2.00000 00000 00019+
7	1.99999 96401 44022+	16	2.00000 00000 00006+
8	1.99999 98889 04744+	17	2.00000 00000 00000+
9	1.99999 99948 43434−	18	2.00000 00000 00000−

verging to the rational number 2 so quickly! The polynomials (26) are also somewhat interesting from the standpoint of numerical analysis, since they provide an excellent example of the loss of significant figures when nearly equal numbers are subtracted; using 19-digit floating-point arithmetic, Gassner incorrectly concluded that $L_{12} > 2$, and Wrench has remarked that 42-digit floating-point arithmetic gives L_{28} correct to only 29 significant digits.

The asymptotic behavior of L_k can be determined by using simple principles of complex variable theory. The denominator of (25) is zero only when $e^{z-1} = z$, i.e. (writing $z = x + iy$) when

$$e^{x-1}\cos y = x \qquad \text{and} \qquad e^{x-1}\sin y = y. \tag{27}$$

Figure 3 shows the superimposed graphs of these two equations, and we note that they intersect at the points $z = z_0, z_1, \bar{z}_1, z_2, \bar{z}_2, \ldots$, where $z_0 = 1$,

$$z_1 = (3.08884\ 30156\ 13044-) + (7.46148\ 92856\ 54255-)i, \tag{28}$$

and the imaginary part $\Im(z_{k+1})$ is roughly equal to $\Im(z_k) + 2\pi$ for large k. Since

$$\lim_{z\to z_k}\left(\frac{1-z}{e^{z-1}-z}\right)(z - z_k) = -1, \qquad \text{for} \qquad k > 0,$$

and since the limit is -2 for $k = 0$, the function

$$R_m(z) = L(z) + \frac{2}{z - z_0} + \frac{z_1}{z - z_1} + \frac{\bar{z}_1}{z - \bar{z}_1} + \frac{z_2}{z - z_2} + \frac{\bar{z}_2}{z - \bar{z}_2} + \cdots + \frac{z_m}{z - z_m} + \frac{\bar{z}_m}{z - \bar{z}_m}$$

has no singularities in the complex plane for $|z| < |z_{m+1}|$. Hence $R_m(z)$ has a power series expansion $\sum_k \rho_k z^k$ which converges absolutely when $|z| < |z_{m+1}|$; it follows that $\rho_k M^k \to 0$ as $k \to \infty$, where $M = |z_{m+1}| - \epsilon$. The coefficients

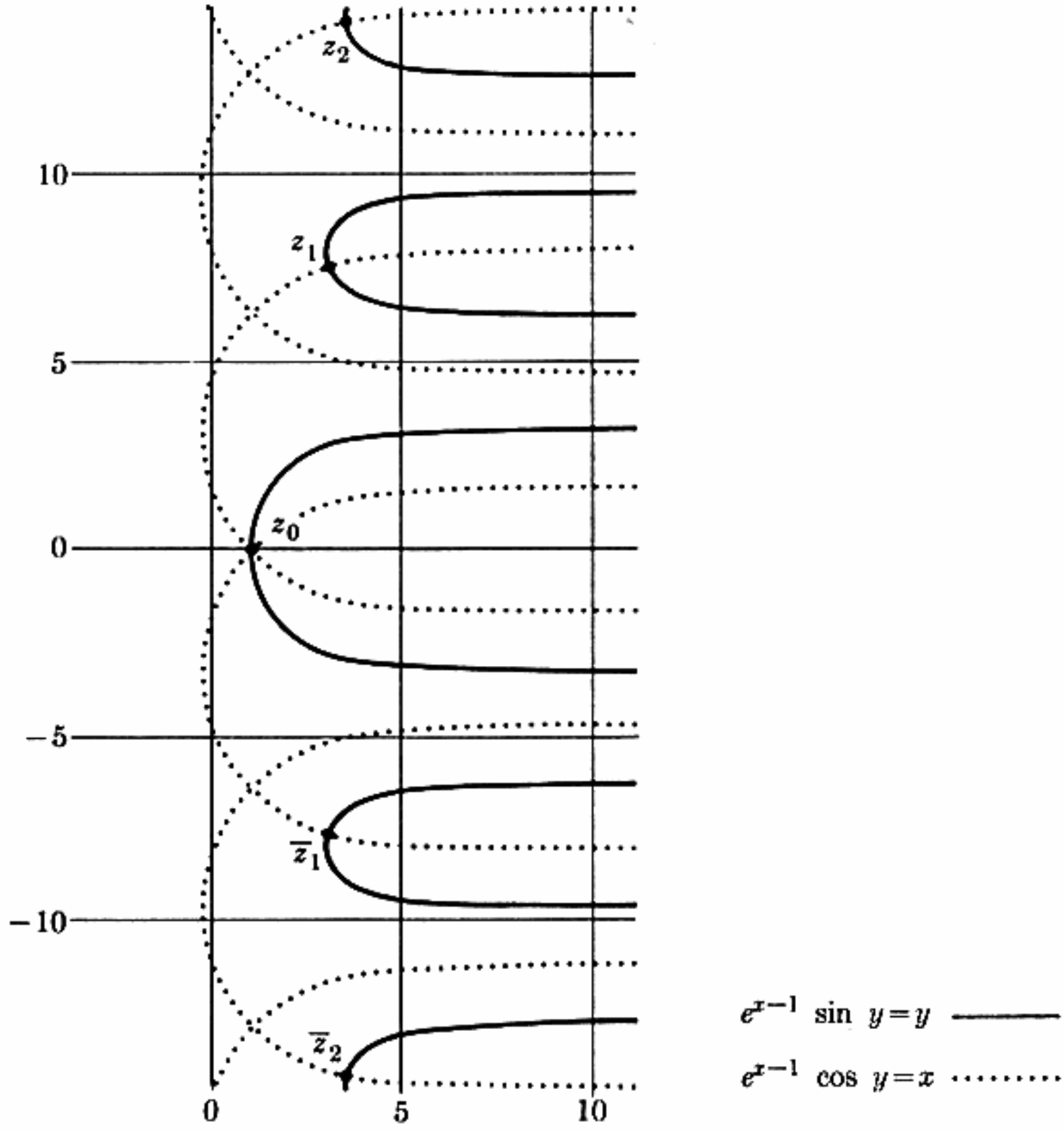


Fig. 3. Roots of $e^{z-1} = z$.

of $L(z)$ are the coefficients of

$$\frac{2}{1-z} + \frac{1}{1-z/z_1} + \frac{1}{1-z/\bar{z}_1} + \cdots + \frac{1}{1-z/z_m} + \frac{1}{1-z/\bar{z}_m} + R_m(z),$$

namely,

$$L_n = 2 + 2r_1^{-n}\cos n\theta_1 + 2r_2^{-n}\cos n\theta_2 + \cdots + 2r_m^{-n}\cos n\theta_m + O(r_{m+1}^{-n}), \quad (29)$$

if we let

$$z_k = r_k e^{i\theta_k}. \quad (30)$$

This shows the asymptotic behavior of L_n. We have

$$\begin{aligned} r_1 &= 8.07556\ 64528\ 89526-, \\ \theta_1 &= 1.17830\ 39784\ 74668+; \end{aligned}$$

$$\begin{aligned} r_2 &= 14.35457-, & \theta_2 &= 1.31269-; \\ r_3 &= 20.62073+, & \theta_3 &= 1.37428-; \\ r_4 &= 26.88795+, & \theta_4 &= 1.41050-; \end{aligned} \quad (31)$$

so the main contribution to $L_n - 2$ is due to r_1 and θ_1, and convergence of

(29) is quite rapid. The stated values of r_1 and θ_1 were determined by J. W. Wrench, Jr. Further analysis [W. W. Hooker, *CACM* **12** (1969), 411–413] shows that $R_m(z) \to -z$ as $m \to \infty$; hence the series $2\sum_{k\geq 0} r_k^{-n} \cos n\theta_k$ actually *converges* to L_n when $n > 1$.

A more careful examination of probabilities can be carried out to determine the complete probability distribution for the length of the kth run and for the total length of the first k runs (see exercises 9, 10, 11). The sum $L_1 + \cdots + L_k$ turns out to be asymptotically $2k - \frac{1}{3}$.

Let us conclude this section by considering the properties of runs when equal elements are allowed to appear in the permutations. The famous nineteenth-century American astronomer Simon Newcomb amused himself by playing a game of solitaire related to this question. He would deal a deck of cards into a pile, so long as the face values were in nondecreasing order; but whenever the next card to be dealt had a face value lower than its predecessor, he would start a new pile. He wanted to know the probability that a given number of piles would be formed after the entire deck had been dealt out in this manner.

Simon Newcomb's problem therefore consists of finding the probability distribution of runs in a random permutation of a multiset. The general answer is rather complicated (see exercise 12), although we have already seen how to solve the special case when all cards have a distinct face value. We will content ourselves here with a derivation of the *average* number of piles which appear in the game.

Suppose first that there are m different types of cards, each occurring exactly p times. An ordinary bridge deck, for example, has $m = 13$ and $p = 4$ if suits are disregarded. A remarkable symmetry applying to this case was discovered by P. A. MacMahon [*Combinatory Analysis* **1** (Cambridge, 1915), 212–213]: The number of permutations with $k+1$ runs is the same as the number with $mp - p - k + 1$ runs. When $p = 1$, this relation is easy to verify (Eq. 7), but for $p > 1$ it is quite surprising.

We can prove the symmetry by setting up a one-to-one correspondence between the permutations in such a way that each permutation with $k+1$ runs corresponds to another having $mp - p - k + 1$. The reader is urged to try his hand at discovering such a correspondence before reading further.

No very simple correspondence is evident; MacMahon's proof was based on generating functions instead of a combinatorial construction. But Foata's correspondence (Theorem 5.1.2B) provides a useful simplification, because it tells us that there is a one-to-one correspondence between permutations with $k+1$ runs and permutations whose two-line notation contains exactly k columns $\binom{y}{x}$ with $x < y$.

Suppose the given multiset is $\{p \cdot 1, p \cdot 2, \ldots, p \cdot m\}$, and consider the permutation whose two-line notation is

$$\begin{pmatrix} 1 & \ldots & 1 & 2 & \ldots & 2 & \ldots & m & \ldots & m \\ x_{11} & \ldots & x_{1p} & x_{21} & \ldots & x_{2p} & \ldots & x_{m1} & \ldots & x_{mp} \end{pmatrix}. \tag{32}$$

We can associate this permutation with another permutation of the same multiset,

$$\begin{pmatrix} 1 & \dots & 1 & 2 & \dots & 2 & \dots & m & \dots & m \\ x'_{11} & \dots & x'_{1p} & x'_{m1} & \dots & x'_{mp} & \dots & x'_{21} & \dots & x'_{2p} \end{pmatrix}, \tag{33}$$

where $x' = m + 1 - x$. If (32) contains k columns of the form $\binom{y}{x}$ with $x < y$, then (33) contains $(m-1)p - k$ such columns; for we need only consider the case $y > 1$, and $x < y$ is equivalent to $x' \geq m + 2 - y$. Since (32) corresponds to a permutation with $k+1$ runs, and (33) corresponds to a permutation with $mp - p - k + 1$ runs, and since the transformation which takes (32) into (33) is reversible [it takes (33) back into (32)], MacMahon's symmetry condition has been established. See exercise 14 for an example of this construction.

Because of the symmetry property, the average number of runs in a random permutation must be $\frac{1}{2}\big((k+1) + (mp - p - k + 1)\big) = 1 + \frac{1}{2}p(m-1)$. For example, the average number of piles resulting from Simon Newcomb's solitaire game using a standard deck will be 25 (so it doesn't appear to be a very exciting way to play solitaire).

We can actually determine the average number of runs in general, using a fairly simple argument, given *any* multiset $\{n_1 \cdot x_1, n_2 \cdot x_2, \dots, n_m \cdot x_m\}$ where the x's are distinct. Let $n = n_1 + n_2 + \cdots + n_m$, and imagine that all of the permutations $a_1\, a_2 \dots a_n$ of this multiset have been written down; we will count how often a_i is greater than a_{i+1}, for each fixed value of i, $1 \leq i < n$. The number of times $a_i > a_{i+1}$ is just half of the number of times $a_i \neq a_{i+1}$; and it is not difficult to see that $a_i = a_{i+1} = x_j$ exactly $Nn_j(n_j - 1)/n(n-1)$ times, where N is the total number of permutations. Hence $a_i = a_{i+1}$ exactly

$$\frac{N}{n(n-1)}\left(n_1(n_1-1) + \cdots + n_m(n_m-1)\right) = \frac{N}{n(n-1)}\left(n_1^2 + \cdots + n_m^2 - n\right)$$

times, and $a_i > a_{i+1}$ exactly

$$\frac{N}{2n(n-1)}\left(n^2 - (n_1^2 + \cdots + n_m^2)\right)$$

times. Summing over i and adding N, since a run ends at a_n in each permutation, we obtain the total number of runs among all N permutations:

$$N\left(\frac{n}{2} - \frac{1}{2n}(n_1^2 + \cdots + n_m^2) + 1\right). \tag{34}$$

Dividing by N gives the desired average number of runs.

Since runs are important in the study of "order statistics," there is a fairly large literature dealing with them, including several other types of runs not considered here. For additional information, see the book *Combinatorial Chance* by F. N. David and D. E. Barton (London: Griffin, 1962), Chapter 10; and

the survey paper by D. E. Barton and C. L. Mallows, *Annals of Math. Statistics* **36** (1965), 236–260. Further connections between Eulerian numbers and permutations appear in *Théorie Géométrique des Polynômes Eulériens* by D. Foata and M. P. Schützenberger, *Lecture Notes in Math.* **138** (Berlin: Springer, 1970), 94 pp.

EXERCISES

1. [*M26*] Derive Euler's formula (13).

▶ **2.** [*M22*] (a) Extend the idea used in the text to prove (8), considering those sequences $a_1\, a_2 \ldots a_n$ which contain exactly q distinct elements, in order to prove that

$$\sum_k \left\langle {n \atop k} \right\rangle \binom{k-1}{n-q} = \left\{ {n \atop q} \right\} q!.$$

(b) Use this identity to prove that

$$\sum_k \left\langle {n \atop k} \right\rangle \binom{k}{m} = \left\{ {n+1 \atop n+1-m} \right\} (n-m)!, \qquad \text{for} \qquad n \geq m.$$

3. [*HM25*] Evaluate the sum $\sum_k \langle {n \atop k} \rangle (-1)^k$.

4. [*M21*] What is the value of

$$\sum_k (-1)^k \left\{ {n \atop k} \right\} k! \binom{n-k}{m} ?$$

5. [*M20*] Find the value of $\langle {p \atop k} \rangle \bmod p$ when p is prime.

▶ **6.** [*M21*] Mr. B. C. Dull noticed that, by Eqs. (4) and (13),

$$n! = \sum_{k\geq 0} \left\langle {n \atop k} \right\rangle = \sum_{k\geq 0} \sum_{j\geq 0} (-1)^{k-j} \binom{n+1}{k-j} j^n.$$

Carrying out the sum on k first, he found that $\sum_{k\geq 0}(-1)^{k-j}\binom{n+1}{k-j} = 0$ for all $j \geq 0$; hence $n! = 0$ for all $n \geq 0$. Did he make a mistake?

7. [*HM40*] Is the probability distribution of runs, given by (14), asymptotically normal? (Cf. exercise 1.2.10–13.)

8. [*M24*] (P. A. MacMahon.) Show that the probability that the first run of a sufficiently long permutation has length l_1, the second has length l_2, . . . , and the kth has length $\geq l_k$, is

$$\det \begin{pmatrix} 1/l_1! & 1/(l_1+l_2)! & 1/(l_1+l_2+l_3)! & \ldots & 1/(l_1+l_2+l_3+\cdots+l_k)! \\ 1 & 1/l_2! & 1/(l_2+l_3)! & \ldots & 1/(l_2+l_3+\cdots+l_k)! \\ 0 & 1 & 1/l_3! & \ldots & 1/(l_3+\cdots+l_k)! \\ \vdots & & & & \vdots \\ 0 & 0 & \ldots & 1 & 1/l_k! \end{pmatrix}.$$

9. [*M30*] Let $h_k(z) = \sum p_{km} z^m$, where p_{km} is the probability that m is the total

length of the first k runs in a random (infinite) sequence. Find "simple" expressions for $h_1(z)$, $h_2(z)$, and the super generating function $h(z, x) = \sum_k h_k(z)x^k$.

10. [*HM30*] Find the asymptotic behavior of the mean and variance of the distributions $h_k(z)$ in the preceding exercise, for large k.

11. [*M40*] Let $H_k(z) = \sum p_{km}z^m$, where p_{km} is the probability that m is the length of the kth run in a random (infinite) sequence. Express $H_1(z)$, $H_2(z)$, and the super generating function $H(z, x) = \sum_k H_k(z)x^k$ in terms of familiar functions.

12. [*M33*] (P. A. MacMahon.) Generalize Eq. (13) to permutations of a multiset, by proving that the number of permutations of $\{n_1 \cdot 1, n_2 \cdot 2, \ldots, n_m \cdot m\}$ having exactly k runs is

$$\sum_{0\le j\le k} (-1)^j \binom{n+1}{j} \binom{n_1 - 1 + k - j}{n_1} \binom{n_2 - 1 + k - j}{n_2} \cdots \binom{n_m - 1 + k - j}{n_m},$$

where $n = n_1 + n_2 + \cdots + n_m$.

13. [*05*] If Simon Newcomb's solitaire game is played with a standard bridge deck, ignoring face value but treating ♣ < ♦ < ♥ < ♠, what is the average number of piles?

14. [*M17*] The permutation 3 1 1 1 2 3 1 4 2 3 3 4 2 2 4 4 has 5 runs; find the corresponding permutation with 9 runs, according to the text's construction for MacMahon's symmetry condition.

▸ **15.** [*M21*] (*Alternating runs.*) The classical nineteenth-century literature of combinatorial analysis did not treat the topic of runs in permutations, as we have considered them, but there were several papers dealing with "runs" which are alternately ascending and descending. Thus 5 3 2 4 7 6 1 8 was considered to have 4 runs,

5 3 2, 2 4 7, 7 6 1, 1 8.

(The first run would be ascending or descending, according as $a_1 < a_2$ or $a_1 > a_2$; thus $a_1\, a_2 \ldots a_n$ and $a_n \ldots a_2\, a_1$ and $(n+1-a_1)(n+1-a_2)\ldots(n+1-a_n)$ all have the same number of alternating runs.) When n elements are being permuted, the maximum number of runs of this kind is $n - 1$.

Find the average number of alternating runs in a random permutation of the set $\{1, 2, \ldots, n\}$. [*Hint:* Consider the proof of (34).]

16. [*M30*] Continuing the previous exercise, let $\langle\langle {n \atop k} \rangle\rangle$ be the number of permutations of $\{1, 2, \ldots, n\}$ which have exactly k alternating runs. Find a recurrence relation, by means of which a table of $\langle\langle {n \atop k} \rangle\rangle$ can be computed; and find the corresponding recurrence relation for the generating function $G_n(z) = \sum_k \langle\langle {n \atop k} \rangle\rangle z^k/n!$. Use the latter recurrence to discover a simple formula for the *variance* of the number of alternating runs in a random permutation of $\{1, 2, \ldots, n\}$.

17. [*M25*] Among all 2^n sequences $a_1\, a_2 \ldots a_n$, where each a_j is either 0 or 1, how many have exactly k runs (i.e., $k - 1$ occurrences of $a_j > a_{j+1}$)?

18. [*M27*] Among all $n!$ sequences $a_1\, a_2 \ldots a_n$, where each a_j is an integer in the range $0 \le a_j \le n - j$, how many have exactly k runs (i.e., $k - 1$ occurrences of $a_j > a_{j+1}$)?

▸ **19.** [*M26*] (J. Riordan.) (a) In how many ways can n nonattacking rooks (i.e., no two in the same row or column) be placed on an $n \times n$ chessboard, so that exactly k lie on a given side of the main diagonal? (b) In how many ways can k nonattacking rooks be placed on a given side of the main diagonal of an $n \times n$ chessboard?

For example, Fig. 4 shows one of the 15619 ways to put eight nonattacking rooks on a standard chessboard with three rooks in the unshaded portion below the main diagonal, together with one of the 1050 ways to put three nonattacking rooks on a triangular board.

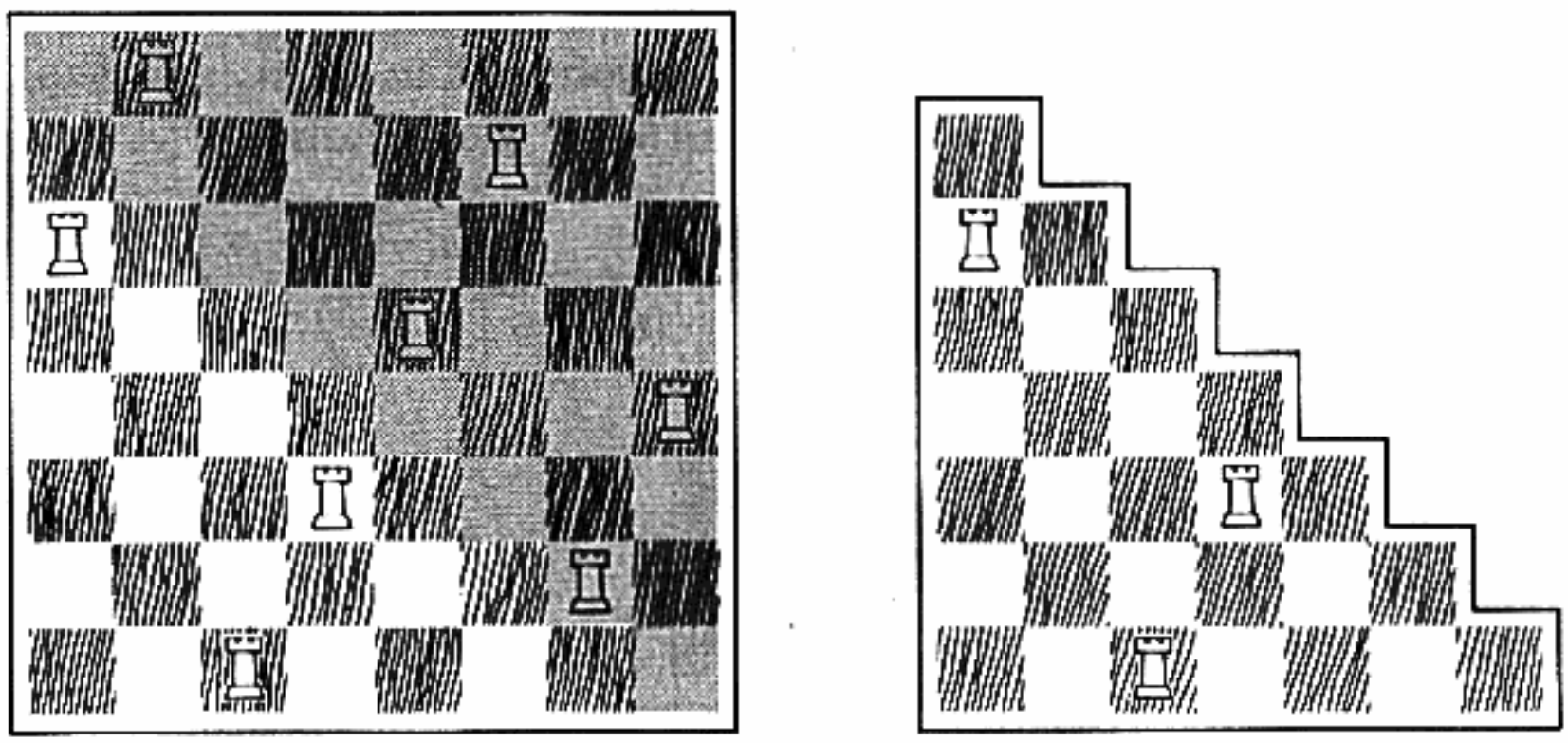


Fig. 4. Nonattacking rooks on a chessboard, with a given number below the main diagonal.

▸ **20.** [*M21*] A permutation is said to require k *readings* if we must scan it k times from left to right in order to read off its elements in nondecreasing order. For example, the permutation

4 9 1 8 2 5 3 6 7

requires four readings: On the first reading we obtain 1, 2, 3; on the second we get 4, 5, 6, 7; then 8; then 9. Find a connection between runs and readings.

21. [*M22*] If the permutation $a_1\, a_2 \ldots a_n$ of $\{1, 2, \ldots, n\}$ has k runs and requires j readings, in the sense of exercise 20, what can be said about $a_n \ldots a_2\, a_1$?

22. [*M26*] (L. Carlitz, D. P. Roselle, and R. A. Scoville.) Show that there is no permutation of $\{1, 2, \ldots, n\}$ with $n + 1 - r$ runs, and requiring s readings, if $rs < n$; but such permutations do exist if $n \geq n + 1 - r \geq s \geq 1$, $rs \geq n$.

23. [*HM42*] (Walter Weissblum.) The "long runs" of a permutation $a_1\, a_2 \ldots a_n$ are obtained by placing vertical lines just before a segment fails to be monotonic; long runs are either increasing or decreasing, depending on the order of their first two elements, so the length of each long run (except possibly the last) is ≥ 2. For example, 7 5|6 2|3 8 9|1 4 has four long runs. Find the average length of the first two long runs of an infinite permutation, and prove that the limiting long-run length is

$$(1 + \cot \tfrac{1}{2})/(3 - \cot \tfrac{1}{2}) \approx 2.4202.$$

24. [*M30*] What is the average number of runs in sequences generated as in exercise 5.1.1–18, as a function of p?

To complete our survey of the combinatorial properties of permutations, we will discuss some remarkable relations which connect permutations with arrays of integers called tableaux. A *Young tableau of shape* $(n_1, n_2, \ldots, n_m)$, where $n_1 \geq n_2 \geq \cdots \geq n_m \geq 0$, is an arrangement of $n_1 + n_2 + \cdots + n_m$ distinct integers in an array of left-justified rows, with n_i elements in row i, such that the entries of each row are in increasing order from left to right, and the entries of each column are increasing from top to bottom. For example,

1	2	5	9	10	15
3	6	7	13		
4	8	12	14		
11					

(1)

is a Young tableau of shape (6, 4, 4, 1). Such arrangements were introduced by Alfred Young in 1900 as an aid to the study of matrix representations of permutations [see, for example, D. E. Rutherford, *Substitutional Analysis* (New York: Hafner, 1968)]. For simplicity, we will simply say "tableau" instead of "Young tableau."

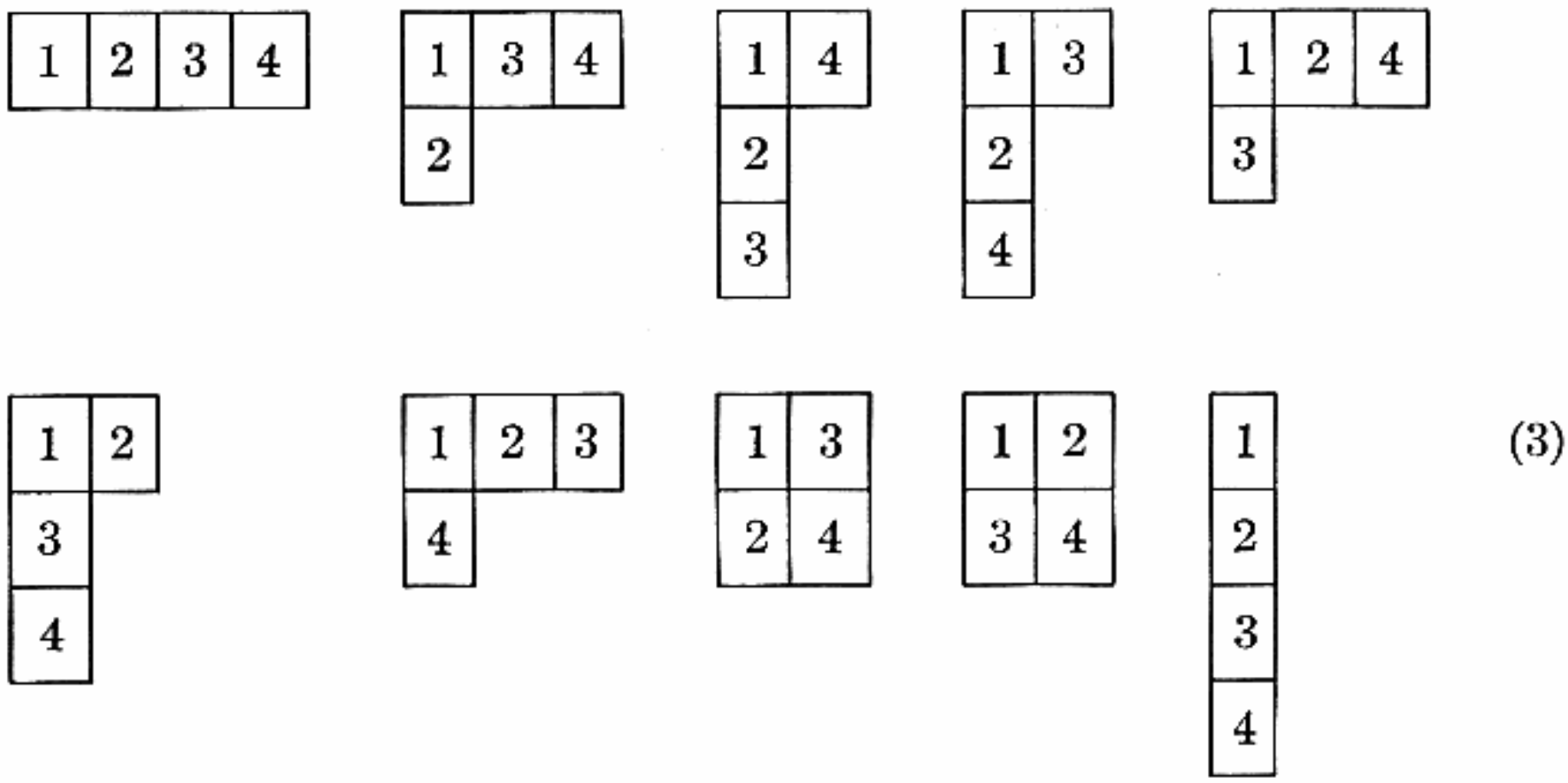
An *involution* is a permutation which is its own inverse. For example, there are 10 involutions of {1, 2, 3, 4},

$$\begin{pmatrix}1&2&3&4\\1&2&3&4\end{pmatrix}\ \begin{pmatrix}1&2&3&4\\2&1&3&4\end{pmatrix}\ \begin{pmatrix}1&2&3&4\\3&2&1&4\end{pmatrix}\ \begin{pmatrix}1&2&3&4\\4&2&3&1\end{pmatrix}\ \begin{pmatrix}1&2&3&4\\1&3&2&4\end{pmatrix}$$
$$\begin{pmatrix}1&2&3&4\\1&4&3&2\end{pmatrix}\ \begin{pmatrix}1&2&3&4\\1&2&4&3\end{pmatrix}\ \begin{pmatrix}1&2&3&4\\2&1&4&3\end{pmatrix}\ \begin{pmatrix}1&2&3&4\\3&4&1&2\end{pmatrix}\ \begin{pmatrix}1&2&3&4\\4&3&2&1\end{pmatrix} \tag{2}$$

The term "involution" originated in classical geometry problems; involutions in the general sense considered here were first studied by H. A. Rothe when he introduced the concept of inverses (see Section 5.1.1).

It may appear strange that we should be discussing both tableaux and involutions at the same time, but there is an extraordinary connection between these two apparently unrelated concepts: *The number of involutions of* $\{1, 2, \ldots, n\}$ *is the same as the number of tableaux that can be formed from the*

elements $\{1, 2, \ldots, n\}$. For example, exactly 10 tableaux can be formed from $\{1, 2, 3, 4\}$, namely,

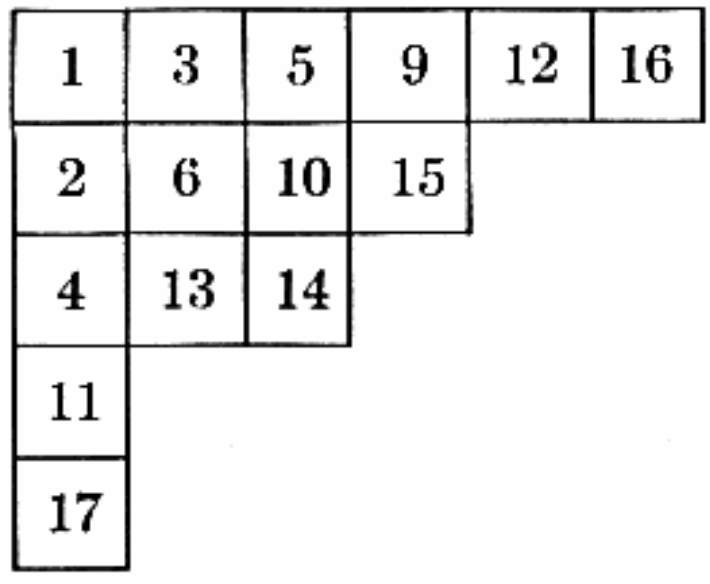


(3)

corresponding respectively to the 10 involutions (2).

This connection between involutions and tableaux is by no means obvious, and there is probably no very simple way to prove it. The proof we will discuss involves an interesting tableau-construction algorithm which has several other surprising properties; it is based on a special procedure which inserts new elements into a tableau.

For example, suppose that we want to insert the element 8 into the tableau



(4)

The method we will use starts by placing the 8 into row 1, in the spot previously occupied by 9, since 9 is the least element greater than 8 in that row. Element 9 is "bumped down" into row 2, where it displaces the 10. The 10 then "bumps" the 13 from row 3 to row 4; and since row 4 contains no element greater than 13, the process terminates by inserting 13 at the right end of row 4. The tableau

has been transformed into

1	3	5	8	12	16
2	6	9	15		
4	10	14			
11	13				
17					

(5)

A precise description of this process, together with a proof that it always preserves the tableau properties, appears in Algorithm I.

Algorithm I (*Insertion into a tableau*). Let $P = (P_{ij})$ be a tableau of positive integers, and let x be a positive integer not in P. This algorithm transforms P into another tableau which contains x in addition to its original elements. The new tableau has the same shape as the old, except for the addition of a new position in row s, column t, where s and t are quantities determined by the algorithm.

(Parenthesized remarks in this algorithm serve to prove its validity, since it is easy to verify inductively that the remarks are valid and that the array P remains a tableau throughout the process. For convenience we will assume that the tableau has been bordered by zeros at the top and left and with ∞'s to the right and below, so that P_{ij} is defined for all $i, j \geq 0$. If we define the relation

$$a \lesssim b \quad \text{if and only if} \quad a < b \quad \text{or} \quad a = b = \infty, \tag{6}$$

the tableau inequalities can be expressed in the convenient form

$$P_{ij} = 0 \quad \text{if} \quad i = 0 \quad \text{or} \quad j = 0;$$
$$P_{ij} \lesssim P_{i(j+1)} \quad \text{and} \quad P_{ij} \lesssim P_{(i+1)j}, \quad \text{for all} \quad i, j \geq 0. \tag{7}$$

The statement "$x \notin P$" means that either $x = \infty$ or $x \neq P_{ij}$ for all $i, j \geq 0$.)

I1. [Input x.] Set $i \leftarrow 1$, set $x_1 \leftarrow x$, and set j to the smallest value such that $P_{1j} = \infty$.

I2. [Find x_{i+1}.] (At this point $P_{(i-1)j} < x_i < P_{ij}$ and $x_i \notin P$.) If $x_i < P_{i(j-1)}$, decrease j by 1 and repeat this step. Otherwise set $x_{i+1} \leftarrow P_{ij}$ and set $r_i \leftarrow j$.

I3. [Replace by x_i.] (Now $P_{i(j-1)} < x_i < x_{i+1} = P_{ij} \lesssim P_{i(j+1)}$, $P_{(i-1)j} < x_i < x_{i+1} = P_{ij} \lesssim P_{(i+1)j}$, and $r_i = j$. Set $P_{ij} \leftarrow x_i$.

I4. [Is $x_{i+1} = \infty$?] (Now $P_{i(j-1)} < P_{ij} = x_i < x_{i+1} \lesssim P_{i(j+1)}$, $P_{(i-1)j} < P_{ij} = x_i < x_{i+1} \lesssim P_{(i+1)j}$, $r_i = j$, and $x_{i+1} \notin P$.) If $x_{i+1} \neq \infty$, increase i by 1 and return to step I2.

I5. [Determine s, t.] Set $s \leftarrow i$, $t \leftarrow j$, and terminate the algorithm. (At this point the conditions

$$P_{st} \neq \infty, P_{(s+1)t} = P_{s(t+1)} = \infty \tag{8}$$

are satisfied.) ∎

Note that this algorithm defines a "bumping sequence"

$$x = x_1 < x_2 < \cdots < x_s < x_{s+1} = \infty, \tag{9}$$

as well as an auxiliary sequence of column indices

$$r_1 \geq r_2 \geq \cdots \geq r_s = t; \tag{10}$$

element P_{ir_i} has been changed from x_{i+1} to x_i, for $1 \leq i \leq s$. For example, when we inserted 8 into (4), the bumping sequence was 8, 9, 10, 13, ∞, and the auxiliary sequence was 4, 3, 2, 2. We could have reformulated the algorithm so that it used much less temporary storage (only the current values of j, x_i, and x_{i+1} need to be remembered), but sequences (9) and (10) have been introduced so that we can prove interesting things about the algorithm.

The key fact we will use about Algorithm I is that it can be "run backwards"; given the values of s and t determined in step I5, we can transform P back into its original form again, determining and removing the element x which was inserted. For example, consider (5) and suppose we are told that element 13 is in the position that used to be blank. Then 13 must have been bumped down from row 3 by the 10, since 10 is the greatest element less than 13 in that row; similarly the 10 must have been bumped from row 2 by the 9, and the 9 must have been bumped from row 1 by the 8. Thus we can go from (5) back to (4). The following algorithm specifies this process in detail:

Algorithm D (*Deletion from a tableau*). Given a tableau P and positive integers s, t satisfying (8), this algorithm transforms P into another tableau, having almost the same shape, but with ∞ in column t of row s. An element x, determined by the algorithm, is deleted from P.

(As in Algorithm I, parenthesized assertions are included here to facilitate proving that P remains a tableau throughout the process.)

D1. [Input s, t.] Set $j \leftarrow t$, $i \leftarrow s$, $x_{s+1} \leftarrow \infty$.

D2. [Find x_i.] (At this point $P_{ij} < x_{i+1} \lesssim P_{(i+1)j}$ and $x_{i+1} \notin P$.) If $P_{i(j+1)} < x_{i+1}$, increase j by 1 and repeat this step. Otherwise set $x_i \leftarrow P_{ij}$ and $r_i \leftarrow j$.

D3. [Replace by x_{i+1}.] (Now $P_{i(j-1)} < P_{ij} = x_i < x_{i+1} \lesssim P_{i(j+1)}$, $P_{(i-1)j} < P_{ij} = x_i < x_{i+1} \lesssim P_{(i+1)j}$, and $r_i = j$.) Set $P_{ij} \leftarrow x_{i+1}$.

D4. [Is $i = 1$?] (Now $P_{i(j-1)} < x_i < x_{i+1} = P_{ij} \lesssim P_{i(j+1)}$, $P_{(i-1)j} < x_i < x_{i+1} = P_{ij} \lesssim P_{(i+1)j}$, and $r_i = j$.) If $i > 1$, decrease i by 1 and return to step D2.

D5. [Determine x.] Set $x \leftarrow x_1$ and terminate the algorithm. (Now $0 < x < \infty$.) ∎

The parenthesized assertions appearing in Algorithms I and D are not only a useful way to prove that the algorithms preserve the tableau structure; they also serve to verify that *Algorithms I and D are perfect inverses of each other*. If we perform Algorithm I first, given some tableau P and some positive integer $x \notin P$, it will insert x and determine positive integers s, t satisfying (8); Algorithm D applied to the result will recompute x and will restore P. Conversely, if we perform Algorithm D first, given some tableau P and some positive integers s, t satisfying (8), it will modify P, deleting some positive integer x; Algorithm I applied to the result will recompute s, t and will restore P. The reason is that the parenthesized assertions of steps I3 and D4 are identical, as are the assertions of steps I4 and D3, and these assertions characterize the value of j uniquely. Hence the auxiliary sequences (9), (10) are the same in each case.

Now we are ready to prove a basic property of tableaux:

Theorem A. *There is a one-to-one correspondence between the set of all permutations of* $\{1, 2, \ldots, n\}$ *and the set of ordered pairs* (P, Q) *of tableaux formed from* $\{1, 2, \ldots, n\}$, *where* P *and* Q *have the same shape.*

(An example of this theorem appears within the proof which follows.)

Proof. It is convenient to prove a slightly more general result. Given any two-line array

$$\begin{pmatrix} q_1 & q_2 & \cdots & q_n \\ p_1 & p_2 & \cdots & p_n \end{pmatrix}, \qquad \begin{array}{l} q_1 < q_2 < \cdots < q_n, \\ p_1, p_2, \ldots, p_n \text{ distinct}, \end{array} \tag{11}$$

we will construct two corresponding tableaux P and Q, where the elements of P are $\{p_1, \ldots, p_n\}$ and the elements of Q are $\{q_1, \ldots, q_n\}$ and the shape of P is the shape of Q.

Let P and Q be empty initially. Then, for $i = 1, 2, \ldots, n$ (in this order), do the following operation: Insert p_i into tableau P using Algorithm I; then set $Q_{st} \leftarrow q_i$, where s and t specify the newly-filled position of P.

For example, if the given permutation is $\left(\begin{smallmatrix}1&3&5&6&8\\7&2&9&5&3\end{smallmatrix}\right)$, we obtain

	P	Q
Insert 7:	7	1
Insert 2:	2 7	1 3
Insert 9:	2 9 7	1 5 3
Insert 5:	2 5 7 9	1 5 3 6
Insert 3:	2 3 5 9 7	1 5 3 6 8

(12)

so the tableaux (P, Q) corresponding to $\left(\begin{smallmatrix}1&3&5&6&8\\7&2&9&5&3\end{smallmatrix}\right)$ are

$$P = \begin{array}{|c|c|}\hline 2 & 3 \\ \hline 5 & 9 \\ \hline 7 \\ \cline{1-1}\end{array}, \qquad Q = \begin{array}{|c|c|}\hline 1 & 5 \\ \hline 3 & 6 \\ \hline 8 \\ \cline{1-1}\end{array}. \tag{13}$$

It is clear from this construction that P and Q always have the same shape; furthermore, since we always add elements on the periphery of Q, in increasing order, Q is a tableau.

Conversely, given two equal-shape tableaux P and Q, we can find the corresponding two-line array (11) as follows. Let the elements of Q be

$$q_1 < q_2 < \cdots < q_n.$$

For $i = n, \ldots, 2, 1$ (in this order), let p_i be the element x that is removed when Algorithm D is applied to P, using the values s, t such that $Q_{st} = q_i$.

For example, this construction will start with (13) and will successively undo the calculation (12) until P is empty, and $\left(\begin{smallmatrix}1&3&5&6&8\\7&2&9&5&3\end{smallmatrix}\right)$ is obtained.

Since Algorithms I and D are inverses of each other, the two constructions we have described are inverses of each other, and the one-to-one correspondence has been established. ∎

The correspondence defined in the proof of Theorem A has many startling properties, and we will now proceed to derive some of them. The reader is urged to try his hand at the example in exercise 1, in order to become familiar with the construction, before reading further.

Once an element has been bumped from row 1 to row 2, it doesn't affect row 1 any longer; furthermore rows 2, 3, . . . are built up from the sequence of "bumped" elements in exactly the same way as rows 1, 2, . . . are built up from the original permutation. These facts suggest that we can look at the construction of Theorem A in another way, concentrating only on the first rows of P and Q. For example, the permutation $\left(\begin{smallmatrix}1&3&5&6&8\\7&2&9&5&3\end{smallmatrix}\right)$ causes the following action in row 1 [cf. (12)]:

$$\begin{aligned}
&1: \quad \text{Insert 7, set } Q_{11} \leftarrow 1.\\
&3: \quad \text{Insert 2, bump 7.}\\
&5: \quad \text{Insert 9, set } Q_{12} \leftarrow 5.\\
&6: \quad \text{Insert 5, bump 9.}\\
&8: \quad \text{Insert 3, bump 5.}
\end{aligned} \tag{14}$$

Thus the first row of P is 2 3, and the first row of Q is 1 5. Furthermore, the remaining rows of P and Q are the tableaux corresponding to the "bumped" two-line array

$$\begin{pmatrix}3&6&8\\7&9&5\end{pmatrix}. \tag{15}$$

In order to study the behavior of the construction on row 1, we can consider the elements which go into a given column of this row. Let us say that (q_i, p_i) is in *class* t with respect to the two-line array

$$\begin{pmatrix}q_1&q_2&\cdots&q_n\\p_1&p_2&\cdots&p_n\end{pmatrix}, \qquad \begin{aligned}&q_1 < q_2 < \cdots < q_n,\\&p_1, p_2, \ldots p_n \text{ distinct,}\end{aligned} \tag{16}$$

if $p_i = P_{1t}$ after Algorithm I has been applied successively to $p_1, p_2, \ldots, p_i$,

starting with an empty tableau P. (Remember that Algorithm I always inserts the given element into row 1.)

It is easy to see that (q_i, p_i) is in class 1 if and only if p_i has $(i - 1)$ inversions, i.e., $p_i = \min\{p_1, p_2, \ldots, p_i\}$ is a "left-to-right minimum." If we cross out the columns of class 1 in (16), we obtain another two-line array

$$\begin{pmatrix} q'_1 & q'_2 & \cdots & q'_m \\ p'_1 & p'_2 & \cdots & p'_m \end{pmatrix} \tag{17}$$

such that (q, p) is in class t with respect to (17) if and only if it is in class $t + 1$ with respect to (16). The operation of going from (16) to (17) represents removing the leftmost position of row 1. This gives us a systematic way to determine the classes. For example in $\left(\begin{smallmatrix} 1 & 3 & 5 & 6 & 8 \\ 7 & 2 & 9 & 5 & 3 \end{smallmatrix}\right)$ the elements which are left-to-right minima are 7 and 2, so class 1 is $\{(1, 7), (3, 2)\}$; in the remaining array $\left(\begin{smallmatrix} 5 & 6 & 8 \\ 9 & 5 & 3 \end{smallmatrix}\right)$ all elements are minima, so class 2 is $\{(5, 9), (6, 5), (8, 3)\}$. In the "bumped" array (15), class 1 is $\{(3, 7), (8, 5)\}$ and class 2 is $\{(6, 9)\}$.

For any fixed value of t, the elements of class t can be labeled

$$(q_{i_1}, p_{i_1}), \ldots, (q_{i_k}, p_{i_k})$$

in such a way that

$$\begin{aligned} q_{i_1} &< q_{i_2} < \cdots < q_{i_k}, \\ p_{i_1} &> p_{i_2} > \cdots > p_{i_k}, \end{aligned} \tag{18}$$

since tableau position P_{1t} takes on the decreasing sequence of values $p_{i_1}, \ldots, p_{i_k}$ as the insertion algorithm proceeds. At the end of the construction we have

$$P_{1t} = p_{i_k}, \qquad Q_{1t} = q_{i_1}; \tag{19}$$

and the "bumped" two-line array which defines rows 2, 3, ... of P and Q contains the columns

$$\begin{pmatrix} q_{i_2} & q_{i_3} & \cdots & q_{i_k} \\ p_{i_1} & p_{i_2} & \cdots & p_{i_{k-1}} \end{pmatrix}, \tag{20}$$

plus other columns formed in a similar way from the other classes.

These observations lead to a simple method for calculating P and Q by hand (see exercise 3), and they also provide us with the means to prove a rather unexpected result:

Theorem B. *If the permutation*

$$\begin{pmatrix} 1 & 2 & \ldots & n \\ a_1 & a_2 & \ldots & a_n \end{pmatrix}$$

corresponds to tableaux (P, Q) in the construction of Theorem A, then the inverse permutation corresponds to (Q, P).

This fact is quite startling, since P and Q are formed by such completely different methods in Theorem A, and since the inverse of a permutation is obtained by juggling the columns of the two-line array rather capriciously.

Proof. Suppose that we have a two-line array (16); interchanging the lines and sorting the columns so that the new top line is in increasing order gives the "inverse" array,

$$\begin{aligned}\begin{pmatrix} q_1 & q_2 & \cdots & q_n \\ p_1 & p_2 & \cdots & p_n \end{pmatrix}^{-1} &= \begin{pmatrix} p_1 & p_2 & \cdots & p_n \\ q_1 & q_2 & \cdots & q_n \end{pmatrix} \\ &= \begin{pmatrix} p'_1 & p'_2 & \cdots & p'_n \\ q'_1 & q'_2 & \cdots & q'_n \end{pmatrix}, \quad \begin{matrix} p'_1 < p'_2 < \cdots < p'_n; \\ q'_1, q'_2, \ldots, q'_n \text{ distinct.} \end{matrix}\end{aligned} \tag{21}$$

We will show that this operation corresponds to interchanging P and Q in the construction of Theorem A.

Exercise 2 reformulates our remarks about class determination so that the class of (q_i, p_i) doesn't depend on the fact that $q_1, q_2, \ldots, q_n$ are in ascending order. Since the resulting condition is symmetrical in the q's and the p's, the operation (21) does not destroy the class structure; if (q, p) is in class t with respect to (16), then (p, q) is in class t with respect to (21). If we therefore arrange the elements of the latter class t as

$$\begin{aligned} p_{i_k} < \cdots < p_{i_2} < p_{i_1}, \\ q_{i_k} > \cdots > q_{i_2} > q_{i_1}; \end{aligned} \tag{22}$$

[cf. (18)], we have

$$P_{1t} = q_{i_1}, \qquad Q_{1t} = p_{i_k} \tag{23}$$

as in (19), and the columns

$$\begin{pmatrix} p_{i_{k-1}} & \cdots & p_{i_2} & p_{i_1} \\ q_{i_k} & \cdots & q_{i_3} & q_{i_2} \end{pmatrix} \tag{24}$$

go into the "bumped" array as in (20). Hence the first rows of P and Q are interchanged. Furthermore the "bumped" two-line array for (21) is the inverse of the "bumped" two-line array for (16), so the proof is completed by induction on the number of rows in the tableaux. ∎

Corollary. *The number of tableaux which can be formed from* $\{1, 2, \ldots, n\}$ *is the number of involutions on* $\{1, 2, \ldots, n\}$.

Proof. If π is an involution corresponding to (P, Q), then $\pi = \pi^{-1}$ corresponds to (Q, P); hence $P = Q$. Conversely, if π is any permutation corresponding to (P, P), then π^{-1} also corresponds to (P, P); hence $\pi = \pi^{-1}$. So there is a one-to-one correspondence between involutions π and tableaux P. ∎

It is clear that the upper-left corner element of a tableau is always the smallest. This suggests a possible way to sort a set of numbers: First we can put them into a tableau, by using Algorithm I repeatedly; this brings the smallest element to the corner. Then we delete this smallest element, rearranging the remaining elements so that they form another tableau; then we delete the new smallest element; and so on.

Let us therefore consider what happens when we delete the corner element from the tableau

1	3	5	8	12	16
2	6	9	15		
4	10	14			
11	13				
17					

(25)

If the 1 is removed, the 2 must come to take its place. Then we can move the 4 up to where the 2 was, but we can't move the 11 to the position of the 4; the 10 can be moved instead, then the 13 in place of the 10. In general, we are led to the following procedure.

Algorithm S (*Delete corner element*). Given a tableau P, this algorithm deletes the upper left corner element of P and moves other elements so that the tableau properties are preserved. The notational conventions of Algorithms I and D are used.

S1. [Initialize.] Set $r \leftarrow 1$, $s \leftarrow 1$.

S2. [Done?] If $P_{rs} = \infty$, the process is complete.

S3. [Compare.] If $P_{(r+1)s} \lesssim P_{r(s+1)}$, go to step S5. (We examine the elements just to the right and below the vacant cell, and we will move the smaller of these.)

S4. [Shift left.] Set $P_{rs} \leftarrow P_{r(s+1)}$, $s \leftarrow s + 1$, and return to S3.

S5. [Shift up.] Set $P_{rs} \leftarrow P_{(r+1)s}$, $r \leftarrow r + 1$, and return to S2. ∎

It is easy to prove that P is still a tableau after Algorithm S has deleted its corner element (see exercise 10). So if we repeat Algorithm S until P is empty, we can read out its elements in increasing order. Unfortunately this doesn't turn out to be as efficient a sorting algorithm as other methods we will see; its minimum running time is proportional to $n^{1.5}$, but similar algorithms which use trees instead of tableau structures have an execution time on the order of $n \log n$.

In spite of the fact that Algorithm S doesn't lead to a superbly-efficient sorting algorithm, it has some very interesting properties.

Theorem C (M. P. Schützenberger). *If P is the tableau formed by the construction of Theorem A from the permutation $a_1\ a_2\ \ldots\ a_n$, and if $a_i = \min\{a_1\ a_2, \ldots, a_n\}$, then Algorithm S changes P into the tableau corresponding to $a_1 \ldots a_{i-1}\ a_{i+1} \ldots a_n$.*

Proof. See exercise 13. ∎

After we apply Algorithm S to a tableau, let us put the deleted element into the newly-vacated place P_{rs}, with parentheses around it to indicate that it isn't really part of the tableau. For example, after applying this procedure to (25) we would have

2	3	5	8	12	16
4	6	9	15		
10	13	14			
11	(1)				
17					

and two more applications yields

4	5	8	12	16	(2)
6	9	14	15		
10	13	(3)			
11	(1)				
17					

Continuing until all elements are parenthesized, then removing the parentheses, gives

17	15	14	13	11	2
16	10	6	4		
12	5	3			
9	1				
8					

(26)

which has the same shape as the original tableau (25). This configuration may be called a *dual tableau*, since it is like a tableau except that the "dual order" (reversing the roles of $<$ and $>$) has been used. Let us denote the dual tableau formed from P in this way by the symbol P^S.

From P^S we can determine P uniquely; in fact, we can obtain the original tableau P from P^S, by applying exactly the same algorithm (reversing the order, since P^S is a dual tableau). For example, two steps of the algorithm applied to (26) give

15	14	13	11	2	(16)
12	10	6	4		
9	5	3			
8	1				
(17)					

and eventually (25) will be reproduced again! This remarkable fact is one of the consequences of our next theorem.

Theorem D (C. Schensted, M. P. Schützenberger). *Let*

$$\begin{pmatrix} q_1 & q_2 & \cdots & q_n \\ p_1 & p_2 & \cdots & p_n \end{pmatrix} \tag{27}$$

be the two-line array corresponding to the tableaux (P, Q).

a) *Using dual (reverse) order on the q's, but not on the p's, the two-line array*

$$\begin{pmatrix} q_n & \cdots & q_2 & q_1 \\ p_n & \cdots & p_2 & p_1 \end{pmatrix} \tag{28}$$

corresponds to $(P^T, (Q^S)^T)$.

(As usual, "T" denotes the operation of transposing rows and columns; P^T is a tableau, while $(Q^S)^T$ is a dual tableau, since the order of the q's is reversed.)

b) *Using dual order on the p's, but not on the q's, the two-line array* (27) *corresponds to* $((P^S)^T, Q^T)$.

c) *Using dual order on both the p's and the q's, the two-line array* (28) *corresponds to* (P^S, Q^S).

Proof. No simple proof of this theorem is known. The fact that case (a) corresponds to (P^T, X) for some dual tableau X is proved in exercise 6; hence by Theorem B, case (b) corresponds to (Y, Q^T) for some dual tableau Y, and Y must have the shape of P^T.

Let $p_i = \min\{p_1, \ldots, p_n\}$; since p_i is the "largest" element in the dual order, it appears on the periphery of Y, and it doesn't bump any elements in the construction of Theorem A. Thus, if we successively insert $p_1, \ldots, p_{i-1}$, $p_{i+1}, \ldots, p_n$ using the dual order, we get $Y - \{p_i\}$, that is, Y with p_i removed. By Theorem C if we successively insert $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n$ using the normal order, we get the tableau $d(P)$ obtained by applying Algorithm S to P. By induction on n, $Y - \{p_i\} = (d(P)^S)^T$. But since

$$(P^S)^T - \{p_i\} = (d(P)^S)^T, \tag{29}$$

by definition of the operation S, and since Y has the same shape as $(P^S)^T$, we must have $Y = (P^S)^T$.

This proves (b), and (a) follows by an application of Theorem B. Applying (a) and (b) successively then shows that case (c) corresponds to $(((P^T)^S)^T, ((Q^S)^T)^T)$; and this is (P^S, Q^S) since $(P^S)^T = (P^T)^S$ by the row-column symmetry of operation S. ∎

In particular, this theorem establishes two surprising facts about the tableau insertion algorithm: If successive insertion of distinct elements $p_1, \ldots, p_n$ into an empty tableau yields tableau P, insertion in the opposite order $p_n, \ldots, p_1$ yields the *transposed* tableau P^T. And if we not only insert the p's in this order $p_n, \ldots, p_1$ but also interchange the roles of $<$ and $>$, as well as 0 and ∞, in the insertion process, we obtain the dual tableau P^S. The reader is urged to try out these processes on some simple examples. The unusual nature of these coincidences might lead us to suspect that some sort of witchcraft is operating behind the scenes! No simple explanation for these phenomena is yet known; there seems to be no obvious way to prove even that case (c) corresponds to tableaux having the same *shape* as P and Q.

The correspondence of Theorem A was given by G. de B. Robinson [*American J. Math.* **60** (1938), 745–760, Sec. 5], in a somewhat vague and different form, as part of a solution to a rather difficult problem in group theory. It is not difficult to verify that his construction is essentially identical to that given here. He stated Theorem B without proof. Many years later, C. Schensted independently rediscovered the correspondence, which he stated in essentially the fashion we have used [*Canadian J. Math.* **13** (1961), 179–191]. He also proved the "P" part of Theorem D(a). M. P. Schützenberger [*Math. Scand.* **12** (1963), 117–128] proved Theorem B and the "Q" part of Theorem D(a), from which (b) and (c) follow. It is possible to extend the correspondence to permutations of *multisets;* the case that $p_1, \ldots, p_n$ need not be distinct was considered by Schensted, and the "ultimate" generalization to the case that both the p's and the q's may contain repeated elements was investigated by Knuth [*Pacific J. Math.* **34** (1970), 709–727].

Let us now turn to a related question: *How many tableaux formed from*

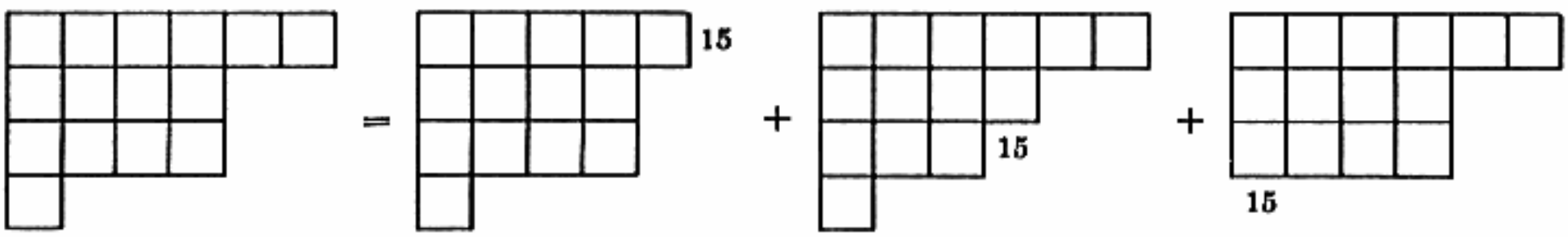
$\{1, 2, \ldots, n\}$ *have a given shape* $(n_1, n_2, \ldots, n_m)$, *where* $n_1 + n_2 + \cdots + n_m = n$? If we denote this number by $f(n_1, n_2, \ldots, n_m)$, it must satisfy the relations

$$f(n_1, n_2, \ldots, n_m) = 0 \qquad \text{unless} \qquad n_1 \geq n_2 \geq \cdots \geq n_m \geq 0; \tag{30}$$

$$f(n_1, n_2, \ldots, n_m, 0) = f(n_1, n_2, \ldots, n_m); \tag{31}$$

$$f(n_1, n_2, \ldots, n_m) = f(n_1 - 1, n_2, \ldots, n_m) + f(n_1, n_2 - 1, \ldots, n_m) + \cdots + f(n_1, n_2, \ldots, n_m - 1), \quad \text{if} \quad n_1 \geq n_2 \geq \cdots \geq n_m \geq 1. \tag{32}$$

The latter recurrence comes from the fact that removing the largest element of a tableau always leaves another tableau; for example, the number of tableaux of shape (6, 4, 4, 1) is $f(5, 4, 4, 1) + f(6, 3, 4, 1) + f(6, 4, 3, 1) + f(6, 4, 4, 0) = f(5, 4, 4, 1) + f(6, 4, 3, 1) + f(6, 4, 4)$, since every tableau of shape (6, 4, 4, 1) on $\{1, 2, \ldots, 15\}$ is formed by inserting the element 15 into the appropriate place in a tableau of shape (5, 4, 4, 1), (6, 4, 3, 1), or (6, 4, 4). Schematically,



(33)

The function $f(n_1, n_2, \ldots, n_m)$ which satisfies these relations has a fairly simple form,

$$f(n_1, n_2, \ldots, n_m) = \frac{\Delta(n_1 + m - 1, n_2 + m - 2, \ldots, n_m) n!}{(n_1 + m - 1)!(n_2 + m - 2)! \ldots m_m!}, \quad \text{for} \quad n_1 + m - 1 \geq n_2 + m - 2 \geq \cdots \geq n_m, \tag{34}$$

where Δ denotes the "discriminant" function

$$\Delta(x_1, x_2, \ldots, x_m) = \det \begin{pmatrix} x_1^{m-1} & x_2^{m-1} & \ldots & x_m^{m-1} \\ \vdots & & & \vdots \\ x_1^2 & x_2^2 & & x_m^2 \\ x_1 & x_2 & & x_m \\ 1 & 1 & \ldots & 1 \end{pmatrix} = \prod_{1 \leq i < j \leq m} (x_i - x_j). \tag{35}$$

Formula (34) was derived by G. Frobenius [*Sitzungsberichte Preuss. Akad. der Wissenchaften* (Berlin, 1900), 516–534, Sec. 3], in connection with an equivalent

problem in group theory, using a rather deep group-theoretical argument; a combinatorial proof was independently given by MacMahon [*Philosophical Trans.* **A-209** (London, 1909), 153–175]. The formula can be established by induction, since relations (30) and (31) are readily proved and (32) follows by setting $y = -1$ in the identity of exercise 17.

Theorem A gives a remarkable identity in connection with this formula for the number of tableaux. If we sum over all shapes, we have

$$n! = \sum_{\substack{k_1 \geq \cdots \geq k_n \geq 0 \\ k_1 + \cdots + k_n = n}} f(k_1, \ldots, k_n)^2 = n!^2 \sum_{\substack{k_1 \geq \cdots \geq k_n \geq 0 \\ k_1 + \cdots + k_n = n}} \frac{\Delta(k_1 + n - 1, \ldots, k_n)^2}{(k_1 + n - 1)!^2 \ldots k_n!^2}$$

$$= n!^2 \sum_{\substack{q_1 > q_2 > \cdots > q_n \geq 0 \\ q_1 + \cdots + q_n = (n+1)n/2}} \frac{\Delta(q_1, \ldots, q_n)^2}{q_1!^2 \ldots q_n!^2};$$

hence

$$\sum_{\substack{q_1 + \cdots + q_n = (n+1)n/2 \\ q_1, \ldots, q_n \geq 0}} \frac{\Delta(q_1, \ldots, q_n)^2}{q_1!^2 \ldots q_n!^2} = 1. \tag{36}$$

The inequalities $q_1 > q_2 > \cdots > q_n$ have been removed in the latter sum, since the summand is a symmetric function of the q's which vanishes when $q_i = q_j$. A similar identity appears in exercise 24.

The formula for the number of tableaux can be expressed in a somewhat more interesting manner, based on the idea of "hooks." The *hook* corresponding to a cell in a tableau is defined to be the cell itself plus the cells lying below and to its right. For example, the shaded area in Fig. 5 is the hook corresponding to cell (2, 3) in row 2, column 3; it contains six cells. Each cell of Fig. 5 has been filled in with the length of its hook.

12	11	8	7	5	4	1
10	9	**6**	**5**	**3**	**2**	•
9	8	**5**	4	2	1	•
6	5	**2**	1	•		
3	2	•				
2	1	•				

Fig. 5. Hooks and hook lengths.

If the shape of the tableau is $(n_1, n_2, \ldots, n_m)$, with $n_m \geq 1$, the longest hook has length $n_1 + m - 1$. Further examination of the hook lengths shows that row 1 contains all the lengths $n_1 + m - 1, n_1 + m - 2, \ldots, 1$ *except* for $(n_1 + m - 1) - (n_m)$, $(n_1 + m - 1) - (n_{m-1} + 1), \ldots, (n_1 + m - 1) - (n_2 + m - 2)$. In Fig. 5, for example, the hook lengths in row 1 are 12, 11,

10, . . . , 1 except for 10, 9, 6, 3, 2; the exceptions correspond to five nonexistent hooks, from nonexistent cells (6, 3), (5, 3), (4, 5), (3, 7), (2, 7) leading up to cell (1, 7). Similarly, row j contains all hook lengths $n_j + m - j, \ldots, 1$, except for $(n_j - m - j) - (n_m), \ldots, (n_j - m - j) - (n_{j+1} - m - j - 1)$. It follows that the product of all the hook lengths is equal to

$$(n_1 + m - 1)! \ldots (n_m)!/\Delta(n_1 + m - 1, \ldots, n_m).$$

This is just what happens in Eq. (34), so we have

Theorem H (J. S. Frame, G. de B. Robinson, R. M. Thrall). *The number of tableaux on* $\{1, 2, \ldots, n\}$ *having a specified shape is* $n!$ *divided by the product of the hook lengths.* ∎

Since this is such a simple result, it deserves a simple proof; but (like most of the known facts about tableaux) no more direct proof is known. Each element of a tableau is the smallest in its hook; if we fill the tableau shape at random, the probability that cell (i, j) will contain the minimum element of the corresponding hook is the reciprocal of the hook length. Multiplying these probabilities over all i and j gives Theorem H; but this argument is fallacious, since the probabilities are far from independent. All known proofs of Theorem C are based on an uninspiring induction argument which doesn't really explain why the theorem is true (since it doesn't really use the properties of hooks).

Theorem H has an interesting connection with the enumeration of trees, which we considered in Chapter 2. We observed that binary trees with n nodes correspond to permutations that can be obtained with a stack, and that such permutations correspond to sequences $a_1\ a_2 \ldots a_{2n}$ of S's and X's, where the number of S's is never less than the number of X's as we read from left to right. (See exercises 2.3.1–6 and 2.2.1–3.) The latter sequences correspond in a natural way to tableaux of shape (n, n); we place in row 1 the indices i such that $a_i = \mathrm{S}$, and in row 2 we put those indices with $a_i = \mathrm{X}$. For example, the sequence

$$\mathrm{S\ S\ S\ X\ X\ S\ S\ X\ X\ S\ X\ X}$$

corresponds to the tableau

1	2	3	6	7	10
4	5	8	9	11	12

. (37)

The column constraint is satisfied in this tableau if and only if the number of S's never exceeds the number of X's from left to right. By Theorem H, the number of tableaux of shape (n, n) is

$$(2n)!/(n + 1)!n!,$$

so this is the number of binary trees (in agreement with Eq. 2.3.4.4–13). Further-

more, this argument solves the more general "ballot problem" considered in the answer to exercise 2.2.1–4, if we use tableaux of shape (n, m) for $n \geq m$. So Theorem H includes some rather complex enumeration problems as simple special cases.

Any tableau A of shape (n, n) on the elements $\{1, 2, \ldots, 2n\}$ corresponds to two tableaux (P, Q) of the same shape, in the following way suggested by MacMahon [*Combinatory Analysis* **1** (1915), 130–131]. Let P consist of the elements $\{1, \ldots, n\}$ as they appear in A; Q is formed by taking the remaining elements, rotating the configuration by 180°, and replacing $n + 1, n + 2, \ldots, 2n$ by $n, n - 1, \ldots, 1$, respectively. For example, (37) splits into

1	2	3	6
4	5		

and

		7	10
8	9	11	12

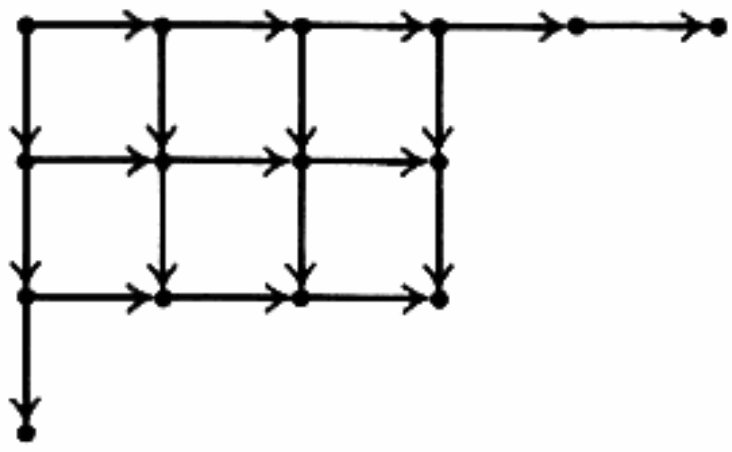
;

rotation and renaming of the latter yields

$$P = \begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 6 \\ \hline 4 & 5 \\ \cline{1-2}\end{array}, \qquad Q = \begin{array}{|c|c|c|c|}\hline 1 & 2 & 4 & 5 \\ \hline 3 & 6 \\ \cline{1-2}\end{array}. \tag{38}$$

Conversely, any pair of equal-shape tableaux of at most two rows, each containing n cells, corresponds in this way to a tableau of shape (n, n). Hence by exercise 7 *the number of permutations* $a_1\, a_2 \ldots a_n$ *of* $\{1, 2, \ldots, n\}$ *containing no decreasing subsequence* $a_i > a_j > a_k$ *for* $i < j < k$ *is the number of binary trees with* n *nodes.* Curiously there seems to be no apparent way to establish a correspondence between such permutations and binary trees, except for the roundabout method via Algorithm I that we have used here. It is interesting to compare this situation to the rather direct correspondence between binary trees and permutations having no instances of $a_j > a_k > a_i$ for $i < j < k$ (see exercise 2 2.1–5).

The number of ways to fill a tableau of shape (6, 4, 4, 1) is obviously the number of ways to put the labels $\{1, 2, \ldots, 15\}$ onto the vertices of the directed graph



(39)

in such a way that the label of vertex u is less than the label of vertex v whenever $u \longrightarrow v$. In other words, it is the number of ways to sort the partial ordering (39) topologically, in the sense of Section 2.2.3.

In general, we can ask the same question for any directed graph which contains no oriented cycles. It would be nice if there were some simple formula which would generalize Theorem H to the case of an arbitrary directed graph; but not all graphs have such pleasant properties as the graphs corresponding to tableaux. Some other classes of directed graphs for which the labelling problem has a simple solution are discussed in the exercises at the close of this section. There also are exercises which show that some directed graphs have *no* simple formula corresponding to Theorem H. For example, the number of ways to do the labelling is not always a divisor of $n!$.

To complete our investigations, let us count the total number of tableaux that can be formed from n distinct elements; we will denote this number by t_n. By the corollary to Theorem B, t_n is the number of involutions of $\{1, 2, \ldots, n\}$. A permutation is its own inverse if and only if its cycle form consists solely of one-cycles (fixed points) and two-cycles (transpositions). Since t_{n-1} of the t_n involutions have (n) as a one-cycle, and since t_{n-2} of them have $(j\ n)$ as a two-cycle, for fixed $j < n$, we obtain the formula

$$t_n = t_{n-1} + (n-1)t_{n-2} \tag{40}$$

which Rothe devised in 1800 to tabulate t_n for small n.

Counting another way, let us suppose that there are k two-cycles and $(n - 2k)$ one-cycles. There are $\binom{n}{2k}$ ways to choose the fixed points, and the multinomial coefficient $(2k)!/(2!)^k$ is the number of ways to arrange the other elements into k distinguishable transpositions; dividing by $k!$ to make the transpositions indistinguishable we therefore obtain

$$t_n = \sum_{k \geq 0} t_n(k), \qquad t_n(k) = \frac{n!}{(n-2k)!2^k k!}. \tag{41}$$

Unfortunately this sum cannot be further simplified, as far as anyone knows, so we resort to two indirect approaches in order to better understand t_n:

a) We can find the generating function

$$\sum_n t_n z^n/n! = e^{z+z^2/2}; \tag{42}$$

see exercise 25.

b) We can determine the asymptotic behavior of t_n. This is an instructive problem, because it involves some general techniques that will be useful to us in other connections, so we will conclude this section with an analysis of the asymptotic behavior of t_n.

The first step in analyzing the asymptotic behavior of (41) is to locate the main contribution to the sum. Since

$$\frac{t_n(k+1)}{t_n(k)} = \frac{(n-2k)(n-2k-1)}{2(k+1)}, \tag{43}$$

we can see that the terms gradually increase from $k = 0$ until k is approximately $\frac{1}{2}(n - \sqrt{n})$, then they decrease to zero when k is approximately $\frac{1}{2}n$. The main contribution clearly comes from the vicinity of $k = \frac{1}{2}(n - \sqrt{n})$. It is usually preferable to have the main contribution at the value 0, so we write

$$k = \tfrac{1}{2}(n - \sqrt{n}) + x, \tag{44}$$

and we will investigate the size of $t_n(k)$ as a function of x.

One useful way to get rid of the factorials in $t_n(k)$ is to use Stirling's approximation, Eq. 1.2.11.2–18. For this purpose it is convenient (as we shall see in a moment) to restrict x to the range

$$-(n^{\epsilon+1/4}) \le x \le n^{\epsilon+1/4}, \tag{45}$$

where $\epsilon = 0.001$, say, so that an error term can be included. A somewhat laborious calculation which should really have been done by computer yields the formula

$$\begin{aligned} t_n(k) = \exp\big(&\tfrac{1}{2}n\ln n - \tfrac{1}{2}n + \sqrt{n} - \tfrac{1}{4}\ln n - 2x^2/\sqrt{n} - \tfrac{1}{4} - \tfrac{1}{2}\ln\pi \\ &- \tfrac{4}{3}x^3/n + 2x/\sqrt{n} + \tfrac{1}{3}/\sqrt{n} - \tfrac{4}{3}x^4/n\sqrt{n} + O(n^{5\epsilon-3/4})\big). \end{aligned} \tag{46}$$

The restriction on x in (45) can be justified by the fact that we may set $x = \pm n^{\epsilon+1/4}$ to get an upper bound for all of the discarded terms, namely

$$e^{-2n^{2\epsilon}} \cdot \exp\big(\tfrac{1}{2}n\ln n - \tfrac{1}{2}n + \sqrt{n} - \tfrac{1}{4}\ln n - \tfrac{1}{4} - \tfrac{1}{2}\ln\pi + O(n^{3\epsilon-1/4})\big), \tag{47}$$

and if we multiply this by n we get an upper bound for the sum of the excluded terms. The upper bound is of lesser order than the terms we will compute for x in the restricted range (45), because of the factor $\exp(-2n^{2\epsilon})$ which is much less than any polynomial in n.

We can evidently remove the factor

$$\exp\big(\tfrac{1}{2}n\ln n - \tfrac{1}{2}n + \sqrt{n} - \tfrac{1}{4}\ln n - \tfrac{1}{4} - \tfrac{1}{2}\ln\pi + \tfrac{1}{3}/\sqrt{n}\big) \tag{48}$$

from the sum, and this leaves us with the task of summing

$$\begin{aligned} &\exp\big(-2x^2/\sqrt{n} - \tfrac{4}{3}x^3/n + 2x/\sqrt{n} - \tfrac{4}{3}x^4/n\sqrt{n} + O(n^{5\epsilon-3/4})\big) \\ &\quad = \exp\left(\frac{-2x^2}{\sqrt{n}}\right)\cdot\left(1 - \frac{4}{3}\,\frac{x^3}{n} + \frac{8}{9}\,\frac{x^6}{n^2}\right) \\ &\qquad \cdot\left(1 + 2\frac{x}{\sqrt{n}} + 2\frac{x^2}{n}\right)\cdot\left(1 - \frac{4}{3}\,\frac{x^4}{n\sqrt{n}}\right)\cdot(1 + O(n^{9\epsilon-3/4})) \end{aligned} \tag{49}$$

over the range $x = \alpha, \alpha + 1, \ldots, \beta - 2, \beta - 1$ where $-\alpha$ and β (not necessarily integers) are approximately equal to $n^{t+1/4}$. Euler's summation formula, Eq. 1.2.11.2–10, can be written

$$\sum_{\alpha \le x < \beta} f(x) = \int_\alpha^\beta f(x)\,dx - \tfrac{1}{2}f(x)\Big|_\alpha^\beta + \tfrac{1}{2}B_2 \cdot \frac{f'(x)}{1!}\Big|_\alpha^\beta + \cdots + \frac{1}{m+1}B_{m+1} \cdot \frac{f^{(m)}(x)}{m!}\Big|_\alpha^\beta + R_{m+1}, \quad (50)$$

by translation of the interval of summation. Here $|R_m| \le (4/(2\pi)^m)\int_\alpha^\beta |f^{(m)}(x)|dx$. If we let $f(x) = x^t \exp(-2x^2/\sqrt{n})$, where t is a fixed nonnegative integer, we find that $\int_\alpha^\beta f(x)\,dx$ differs from $\int_{-\infty}^{\infty} f(x)\,dx$ by $O(\exp(-2n^t))$, so we may take $\alpha = -\infty$, $\beta = \infty$. Euler's summation formula will give an asymptotic series for $\sum f(x)$ as $n \to \infty$, in this case, since

$$f^{(m)}(x) = n^{(t-m)/4} g^{(m)}(n^{-1/4}x), \qquad g(y) = y^t e^{-2y^2}, \quad (51)$$

and $g(y)$ is a well-behaved function independent of n; this shows that $R_m = O(n^{(t+1-m)/4})$. Since $f^{(m)}(-\infty) = f^{(m)}(\infty) = 0$, we have proved that

$$\sum_{\alpha \le x < \beta} f(x) = \int_{-\infty}^{\infty} f(x)\,dx + O(n^{-m}), \qquad \text{for all} \qquad m \ge 0; \quad (52)$$

only the integral is really significant, given this particular choice of $f(x)$! The integral is not difficult to evaluate (see exercise 26), so we can multiply out and sum formula (49), giving

$$\sqrt{\frac{\pi}{2}}\, n^{1/4}(1 - \tfrac{1}{24}n^{-1/4} + O(n^{-1/2}));$$

$$t_n = \frac{1}{\sqrt{2}}\, n^{n/2} e^{-n/2+\sqrt{n}-1/4}(1 + \tfrac{7}{24}n^{-1/2} + O(n^{-3/4})). \quad (53)$$

In principle, this method could be extended to obtain $O(n^{-k})$ for any k, instead of $O(n^{-3/4})$. This asymptotic series for t_n was first determined (using a different method) by Moser and Wyman, *Canadian J. Math.* **7** (1955), 159–168. For further examples and extensions of the technique used to derive (53), see the conclusion of Section 5.2.2.

EXERCISES

1. [*16*] What tableaux (P, Q) correspond to the two-line array

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 6 & 4 & 9 & 5 & 7 & 1 & 2 & 8 & 3 \end{pmatrix},$$

in the construction of Theorem A? What two-line array corresponds to the tableaux

$$P = \begin{array}{|c|c|c|}\hline 1 & 4 & 7 \\ \hline 2 & 8 \\ \cline{1-2} 5 & 9 \\ \cline{1-2}\end{array}\,, \qquad Q = \begin{array}{|c|c|c|}\hline 1 & 3 & 7 \\ \hline 4 & 5 \\ \cline{1-2} 8 & 9 \\ \cline{1-2}\end{array}\,?$$

2. [*M21*] Prove that (q, p) belongs to class t with respect to (16) if and only if t is the largest number of indices $i_1, \ldots, i_t$ such that

$$p_{i_1} < p_{i_2} < \cdots < p_{i_t} = p, \qquad q_{i_1} < q_{i_2} < \cdots < q_{i_t} = q.$$

▸ **3.** [*M24*] Show that the correspondence defined in the proof of Theorem A can also be carried out by constructing a table such as this:

Line 0	1	3	5	6	8
Line 1	7	2	9	5	3
Line 2	∞	7	∞	9	5
Line 3		∞		∞	7
Line 4					∞

Here lines 0 and 1 constitute the given two-line array. For $k \geq 1$, line $k+1$ is formed from line k by the following procedure:

a) Set $p \leftarrow \infty$.
b) Let column j be the leftmost column in which line k contains an integer $< p$, but line $k+1$ is blank. If no such columns exist, and if $p = \infty$, line $k+1$ is complete; if no such columns exist and $p < \infty$, return to (a).
c) Insert p into column j in line $k+1$, then set p equal to the entry in column j of line k and return to (b).

Once the table has been constructed in this way, row k of P consists of those integers in line k which are not in line $(k+1)$; row k of Q consists of those integers in line 0 which appear in a column containing ∞ in line $k+1$.

4. [*M26*] (M. P. Schützenberger.) Let π be an involution with k fixed points. Prove that the tableau corresponding to π, in the proof of the corollary to Theorem B, has exactly k columns of odd length.

▸ **5.** [*M30*] Let $a_1 \ldots a_{j-1}\, a_j \ldots a_n$ be a permutation of distinct elements, and assume that $1 < j \leq n$. The permutation $a_1 \ldots a_{j-2}\, a_j\, a_{j-1}\, a_{j+1} \ldots a_n$, obtained by interchanging a_{j-1} with a_j, is called "admissible" if either

i) $j \geq 3$ and a_{j-2} lies between a_{j-1} and a_j; or
ii) $j < n$ and a_{j+1} lies between a_{j-1} and a_j.

For example, exactly three admissible interchanges can be performed on the permutation 1 5 4 6 8 3 7; we can interchange the 1 and the 5 since $1 < 4 < 5$; we can

interchange the 8 and the 3 since $3 < 6 < 8$ (or since $3 < 7 < 8$); but we cannot interchange the 5 and the 4, or the 3 and the 7.

a) Prove that an admissible interchange does not change the tableau P formed from the permutation by successive insertion of the elements $a_1, a_2, \ldots, a_n$ into an initially empty tableau.

b) Conversely, prove that any two permutations which have the same P tableau can be transformed into each other by a sequence of one or more admissible interchanges. [*Hint:* Given that the shape of P is $(n_1, n_2, \ldots, n_m)$, show that any permutation which corresponds to P can be transformed into the "canonical permutation" $P_{m1} \ldots P_{mn_m} \ldots P_{21} \ldots P_{2n_2} P_{11} \ldots P_{1n_1}$ by a sequence of admissible interchanges.]

▸ **6.** [*M22*] Let P be the tableau corresponding to the permutation $a_1\, a_2 \ldots a_n$; use exercise 5 to prove that P^T is the tableau corresponding to $a_n \ldots a_2\, a_1$.

7. [*M20*] (C. Schensted.) Let P be the tableau corresponding to the permutation $a_1\, a_2 \ldots a_n$. Prove that the number of *columns* in P is the longest length c of an increasing subsequence $a_{i_1} < a_{i_2} < \cdots < a_{i_c}$, where $i_1 < i_2 < \cdots < i_c$; the number of *rows* in P is the longest length r of a decreasing subsequence $a_{j_1} > a_{j_2} > \cdots > a_{j_r}$ where $j_1 < j_2 < \cdots < j_r$.

8. [*M18*] (P. Erdös, G. Szekeres.) Prove that any permutation containing more than n^2 elements has a monotonic subsequence of length greater than n; but there are permutations of n^2 elements with no monotonic subsequences of length greater than n. [*Hint:* See the previous exercise.]

9. [*M24*] Continuing exercise 8, find a "simple" formula for the exact number of permutations of $\{1, 2, \ldots, n^2\}$ having no monotonic subsequences of length greater than n.

10. [*M20*] Prove that P is a tableau when Algorithm S terminates, if it was a tableau initially.

11. [*20*] Given only the values of r and s after Algorithm S terminates, is it possible to restore P to its original condition?

12. [*M24*] How many times is step S3 performed, if Algorithm S is used repeatedly to delete all elements of a tableau P whose shape is $(n_1, n_2, \ldots, n_m)$? What is the minimum of this quantity, taken over all shapes with $n_1 + n_2 + \cdots + n_m = n$?

13. [*M28*] Prove Theorem C.

14. [*M43*] Find a more direct proof of Theorem D, part (c).

15. [*M20*] How many permutations of the multiset $\{l \cdot a, m \cdot b, n \cdot c\}$ have the property that, as we read the permutation from left to right, the number of c's never exceeds the number of b's, and the number of b's never exceeds the number of a's? (For example, *a a b c a b b c a c a* is such a permutation.)

16. [*M08*] In how many ways can the partial ordering represented by (39) be sorted topologically?

17. [*HM25*] Let

$$g(x_1, x_2, \ldots, x_n; y) = x_1\, \Delta(x_1 + y, x_2, \ldots, x_n) + x_2\, \Delta(x_1, x_2 + y, \ldots, x_n) + \cdots + x_n\, \Delta(x_1, x_2, \ldots, x_n + y).$$

Prove that

$$g(x_1, x_2, \ldots, x_n; y) = \left(x_1 + x_2 + \cdots + x_n + \binom{n}{2} y\right) \Delta(x_1, x_2, \ldots, x_n).$$

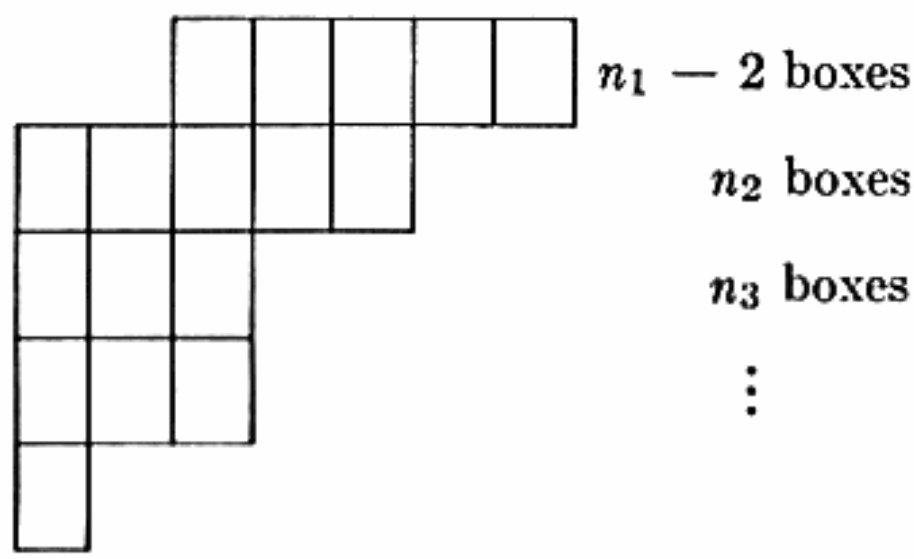
[*Hint:* The polynomial g is homogeneous (all terms have the same total degree); and it is antisymmetric in the x's (interchanging x_i and x_j changes the sign of g).]

18. [*HM30*] Generalizing exercise 17, evaluate the sum

$$x_1^m \Delta(x_1 + y, x_2, \ldots, x_n) + x_2^m \Delta(x_1, x_2 + y, \ldots, x_n) + \cdots + x_n^m \Delta(x_1, x_2, \ldots, x_n + y),$$

when $m \geq 0$.

19. [*M40*] Find a formula for the number of ways to fill an array which is like a tableau but with two boxes removed at the left of row 1; for example,



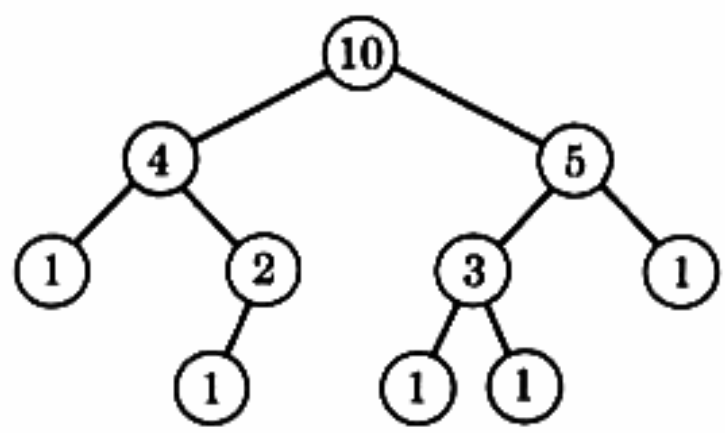
is such a shape. (The rows and columns are to be in increasing order, as in ordinary tableaux.)

In other words, how many of the $f(n_1, n_2, \ldots, n_m)$ tableaux of shape

$$(n_1, n_2, \ldots, n_m)$$

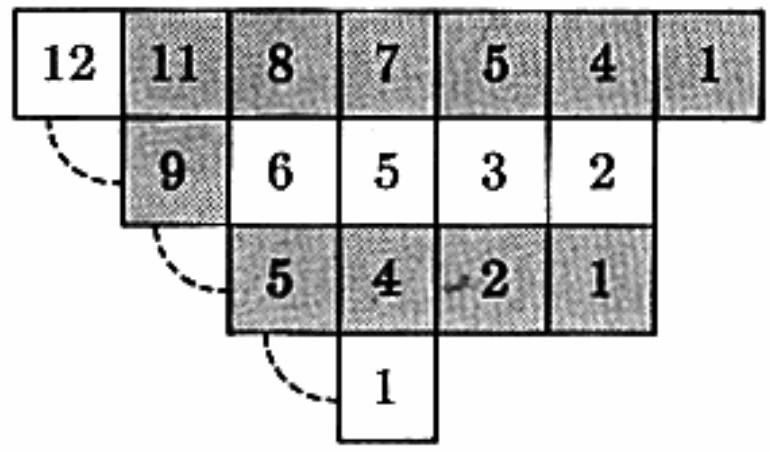
on $\{1, 2, \ldots, n_1 + \cdots + n_m\}$ have both of the elements 1 and 2 in the first row?

▸ **20.** [*M24*] Prove that the number of ways to label the nodes of a given binary tree with the elements $\{1, 2, \ldots, n\}$, such that the label of each node is less than that of its descendants, is $n!$ divided by the product of the "subtree lengths," i.e., the number of nodes in each subtree. (Cf. Theorem H.) For example, the number of ways to label the nodes of



is $10! / 10 \cdot 4 \cdot 5 \cdot 1 \cdot 2 \cdot 3 \cdot 1 \cdot 1 \cdot 1 \cdot 1 = 9 \cdot 8 \cdot 7 \cdot 6$.

21. [*HM31*] (R. M. Thrall.) Let $n_1 > n_2 > \cdots > n_m$ specify the shape of a "shifting tableau" where row $i+1$ starts one position to the right of row i; for example, a shifting tableau of shape (7, 5, 4, 1) has the form of the diagram

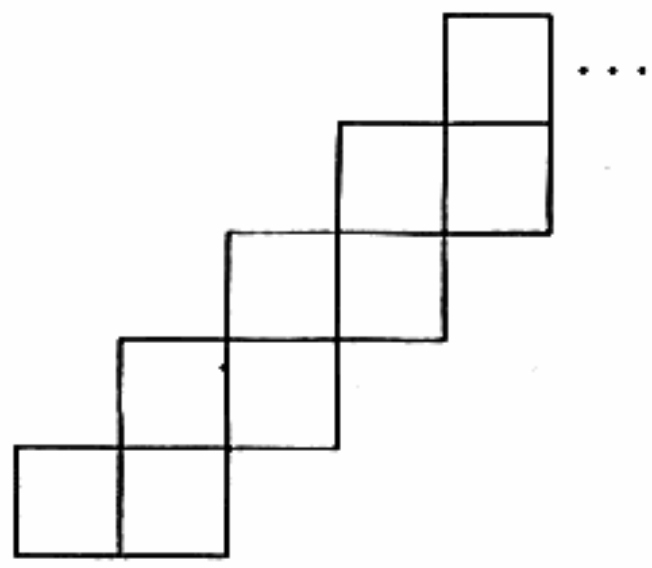


Prove that the number of ways to put the integers $1, 2, \ldots, n = n_1 + n_2 + \cdots + n_m$ into shifting tableaux of shape $(n_1, n_2, \ldots, n_m)$, so that rows and columns are in increasing order, is $n!$ divided by the product of the "generalized hook lengths"; a generalized hook of length 11, corresponding to the cell in row 1 column 2, has been shaded in the above diagram. (Hooks in the "inverted staircase" portion of the array, at the left, have a U-shape, tilted 90°, instead of an L-shape.) Thus there are

$$17!/12\cdot 11\cdot 8\cdot 7\cdot 5\cdot 4\cdot 1\cdot 9\cdot 6\cdot 5\cdot 3\cdot 2\cdot 5\cdot 4\cdot 2\cdot 1\cdot 1$$

ways to fill the above shape with rows and columns in increasing order.

▸ **22.** [*HM30*] (D. André.) In how many ways, A_n, can the numbers $\{1, 2, \ldots, n\}$ be placed into the array of n cells



in such a way that the rows and columns are in increasing order? Find the generating function $g(z) = \sum A_n z^n/n!$.

23. [*M39*] In how many ways can an array of shape $(n_1, n_2, \ldots, n_m)$ be filled with elements from the set $\{1, 2, \ldots, N\}$ *with repetitions allowed*, so that the rows are nondecreasing and the columns are strictly increasing? For example, the simple m-rowed shape $(1, 1, \ldots, 1)$ can be filled in $\binom{N}{m}$ ways; the 1-rowed shape (m) can be filled in $\binom{m+N-1}{m}$ ways; the shape (2, 2) in $\frac{1}{3}\binom{N+1}{2}\binom{N}{2}$ ways.

24. [*M28*] Prove that

$$\sum_{\substack{q_1+\cdots+q_n=t\\ 0\le q_1,\ldots,q_n\le m}} \binom{m}{q_1}\cdots\binom{m}{q_n}\Delta(q_1,\ldots,q_n)^2$$

$$= n!\binom{nm-(n^2-n)}{t-\frac{1}{2}(n^2-n)}\binom{m}{n-1}\binom{m}{n-2}\cdots\binom{m}{0}\Delta(n-1,\ldots,0)^2.$$

[*Hints:* Prove that $\Delta(k_1+n-1,\ldots,k_n)=\Delta(m-k_n+n-1,\ldots,m-k_1)$; decompose an $n\times(m-n+1)$ tableau in a fashion analogous to (38); and manipulate the sum as in the derivation of (36).]

25. [*M20*] Why is (42) the generating function for involutions?

26. [*HM21*] Evaluate $\int_{-\infty}^{\infty} x^t \exp(-2x^2/\sqrt{n})\,dx$ when t is a nonnegative integer.

27. [*M24*] Let Q be a Young tableau on $\{1, 2, \ldots, n\}$; let the element i be in row r_i and column c_i. We say that i is "above" j when $r_i < r_j$.

a) Prove that, for $1 \le i < n$, i is above $i+1$ if and only if $c_i \ge c_{i+1}$.

b) Given that Q is such that (P, Q) corresponds to the permutation

$$\begin{pmatrix} 1 & 2 & \ldots & n \\ a_1 & a_2 & \ldots & a_n \end{pmatrix},$$

prove that i is above $i+1$ if and only if $a_i > a_{i+1}$. (Therefore we can determine the number of runs in the permutation, knowing only Q. This result is due to M. P. Schützenberger.)

c) Prove that, for $1 \le i < n$, i is above $i+1$ in Q if and only if $i+1$ is above i in Q^s.

28. [*M47*] What is the asymptotic behavior of the average length of the longest increasing sequence of a random permutation on $\{1, 2, \ldots, n\}$? (This is the average length of row 1 in the correspondence of Theorem A. Extensive tables computed by R. M. Baer and P. Brock, *Math. Comp.* **22** (1968), 385–410, in connection with what they call "natural sorting" for some obscure reason, suggest that the average is approximately $2\sqrt{n}$; the data is also consistent with other hypotheses such as $1.97\sqrt{n}$.)

29. [*M50*] Investigate three-dimensional arrays, in order to see how many of the properties of two-dimensional tableaux can be generalized.

30. [*M42*] (M. P. Schützenberger.) Show that the operation of going from P to P^s is a special case of an operation applicable in connection with *any* finite partially ordered set, not merely a tableau: Label the elements of a partially ordered set with the integers $\{1, 2, \ldots, n\}$ such that the partial order is consistent with the labeling. Find a dual labeling analogous to (26), by successively deleting the tables 1, 2, ... while moving the other labels in a fashion analogous to Algorithm S and placing (1), (2), ... in the vacated places. Show that this operation, when repeated on the dual labeling in reverse numerical order, yields the original labeling; and explore other properties of the operation.

31. [*HM30*] Let x_n be the number of ways to place n mutually nonattacking rooks on an $n\times n$ chessboard, where the arrangement is unchanged by reflection about one of the diagonals and by 180° rotation. Find the asymptotic behavior of x_n.

Let's begin our discussion of good "sortmanship" by conducting a little experiment. How would you solve the following programming problem?

> "Memory locations R+1, R+2, R+3, R+4, and R+5 contain five numbers. Write a computer program which rearranges these numbers, if necessary, so that they are in ascending order."

(If you already are familiar with some sorting methods, please do your best to forget about them momentarily; imagine that you are attacking this problem for the first time, without any prior knowledge of how to proceed.)

Before reading any further, you are requested to construct a solution to this problem.

. . . . . .

The time you spent working on the above problem will pay dividends as you continue to read this chapter. Chances are your solution is one of the following types:

A. *An insertion sort.* The items are considered one at a time, and each new item is inserted into the appropriate position relative to the previously-sorted items. (This is the way many bridge players sort their hands, picking up one card at a time.)

B. *An exchange sort.* If two items are found to be out of order, they are interchanged. This process is repeated until no more exchanges are necessary.

C. *A selection sort.* First the smallest (or perhaps the largest) item is located, and it is somehow separated from the rest; then the next smallest (or next largest) is selected, and so on.

D. *An enumeration sort.* Each item is compared with each of the others; counting the number of smaller keys determines the item's final position.

E. *A special-purpose sort,* which works nicely for sorting five elements as stated in the problem, but does not readily generalize to larger numbers of items.

F. *A lazy attitude,* with which you ignored the suggestion above and decided not to solve the problem at all. Sorry, by now you have read too far and you have lost your chance.

G. *A new, super sorting technique* which is a definite improvement over known methods. (Please communicate this to the author at once.)

If the problem had been posed for, say, 1000 items, not merely 5, you may also have discovered some of the more subtle techniques which will be mentioned later. At any rate, when attacking a new problem it is often wise to find some fairly obvious procedure which works, and then try to improve upon it. Cases A, B, and C above lead to important classes of sorting techniques which are refinements of the simple ideas stated.

Many different sorting algorithms have been invented, and we will be discussing about 25 of them in this book. This rather alarming number of methods is actually only a fraction of the algorithms that have been devised so far; many methods which are now obsolete will be omitted from our discussion, or mentioned only briefly. Why are there so many sorting methods? For computer programming, this is a special case of the question, "Why are there so many x methods?", where x ranges over the set of problems; and the answer is that each method has its own advantages and disadvantages, so that it outperforms the others on some configurations of data and hardware. Unfortunately, there is no known "best" way to sort; there are *many* best methods, depending on what is to be sorted on what machine for what purpose. In the words of Rudyard Kipling, "There are nine and sixty ways of constructing tribal lays, and every single one of them is right."

It is a good idea to learn the characteristics of each sorting method, so that an intelligent choice can be made for particular applications. Fortunately, it is not a formidable task to learn these algorithms, since they are interrelated in interesting ways.

At the beginning of this chapter we defined the basic terminology and notation to be used in our study of sorting: The records

$$R_1, R_2, \ldots, R_N \tag{1}$$

are to be sorted into nondecreasing order of their keys $K_1, K_2, \ldots, K_N$, essentially by discovering a permutation $p(1)\ p(2) \ldots p(N)$ such that

$$K_{p(1)} \le K_{p(2)} \le \cdots \le K_{p(N)}. \tag{2}$$

In the present section we are concerned with *internal sorting*, when the number of records to be sorted is small enough that the entire process can be performed in a computer's high-speed memory.

In some cases we will want the records to be physically rearranged in memory so that their keys are in order, while in other cases it may be sufficient merely to have an auxiliary table of some sort which specifies the permutation. If the records and/or the keys each take up quite a few words of computer memory, it is often better to make up a new table of link addresses which point to the records, and to manipulate these link addresses instead of moving the bulky records around. This method is called *address table sorting* (see Fig. 6). If the key is short but the satellite information of the records is long, the key may be placed with the link addresses for greater speed; this is called *key sorting*. Other sorting schemes utilize an auxiliary link field which is included in each record; these links are manipulated in such a way that, in the final result, the records are linked together to form a straight linear list, with each link pointing to the following record. This is called *list sorting* (see Fig. 7).

After sorting with an address table or list method, the records can be rearranged into increasing order as desired. There are several ways to do this,

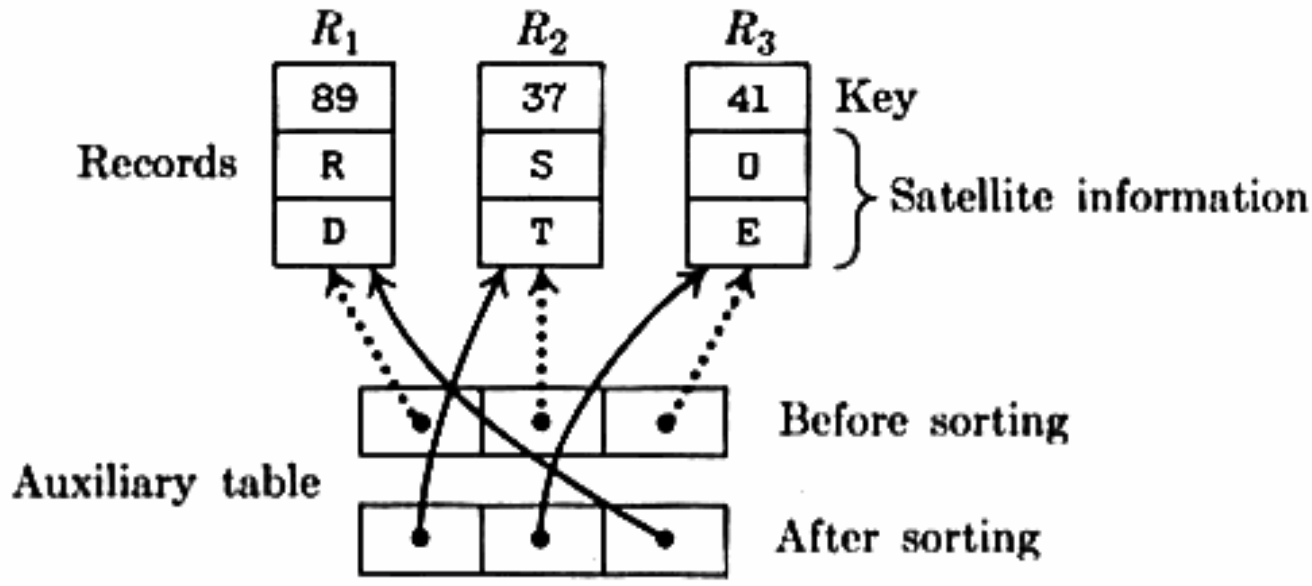


Fig. 6. Address table sorting.

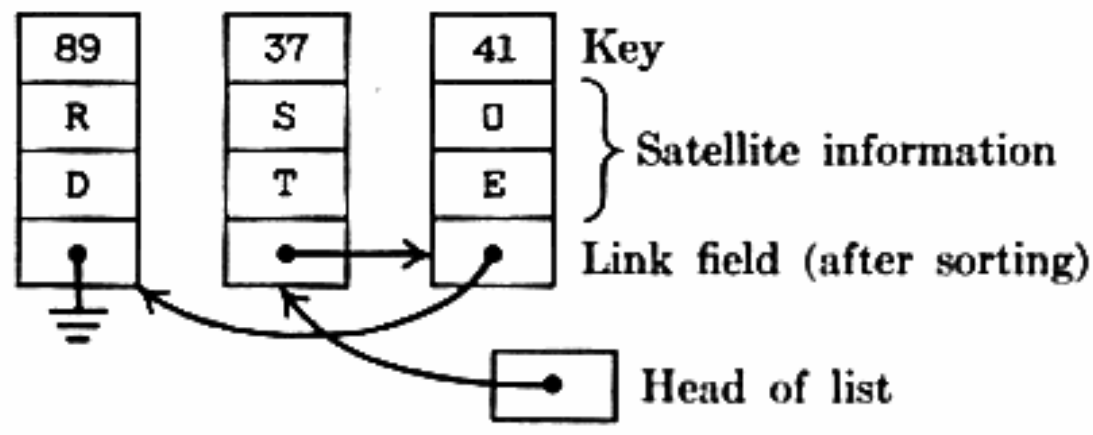


Fig. 7. List sorting.

requiring only enough additional memory space to hold one record (see exercises 10 through 12); alternatively, we can simply move the records into a new area capable of holding all records. The latter method is usually about twice as fast as the former, but it demands nearly twice as much storage space. It is unnecessary to move the records at all, in many applications, since the link fields are often adequate for subsequent addressing operations.

All of the sorting methods which we shall examine "in depth" will be illustrated in four ways, by means of

a) an English-language description of the algorithm,

b) a flow diagram,

c) a MIX program, and

d) an example of the sorting method applied to a given set of numbers.

[Wherever appropriate, the latter example will deal with a certain set of 16 numbers which were chosen at random by the author on March 19, 1963, using a set of decimal dice; cf. exercise 3.1–1(c).]

For convenience, the MIX programs will usually be given under the assumption that the key is numeric and that it fits in a single word; sometimes we will even restrict the key to part of a word. The order relation $<$ will be ordinary arithmetic order; and the record will consist of the key alone, with no satellite information. These assumptions make the programs shorter and easier to understand, and it should be clear how to modify the programs (e.g., by

using address table sorting) for the general case. An analysis of the running time of each sorting algorithm will be given with the MIX programs.

Sorting by counting. As a simple example of the way in which we shall study internal sorting methods, let us consider the "counting" idea mentioned near the beginning of this section. This simple method is based on the idea that the jth key in the final sorted sequence is greater than exactly $(j - 1)$ of the other keys. Putting this another way, if we know that a certain key exceeds exactly 27 others, the corresponding record should go into position 28 after sorting. So the idea is to compare each pair of keys, counting how many are less than each particular one.

The obvious way to do the comparisons is to

$$\big((\text{compare } K_j \text{ with } K_i) \text{ for } 1 \le j \le N\big) \text{ for } 1 \le i \le N;$$

but it is easy to see that over half of these comparisons are redundant, since it is unnecessary to compare a key with itself, and it is unnecessary to compare K_a with K_b and later to compare K_b with K_a. We need merely to

$$\big((\text{compare } K_j \text{ with } K_i) \text{ for } 1 \le j \le i\big) \text{ for } 1 < i \le N.$$

Hence we are led to the following algorithm.

Algorithm C (*Comparison counting*). This algorithm sorts $R_1, \ldots, R_N$ on the keys $K_1, \ldots, K_N$ by maintaining an auxiliary table COUNT[1], ..., COUNT[N] to count the number of keys less than a given key. After the conclusion of the algorithm, COUNT[j] + 1 specifies the final position of record R_j.

C1. [Clear COUNTs.] Set COUNT[1] through COUNT[N] to zero.

C2. [Loop on i.] Perform step C3, for $i = N, N - 1, \ldots, 2$; then terminate the algorithm.

C3. [Loop on j.] Perform step C4, for $j = i - 1, i - 2, \ldots, 1$.

C4. [Compare K_i, K_j.] If $K_i < K_j$, increase COUNT[j] by 1; otherwise increase COUNT[i] by 1. ∎

Note that this algorithm involves no movement of records. It is similar to an address table sort, since the COUNT table specifies the final arrangement of records; but it is somewhat different because COUNT[j] tells us where to move R_j, instead of indicating which record should be moved into the place of R_j. (Thus the "inverse" of the permutation $p(1) \ldots p(n)$ is specified in the COUNT table; see Section 5.1.1.)

In our discussion preceding this algorithm we failed to consider the possibility of equal keys. This is a potentially serious omission, for if equal keys corresponded to equal COUNTs the final rearrangement of records would be rather complicated. Fortunately, as exercise 2 shows, Algorithm C gives the correct result no matter how many equal keys are present.

Table 1

SORTING BY COUNTING (ALGORITHM C)

KEYS:	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
COUNT(init.):	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
COUNT($i = N$):	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	12
COUNT($i = N - 1$):	0	0	0	0	2	0	2	0	0	0	0	0	0	0	13	12
COUNT($i = N - 2$):	0	0	0	0	3	0	3	0	0	0	0	0	0	11	13	12
COUNT($i = N - 3$):	0	0	0	0	4	0	4	0	1	0	0	0	9	11	13	12
COUNT($i = N - 4$):	0	0	1	0	5	0	5	0	2	0	0	7	9	11	13	12
COUNT($i = N - 5$):	1	0	2	0	6	1	6	1	3	1	2	7	9	11	13	12
. . . . .																
COUNT($i = 2$):	6	1	8	0	15	3	14	4	10	5	2	7	9	11	13	12

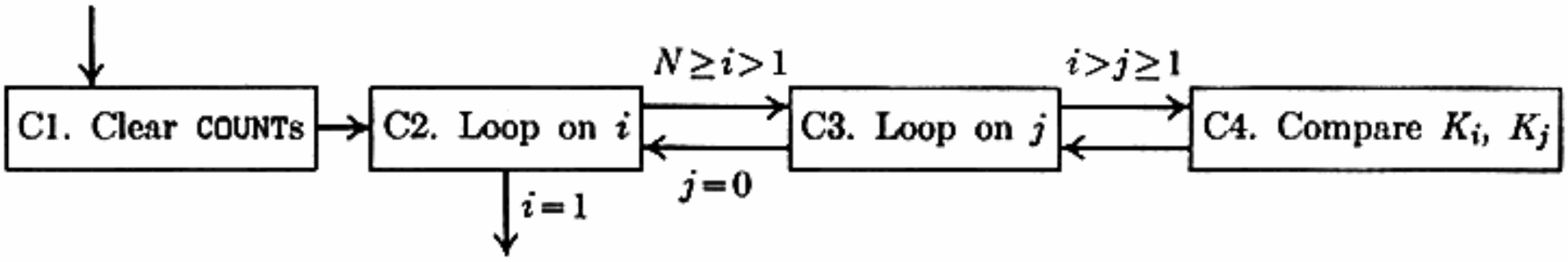


Fig. 8. Algorithm C: Comparison counting.

Program C (*Comparison counting*). The following MIX implementation of Algorithm C assumes that R_j is stored in location INPUT $+ j$, and COUNT[j] in location COUNT $+ j$, for $1 \leq j \leq N$; rI1 $\equiv i$; rI2 $\equiv j$; rA $\equiv K_i \equiv R_i$; rX $\equiv$ COUNT[i].

```
1  START  ENT1  N          1        C1. Clear COUNTs.
2         STZ   COUNT,1    N        COUNT[i] ← 0.
3         DEC1  1          N
4         J1P   *-2        N        N ≥ i > 0.
5         ENT1  N          1        C2. Loop on i.
6         JMP   1F         1
7  2H     LDA   INPUT,1    N - 1
8         LDX   COUNT,1    N - 1
9  3H     CMPA  INPUT,2      A      C4. Compare K_i, K_j.
10        JGE   4F           A      Jump if K_i ≥ K_j.
11        LD3   COUNT,2      B      COUNT[j]
12        INC3  1            B      + 1
13        ST3   COUNT,2      B      → COUNT[j].
14        JMP   5F           B
15 4H     INCX  1          A - B    COUNT[i] + 1 → COUNT[i].
16 5H     DEC2  1            A      C3. Loop on j.
17        J2P   3B           A
18        STX   COUNT,1    N - 1
19        DEC1  1          N - 1
20 1H     ENT2  -1,1         N      N ≥ i > j > 0.
21        J2P   2B           N      ∎
```

The running time of this program is $13N + 6A + 5B - 4$ units, where N is the number of records; A is the number of choices of two things from a set of N objects, namely $\binom{N}{2} = (N^2 - N)/2$; and B is the number of pairs of indices with $j < i$ and $K_j > K_i$. Thus, B is the number of *inversions* of the permutation $K_1, \ldots, K_N$; this is the quantity that was analyzed extensively in Section 5.1.1, where we found (Eqs. 5.1.1–12, 13) that, for unequal keys in random order,

$$B = (\min 0, \text{ave } (N^2 - N)/4, \max (N^2 - N)/2, \text{dev } \sqrt{N(N-1)(N+2.5)/6}).$$

Hence Program C requires between $3N^2 + 10N - 4$ and $5.5N^2 + 7.5N - 4$ units of time, and the average running time lies halfway between these two extremes. For example, the data in Table 1 has $N = 16$, $A = 120$, $B = 41$, so Program C will sort it in $1129u$. See exercise 5 for a modification of Program C which has slightly different timing characteristics.

The factor N^2 which dominates this running time shows that Algorithm C isn't an efficient way to sort when N is large; doubling the number of records increases the running time fourfold. Since the method requires a comparison of all distinct pairs of keys (K_i, K_j), there is no apparent way to get rid of the dependence on N^2, although we will see later in this chapter that the average running time can be reduced to order $N \log N$ using the "partition exchange" technique. Our main interest in Algorithm C is its simplicity, not its efficiency; it serves as an example of the style in which we will be describing more complex (and more efficient) methods.

There is another way to sort by counting which *is* quite important from the standpoint of efficiency; it is primarily applicable in the case that many equal keys are present, and when all keys fall into the range $u \le K_j \le v$, where $(v - u)$ is small. These assumptions appear to be quite restrictive, but in fact we shall see quite a few applications of the idea; for example, if we apply this method to the leading digits of keys instead of applying it to entire keys, the file will be partially sorted and it will be comparatively simple to complete the job.

In order to understand the principles involved, suppose that all keys lie between 1 and 100. In one pass through the file we can count how many 1's, 2's, . . . , 100's are present; and in a second pass we can move the records into the appropriate place in an output area. The following algorithm spells things out in more detail:

Algorithm D (*Distribution counting*). Assuming that all keys are integers in the range $u \le K_j \le v$ for $1 \le j \le N$, this algorithm sorts the records $R_1, \ldots, R_N$ by making use of an auxiliary table COUNT[u], . . . , COUNT[v]. At the conclusion of the algorithm the records are moved to an output area $S_1, \ldots, S_N$ in the desired order.

D1. [Clear COUNTs.] Set COUNT[u] through COUNT[v] all to zero.

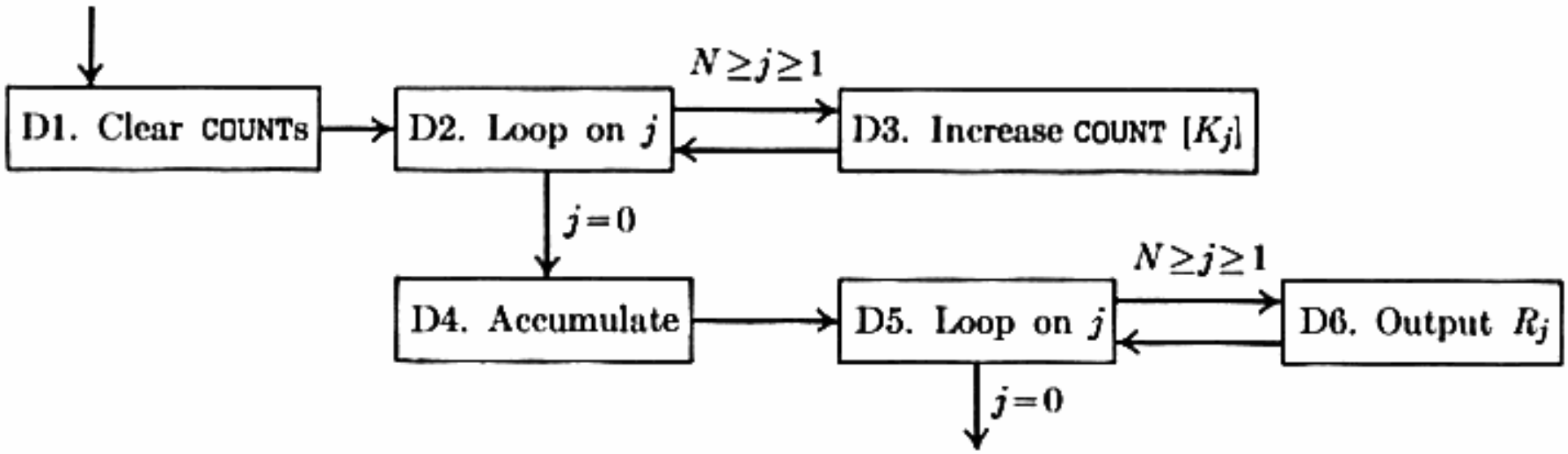


Fig. 9. Algorithm D: Distribution counting.

D2. [Loop on j.] Perform step D3 for $1 \leq j \leq N$; then go to step D4.

D3. [Increase COUNT$[K_j]$.] Increase the value of COUNT$[K_j]$ by 1.

D4. [Accumulate.] (At this point COUNT$[i]$ is the number of keys which are equal to i.) Set COUNT$[i] \leftarrow$ COUNT$[i]$ + COUNT$[i-1]$, for $i = u+1, u+2, \ldots, v$.

D5. [Loop on j.] (At this point COUNT$[i]$ is the number of keys which are less than or equal to i; in particular COUNT$[v] = N$.) Perform step D6 for $j = N, N-1, \ldots, 1$; then terminate the algorithm.

D6. [Output R_j.] Set $i \leftarrow$ COUNT$[K_j]$, $S_i \leftarrow R_j$, and COUNT$[K_j] \leftarrow i - 1$. ∎

An example of this algorithm is worked out in exercise 6; a MIX program appears in exercise 9. Under the conditions stated above, this sorting procedure is very fast.

Sorting by comparison counting as in Algorithm C was first mentioned in print by E. H. Friend [*JACM* **3** (1956), 152], although he didn't claim it as his own invention. Distribution sorting as in Algorithm D was first developed by H. Seward in 1954 for use with radix sorting techniques which we will discuss later (see Section 5.2.5); it was also published under the name "Mathsort" by W. Feurzig, *CACM* **3** (1960), 601.

EXERCISES

1. [*15*] Would Algorithm C still work if i varies from 2 up to N in step C2, instead of from N down to 2? What if j varies from 1 up to $i - 1$ in step C3?

2. [*21*] Show that Algorithm C works properly when equal keys are present. If $K_j = K_i$ and $j < i$, does R_j come before or after R_i in the final ordering?

▶ **3.** [*21*] Would Algorithm C still work properly if the test in step D4 were changed from "$K_i < K_j$" to "$K_i \leq K_j$"?

4. [*16*] Write a `MIX` program which "finishes" the sorting begun by Program C; your program should transfer the keys to locations `OUTPUT+1` through `OUTPUT+N`, in ascending order. How much time does your program require?

5. [*22*] Does the following set of changes improve Program C?

New line 8a: `INCX 0,2`
Change line 10: `JGE  5F`
Change line 14: `DECX 1`
Delete line 15

6. [*18*] Simulate Algorithm D by hand, showing intermediate results when the 16 records `5T`, `0C`, `5U`, `0D`, `9.`, `1N`, `8S`, `2R`, `6A`, `4A`, `1G`, `5L`, `6T`, `6I`, `7O`, `7N` are being sorted. Here the numeric digit is the key, and the alphabetic information is just carried along with the records.

7. [*13*] Is Algorithm D a "stable" sorting method?

8. [*15*] Would Algorithm D still work properly if j were to vary from 1 up to N in step D5, instead of from N down to 1?

9. [*23*] Write a `MIX` program for Algorithm D, analogous to Program C and exercise 4. What is the execution time of your program, as a function of N and $(v - u)$?

10. [*25*] Design an efficient algorithm which replaces the N quantities $(R_1, \ldots, R_N)$ by $(R_{p(1)}, \ldots, R_{p(N)})$, respectively, given the values of $R_1, \ldots, R_N$ and the permutation $p(1) \ldots p(N)$ of $\{1, \ldots, N\}$. Try to avoid using excess memory space. (This problem arises if we wish to rearrange records in memory after an address table sort, without requiring room for storing $2N$ records.)

11. [*M27*] Write a `MIX` program for the algorithm of exercise 10, and analyze its efficiency.

▶ **12.** [*25*] Design an efficient algorithm suitable for rearranging the records $R_1, \ldots, R_N$ into sorted order, after a list sort (Fig. 7) has been completed. Try to avoid using excess memory space.

▶ **13.** [*27*] Algorithm D requires space for $2N$ records $R_1, \ldots, R_N$ and $S_1, \ldots, S_N$. Show that it is possible to get by with only N records $R_1, \ldots, R_N$, if a new unshuffling procedure is substituted for steps D5 and D6. (Thus the problem is to design an algorithm which rearranges $R_1, \ldots, R_N$ in place, based on the values of

$$\texttt{COUNT}[u], \ldots, \texttt{COUNT}[v]$$

after step D4, without using additional memory space; this is essentially a generalization of the problem considered in exercise 10.)

5.2.1. Sorting by Insertion

One of the important families of sorting techniques is based on the "bridge player" method mentioned near the beginning of Section 5.2: Before examining record R_j, we assume that the preceding records $R_1, \ldots, R_{j-1}$ have already been sorted; then we insert R_j into its proper place among the previously sorted records. Several interesting variations on this basic theme are possible.

Straight insertion. The simplest insertion sort is the most obvious one. Assume that $1 < j \leq N$ and that records $R_1, \ldots, R_{j-1}$ have been rearranged so that

$$K_1 \leq K_2 \leq \cdots \leq K_{j-1}.$$

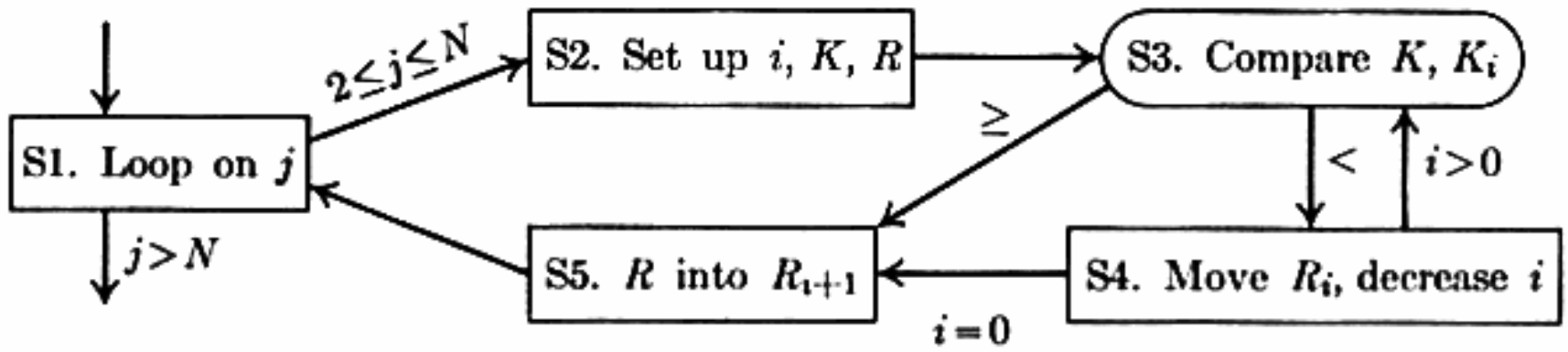


Fig. 10. Algorithm S: Straight insertion.

(Remember that, throughout this chapter, K_j denotes the key portion of R_j.) We compare the new key K_j with K_{j-1}, K_{j-2}, . . . , in turn, until discovering that R_j should be inserted between records R_i and R_{i+1}; then we move records $R_{i+1}, \ldots, R_{j-1}$ up one space and put the new record into position $i + 1$. It is convenient to combine the comparison and moving operations, interleaving them as shown in the following algorithm; since R_j "settles to its proper level" this method of sorting has often been called the *sifting* or *sinking* technique.

Algorithm S (*Straight insertion sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \le \cdots \le K_N$.

S1. [Loop on j.] Perform steps S2 through S5 for $j = 2, 3, \ldots, N$; then terminate the algorithm.

S2. [Set up i, K, R.] Set $i \leftarrow j - 1$, $K \leftarrow K_j$, $R \leftarrow R_j$. (In the following steps we will attempt to insert R into the correct position, by comparing K with K_i for decreasing values of i.)

S3. [Compare K, K_i.] If $K \ge K_i$, go to step S5. (We have found the desired position for record R.)

S4. [Move R_i, decrease i.] Set $R_{i+1} \leftarrow R_i$, then $i \leftarrow i - 1$. If $i > 0$, go back to step S3. (If $i = 0$, K is the smallest key found so far, so record R belongs in position 1.)

S5. [R into R_{i+1}.] Set $R_{i+1} \leftarrow R$. ∎

Table 1 shows how our sixteen example numbers are sorted by Algorithm S. This method is extremely easy to implement on a computer; in fact the following MIX program is the shortest decent sorting routine in this book.

Program S (*Straight insertion sort*). The records to be sorted are in locations INPUT+1 through INPUT+N; they are sorted in place in the same area, on a full-word key. rI1 $\equiv j - N$; rI2 $\equiv i$; rA $\equiv R \equiv K$; assume that $N \ge 2$.

01	START	ENT1	2-N	1	*S1. Loop on j.* $j \leftarrow 2$.
02	2H	LDA	INPUT+N,1	$N - 1$	*S2. Set up i, K, R.*
03		ENT2	N-1,1	$N - 1$	$i \leftarrow j - 1$.
04	3H	CMPA	INPUT,2	$B + N - 1 - A$	*S3. Compare K, K_i.*
05		JGE	5F	$B + N - 1 - A$	To S5 if $K \ge K_i$.

06	4H	LDX	INPUT,2	B	*S4. Move R_i, decrease i.*
07		STX	INPUT+1,2	B	$R_{i+1} \leftarrow R_i$.
08		DEC2	1	B	$i \leftarrow i - 1$.
09		J2P	3B	B	To S3 if $i > 0$.
10	5H	STA	INPUT+1,2	$N - 1$	*S5. R into R_{i+1}.*
11		INC1	1	$N - 1$	
12		J1NP	2B	$N - 1$	$2 \le j \le N$. ∎

The running time of this program is $9B + 10N - 3A - 9$ units, where N is the number of records sorted, A is the number of times i decreases to zero in step S4, and B is the number of moves. Clearly A is the number of times $K_j < (K_1, \ldots, K_{j-1})$, for $1 < j \le N$; this is the number of left-to-right minima, namely the quantity which was analyzed carefully in Section 1.2.10. Some reflection shows us that B is also a familiar quantity: The number of moves for fixed j is the number of inversions of K_j, so B is the number of inversions of the permutation $K_1\, K_2 \ldots K_N$. Hence by Eqs. 1.2.10–16 and 5.1.1–12, 13,

$$A = (\min 0,\ \text{ave}\ H_N - 1,\ \max N - 1,\ \text{dev}\ \sqrt{H_n - H_n^{(2)}});$$

$$B = (\min 0,\ \text{ave}\ (N^2 - N)/4,\ \max (N^2 - N)/2,\ \text{dev}\ \sqrt{N(N-1)(N+2.5)/6});$$

and the average running time of Program S, assuming that the input keys are distinct and randomly ordered, is $(2.25N^2 + 7.75N - 3H_N - 6)u$. Exercise 33 explains how to improve this slightly.

The example data in Table 1 involves 16 items; there are two left-to-right minima, namely 087 and 061; and there are 41 inversions, as we have seen in the previous section. Hence $N = 16$, $A = 2$, $B = 41$, and the total sorting time is $514u$.

Table 1

EXAMPLE OF STRAIGHT INSERTION

^503 : 087
087 503 ^: 512
^087 503 512 : 061
061 087 503 512 ^: 908
061 087 ^503 512 908 : 170
061 087 170 503 512 ^908 : 897
. . . . .
061 087 154 170 275 426 503 509 512 612 653 677 ^765 897 908 : 703
061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908

Binary insertion and two-way insertion. While the jth record is being processed during a straight insertion sort, we compare its key with about $\frac{1}{2}j$ of the previously sorted keys, on the average; therefore the total number of comparisons performed comes to about $\frac{1}{2}(1 + 2 + \cdots + N) \approx \frac{1}{4}N^2$, and this gets very large when N is only moderately large. In Section 6.2.1 we shall study "binary search" techniques, which show where to insert the jth item after only about $\log_2 j$ well-chosen comparisons have been made. For example, when inserting the 64th record we can start by comparing K_{64} with K_{32}; if it is less, we compare it with K_{16}, but if it is greater we compare it with K_{48}, etc., so that the proper place to insert R_{64} will be known after making only six comparisons. The total number of comparisons for inserting all N items comes to about $N \log_2 N$, a substantial improvement over $\frac{1}{4}N^2$; and Section 6.2.1 shows that the corresponding program need not be much more complicated than a program for straight insertion. This method is called *binary insertion;* it was mentioned by John Mauchly as early as 1946, in the first published discussion of computer sorting.

The unfortunate difficulty with binary insertion is that it solves only half of the problem; after we have found where record R_j is to be inserted, we still need to move about $\frac{1}{2}j$ of the previously sorted records in order to make room for R_j, so the total running time is still essentially proportional to N^2. Some computers (e.g., the IBM 705) have a built-in "tumble" instruction which does such move operations at high speed, but as N increases the dependence on N^2 eventually takes over. For example, an analysis by H. Nagler [*CACM* **3** (1960), 618–620] indicates that binary insertion should not be recommended for sorting more than about $N = 128$ records on the 705, when each record is 80 characters long, and similar analyses apply to other machines.

Of course, a clever programmer can think of various ways to reduce the amount of moving that is necessary; the first such trick, proposed early in the 1950's, is illustrated in Table 2. Here the first item is placed in the center of

Table 2

TWO-WAY INSERTION

			^503				
			087	503^			
		^087	503	512			
		061	087	503	512^		
		061	087^	503	512	908	
	061	087	170	503	512	^908	
	061	087	170^	503	512	897	908
061	087	170	275	503	512	897	908

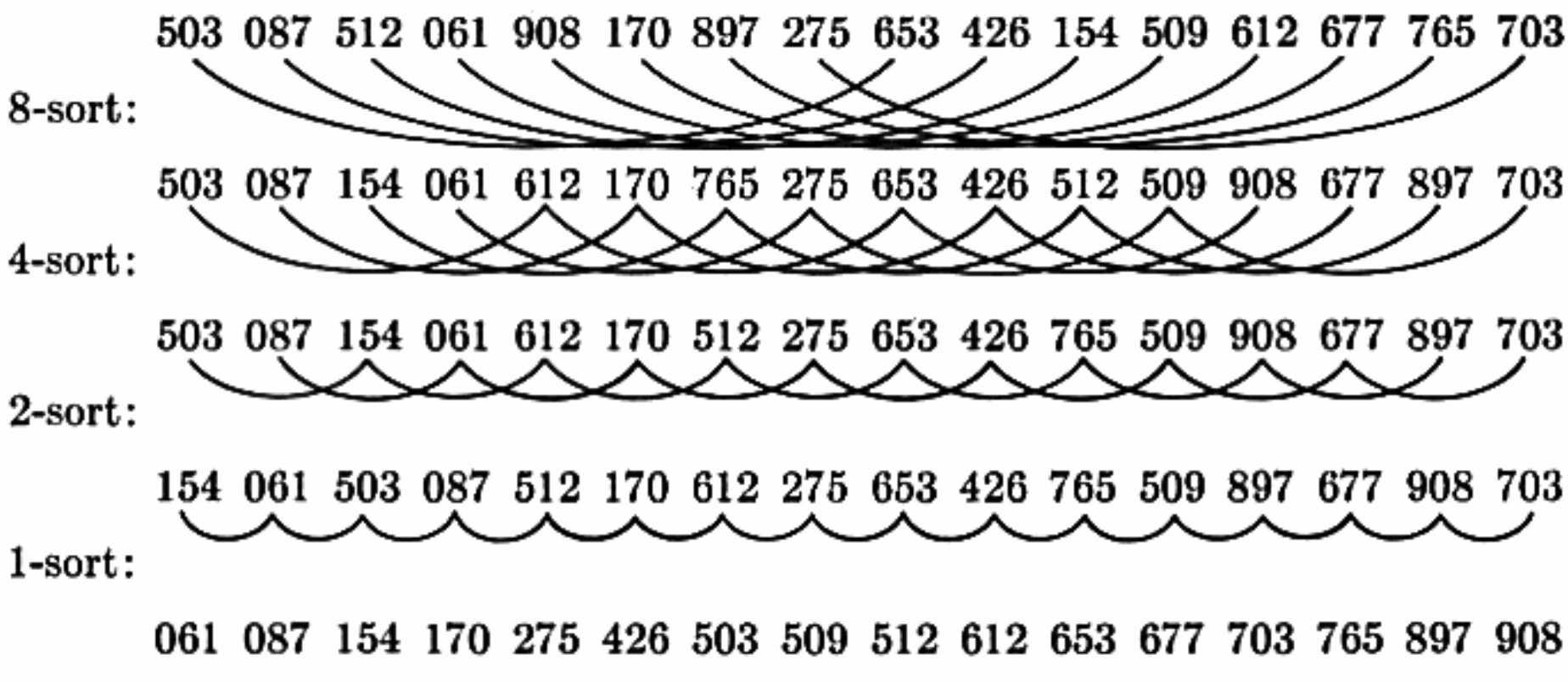
an output area, and space is made for subsequent items by moving to the right or to the left, whichever is most convenient. This saves about half the running time of ordinary binary insertion, at the expense of a somewhat more complicated program. It is possible to use this method without using up more space than required for N records (see exercise 6); but we shall not dwell any longer on this "two-way" method of insertion, since considerably more interesting methods have been developed.

Shell's method. If we have a sorting algorithm which moves items only one position at a time, its average running time will be, at best, proportional to N^2, since each record must travel an average of about $\frac{1}{3}N$ positions during the sorting process (see exercise 7). Therefore, if we want to make substantial improvements over straight insertion, we need some mechanism by which the records can take long leaps instead of short steps.

Such a method was proposed in 1959 by Donald L. Shell [*CACM* **2** (July, 1959), 30–32]; we shall call it the *diminishing increment sort*. Table 3 illustrates the general idea behind his method: First we divide the 16 records into 8 groups of two each, namely (R_1, R_9), (R_2, R_{10}), . . . , (R_8, R_{16}). Sorting each group of records separately takes us to the second line of Table 3; this is called the "first pass." Note that 154 has changed places with 512; 908 and 897 both jump to the right. Now we divide the records into 4 groups of four each, namely (R_1, R_5, R_9, R_{13}), . . . , $(R_4, R_8, R_{12}, R_{16})$, and again each group is sorted separately; this "second pass" takes us to line 3. A third pass sorts two groups of eight records, then a fourth pass completes the job by sorting all 16 records. Each of the intermediate sorting processes involves either a comparatively short file or a file which is comparatively well ordered, so straight insertion can

Table 3

DIMINISHING INCREMENT SORT (8, 4, 2, 1)



be used for each sorting operation; the records tend to converge quickly to their final destination.

The sequence of increments 8, 4, 2, 1 is not sacred; *any* sequence $h_t, h_{t-1}, \ldots, h_1$ can be used, so long as the last increment h_1 equals 1. For example, Table 4 shows the same data sorted with increments 7, 5, 3, 1. Some sequences are much better than others; we will discuss the choice of increments later.

Algorithm D (*Diminishing increment sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \le \cdots \le K_N$. An auxiliary sequence of increments $h_t, h_{t-1}, \ldots, h_1$ is used to control the sorting process, where $h_1 = 1$; proper choice of these increments can significantly decrease the sorting time. This algorithm reduces to Algorithm S when $t = 1$.

D1. [Loop on s.] Perform step D2 for $s = t, t - 1, \ldots, 1$; then terminate the algorithm.

D2. [Loop on j.] Set $h \leftarrow h_s$, and perform steps D3 through D6 for $h < j \le N$. (We will use a straight insertion method to sort elements that are h positions apart, so that $K_i \le K_{i+h}$ for $1 \le i \le N - h$. Steps D3 through D6 are essentially the same as steps S2 through S5, respectively, in Algorithm S.)

D3. [Set up i, K, R.] Set $i \leftarrow j - h$, $K \leftarrow K_j$, $R \leftarrow R_j$.

D4. [Compare K, K_i.] If $K \ge K_i$, go to step D6.

D5. [Move R_i, decrease i.] Set $R_{i+h} \leftarrow R_i$, then $i \leftarrow i - h$. If $i > 0$, go back to step D4.

D6. [R into R_{i+h}.] Set $R_{i+h} \leftarrow R$. ∎

The corresponding MIX program is not much longer than our program for straight insertion. Lines 08–19 of the following code are a direct translation of Program S into the more general framework of Algorithm D.

Program D (*Diminishing increment sort*). We assume that the increments are stored in an auxiliary table, with h_s in location H + s; all increments are less than N. Register assignments: rI1 ≡ $j - N$; rI2 ≡ i; rA ≡ $R \equiv K$; rI3 ≡ s; rI4 ≡ h. Note that this program modifies itself, in order to obtain efficient execution of the inner loop.

```
01    START  ENT3  T              1     D1. Loop on s.  s ← t.
02    1H     LD4   H,3            T     D2. Loop on j.  h ← h_s.
03           ENT1  INPUT,4        T     Modify the addresses of three instructions
04           ST1   6F(0:2)        T           in the main loop.
05           ST1   7F(0:2)        T
06           ENN1  -N,4           T     rI1 ← N − h.
07           ST1   4F(0:2)        T
08           ENT1  1-N,4          T     j ← h + 1.
```

09	2H	LDA	INPUT+N,1	$NT - S$	*D3. Set up i, K, R.*
10	4H	ENT2	N-H,1	$NT - S$	$i \leftarrow j - h$. (Instruction modified)
11	5H	CMPA	INPUT,2	$B + NT - S - A$	*D4. Compare K, K_i.*
12		JGE	7F	$B + NT - S - A$	To D6 if $K \geq K_i$.
13		LDX	INPUT,2	B	*D5. Move R_i, decrease i.*
14	6H	STX	INPUT+H,2	B	$R_{i+h} \leftarrow R_i$. (Instruction modified)
15		DEC2	0,4	B	$i \leftarrow i - h$.
16		J2P	5B	B	To D4 if $i > 0$.
17	7H	STA	INPUT+H,2	$NT - S$	*D6. R into R_{i+h}.* (Instruction modified)
18	8H	INC1	1	$NT - S$	$j \leftarrow j + 1$.
19		J1NP	2B	$NT - S$	To D3 if $j \leq N$.
20		DEC3	1	T	
21		J3P	1B	T	$t \geq s \geq 1$. ∎

***Analysis of Shell's Method.** In order to choose a good sequence of increments $h_t, \ldots, h_1$ for use in Algorithm D, we need to analyze the running time as a function of those increments. This leads to some fascinating mathematical problems, not yet completely resolved; nobody has been able to determine the best possible sequence of increments for large values of N. Yet a good many interesting facts are known about the behavior of Shell's diminishing increment sort, and we will summarize them here; details appear in the exercises below. [Readers who are not mathematically inclined should skim over the next few pages, continuing at the formulas numbered (8) below.]

The frequency counts shown with Program D indicate that five factors determine the execution time: the size of the file, N; the number of passes (i.e., the number of increments), $T = t$; the sum of the increments,

$$S = h_1 + \cdots + h_t;$$

the number of comparisons, $B + NT - S - A$; and the number of moves, B. As in the analysis of Program S, A is the number of left-to-right minima encountered in the intermediate sorting operations, and B is the number of inversions in the subfiles. The factor which governs the running time is B, so we shall devote most of our attention to it. For purposes of analysis we shall assume that the keys are distinct and initially in random order.

Let us call the operation of step D2 "h-sorting," so that Shell's method consists of h_t-sorting, followed by h_{t-1}-sorting, . . . , followed by h_1-sorting. A file in which $K_i \leq K_{i+h}$ for $1 \leq i \leq N - h$ will be called "h-ordered."

Consider first the simplest generalization of straight insertion, when there are just two increments, $h_2 = 2$ and $h_1 = 1$. During the second pass we have a 2-ordered sequence of keys, $K_1\, K_2 \ldots K_N$. It is easy to see that the number of permutations $a_1\, a_2 \ldots a_n$ of $\{1, 2, \ldots, n\}$ such that $a_i \leq a_{i+2}$ for $1 \leq i \leq n - 2$ is

$$\binom{n}{\lfloor n/2 \rfloor},$$

since we obtain exactly one 2-ordered permutation for each choice of $\lfloor n/2 \rfloor$ elements to put in even-numbered positions $a_2\, a_4 \ldots$, with the remaining $\lceil n/2 \rceil$ elements going in odd-numbered positions. Each 2-ordered permutation

is equally likely after a random file has been 2-sorted. What is the average number of inversions among all such permutations?

Let A_n be the total number of inversions among all 2-ordered permutations of $\{1, 2, \ldots, n\}$. Clearly $A_1 = 0$, $A_2 = 1$, $A_3 = 2$; and by considering the six cases

1 3 2 4 1 2 3 4 1 2 4 3 2 1 3 4 2 1 4 3 3 1 4 2

we find that $A_4 = 1 + 0 + 1 + 1 + 2 + 3 = 8$. In order to investigate A_n in general, consider the "lattice diagram" illustrated in Fig. 11 for $n = 15$. A 2-ordered permutation of $\{1, 2, \ldots, n\}$ can be represented as a path from the upper left corner point (0, 0) to the lower right corner point $(\lceil n/2 \rceil, \lfloor n/2 \rfloor)$, if we make the kth step of the path go downwards or to the right, respectively, according as k appears in an odd or an even position in the permutation. This rule defines a one-to-one correspondence between 2-ordered permutations and n-step paths from corner to corner of the lattice diagram; for example, the path shown corresponds to the permutation

$$2\ 1\ 3\ 4\ 6\ 5\ 7\ 10\ 8\ 11\ 9\ 12\ 14\ 13\ 15. \qquad (1)$$

Furthermore, we can attach "weights" to the vertical lines of the path, as the diagram shows; a line from (i, j) to $(i + 1, j)$ gets weight $|i - j|$. A little study will convince the reader that the sum of these weights along each path is equal to the number of inversions of the corresponding permutation. (See exercise 12.) Thus, for example, (1) has $1 + 0 + 1 + 0 + 1 + 2 + 1 + 0 = 6$ inversions.

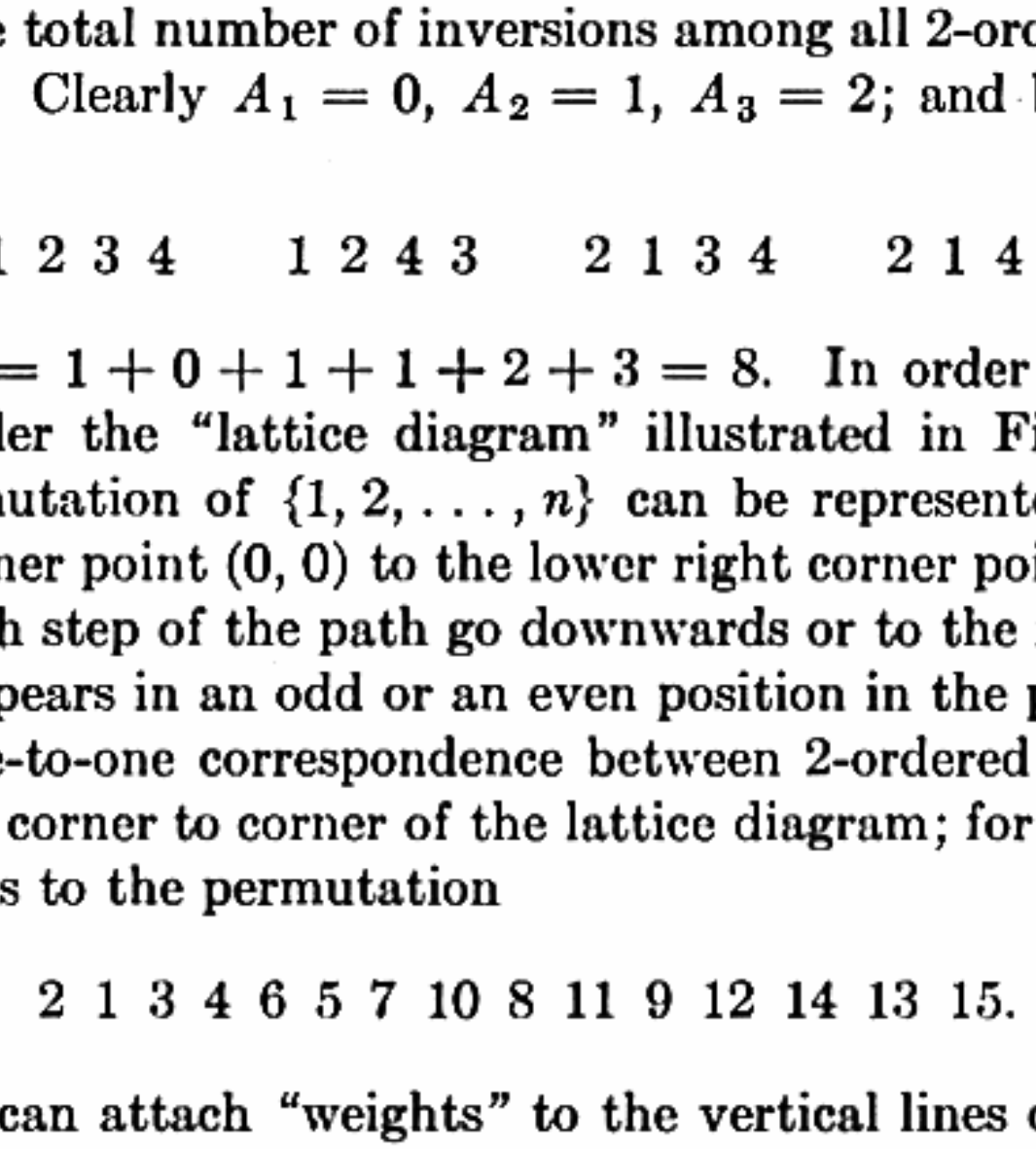


Fig. 11. Correspondence between 2-ordering and paths in a lattice. Italicized numbers are weights which correspond to the number of inversions in the 2-ordered permutation.

When $a \le a'$ and $b \le b'$, the number of relevant paths from (a, b) to (a', b') is the number of ways to mix $a' - a$ vertical lines with $b' - b$ horizontal lines, namely

$$\binom{a' - a + b' - b}{a' - a};$$

hence the number of permutations whose corresponding path traverses the vertical line segment from (i, j) to $(i + 1, j)$ is

$$\binom{i + j}{i}\binom{n - i - j - 1}{\lfloor n/2 \rfloor - j}.$$

Multiplying by the associated weight and summing over all segments gives

$$A_{2n} = \sum_{\substack{0 \le i \le n \\ 0 \le j \le n}} |i - j| \binom{i + j}{i}\binom{2n - i - j - 1}{n - j};$$

$$A_{2n+1} = \sum_{\substack{0 \le i \le n \\ 0 \le j \le n}} |i - j| \binom{i + j}{i}\binom{2n - i - j}{n - j}. \tag{2}$$

The absolute value signs in these sums make the calculations somewhat tricky, but exercise 14 shows that A_n has the surprisingly simple form $\lfloor n/2 \rfloor 2^{n-2}$. Hence the average number of inversions in a random 2-ordered permutation is

$$\lfloor n/2 \rfloor 2^{n-2} \Big/ \binom{n}{\lfloor n/2 \rfloor};$$

by Stirling's approximation this is asymptotically $\sqrt{\pi/128}\, n^{3/2} \approx 0.15n^{3/2}$. The maximum number of inversions is easily seen to be

$$\binom{\lfloor n/2 \rfloor + 1}{2} \approx \frac{1}{8} n^2.$$

It is instructive to study the distribution of inversions more carefully, by examining the generating functions

$$h_1(z) = 1, \qquad h_2(z) = 1 + z,$$
$$h_3(z) = 1 + 2z, \qquad h_4(z) = 1 + 3z + z^2 + z^3, \qquad \ldots, \tag{3}$$

as in exercise 15. In this way we find that the standard deviation is also proportional to $n^{3/2}$, so the distribution is not extremely stable about the mean.

Now let us consider the general two-pass case of Algorithm D, when the increments are h and 1:

Theorem H. *The average number of inversions in an h-ordered permutation of $\{1, 2, \ldots, n\}$ is*

$$f(n, h) = \frac{2^{2q-1}q!q!}{(2q+1)!}\left(\binom{h}{2}q(q+1) + \binom{r}{2}(q+1) - \frac{1}{2}\binom{h-r}{2}q\right), \quad (4)$$

where $q = \lfloor n/h \rfloor$, $r = n \bmod h$.

This theorem is due to Douglas H. Hunt [Bachelor's thesis, Princeton University (April, 1967)]. Note that when $h \geq n$ the formula correctly gives $f(n, h) = \frac{1}{2}\binom{n}{2}$.

Proof. An h-ordered permutation contains r sorted subsequences of length $q+1$, and $h-r$ of length q. Each inversion comes from a pair of distinct subsequences, and a given pair of distinct subsequences in a random h-ordered permutation defines a random 2-ordered permutation. The average number of inversions is therefore the sum of the average numbers of inversions between each pair of distinct subsequences, namely

$$\binom{r}{2}\frac{A_{2q+2}}{\binom{2q+2}{q+1}} + r(h-r)\frac{A_{2q+1}}{\binom{2q+1}{q}} + \binom{h-r}{2}\frac{A_{2q}}{\binom{2q}{q}} = f(n, h). \quad \blacksquare$$

Corollary. *If the sequence of increments $h_t, \ldots, h_1$ satisfies the condition*

$$h_{s+1} \bmod h_s = 0, \qquad \text{for} \qquad t > s \geq 1, \quad (5)$$

then the average number of move operations in Algorithm D is

$$\sum_{t \geq s \geq 1} (r_s f(q_s + 1, h_{s+1}/h_s) + (h_s - r_s)f(q_s, h_{s+1}/h_s)), \quad (6)$$

where $r_s = N \bmod h_s$, $q_s = \lfloor N/h_s \rfloor$, $h_{t+1} = N$, and f is defined in (4).

Proof. The process of h_s-sorting consists of a straight insertion sort on r_s (h_{s+1}/h_s)-ordered subfiles of length $q_s + 1$, and on $(h_s - r_s)$ such subfiles of length q_s. The divisibility condition implies that each of these subfiles is a "random" (h_{s+1}/h_s)-ordered permutation, in the sense that each (h_{s+1}/h_s)-ordered permutation is equally likely, since we are assuming that the original input was a random permutation of distinct elements. $\blacksquare$

Condition (5) in this corollary is always satisfied for *two-pass* Shell sorts when the increments are h and 1, respectively. If $q = \lfloor N/h \rfloor$ and $r = N \bmod h$, the quantity B in Program D will have an average value of

$$rf(q+1, N) + (h-r)f(q, N) + f(N, h) = \frac{r}{2}\binom{q+1}{2} + \frac{h-r}{2}\binom{q}{2} + f(N, h).$$

To a first approximation, the function $f(n, h)$ equals $(\sqrt{\pi}/8)n^{3/2}h^{1/2}$; compare it to the smooth curve in Fig. 12. Hence the running time for a two-pass Pro-

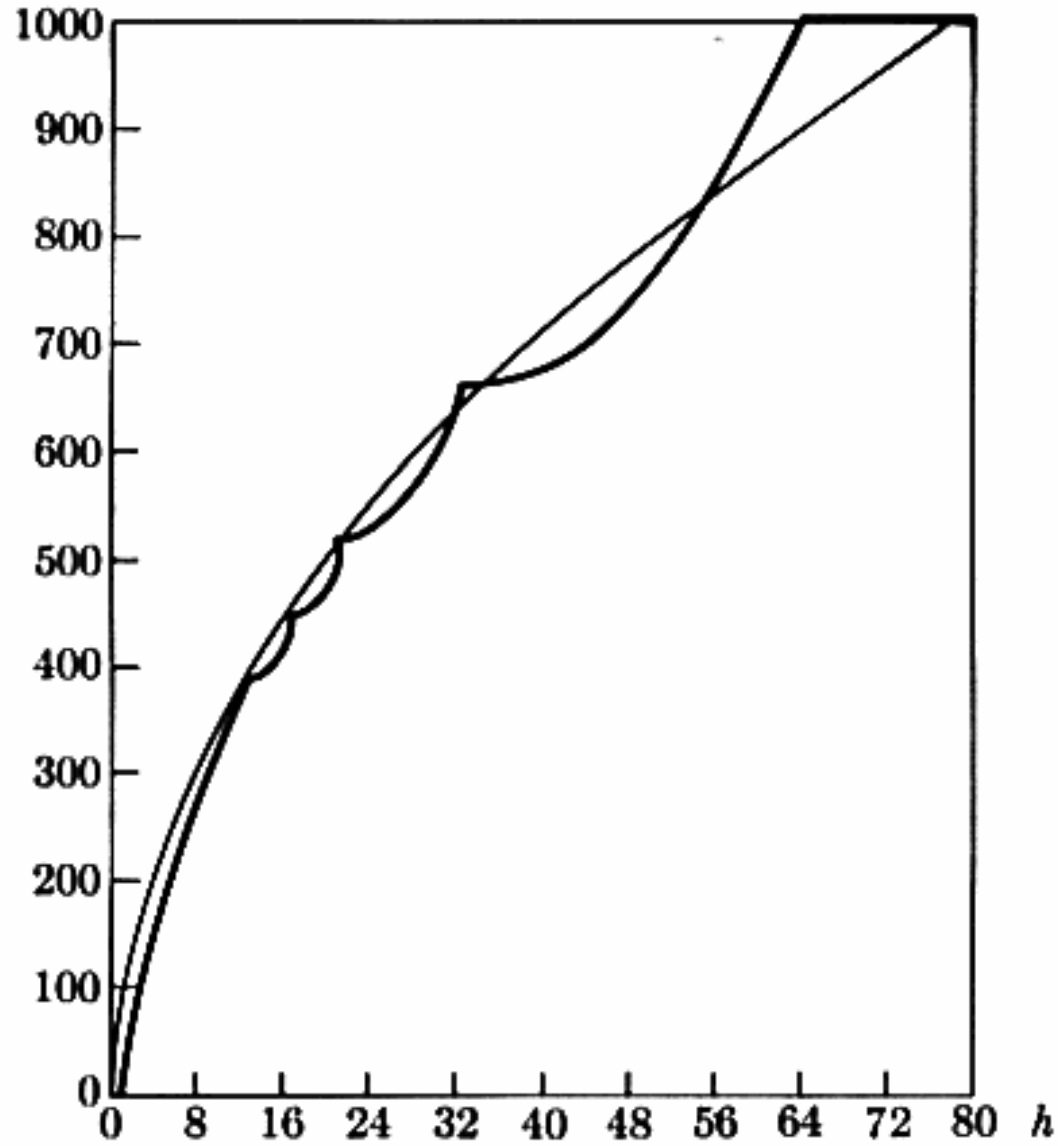


Fig. 12 The average number, $f(n, h)$, of inversions in an h-ordered file of n elements, shown for $n = 64$.

gram D is approximately proportional to $2N^2/h + \sqrt{\pi N^3 h}$. The best choice of h is therefore approximately $\sqrt[3]{16N/\pi} \approx 1.72\sqrt[3]{N}$; and with this choice of h we get an average running time proportional to $N^{5/3}$.

Thus we can make a substantial improvement over straight insertion, from $O(N^2)$ to $O(N^{1.667})$, just by using Shell's method with two increments. Clearly we can do even better when more increments are used. Exercise 18 discusses the optimum choice of $h_t, \ldots, h_1$ when t is fixed and when the h's are constrained by the divisibility condition; the running time decreases to $O(N^{1.5+\epsilon/2})$, where $\epsilon = 1/(2^t - 1)$, for large N. We cannot break the $N^{1.5}$ barrier by using the formulas above, since the last pass always contributes

$$f(N, h_2) \approx (\sqrt{\pi}/8)N^{3/2}h_2^{1/2}$$

inversions to the sum.

But our intuition tells us that we can do even better when the increments $h_t, \ldots, h_1$ do *not* satisfy the divisibility condition (5). For example, 8-sorting followed by 4-sorting followed by 2-sorting does not allow any interaction between keys in even and odd positions; therefore the final 1-sorting pass is faced with $O(N^{3/2})$ inversions. By contrast, 7-sorting followed by 5-sorting followed by 3-sorting jumbles things up in such a way that the final 1-sorting pass cannot encounter more than $2N$ inversions! (See exercise 26.) Indeed, an astonishing phenomenon occurs:

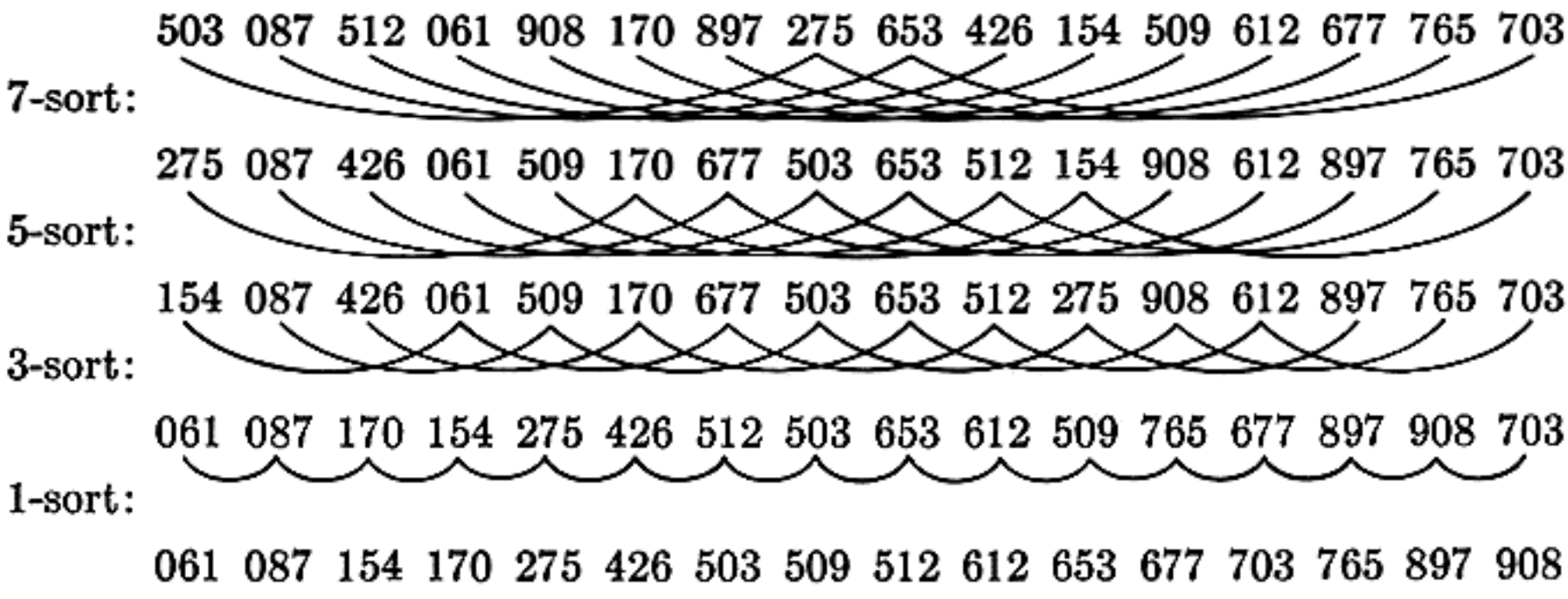
Theorem K. *If a k-ordered file is h-sorted, it remains k-ordered.*

Thus a file which is first 7-sorted, then 5-sorted, becomes both 7-ordered and 5-ordered. And if we 3-sort it, the result is ordered by 7's, 5's, and 3's.

Examples of this remarkable property appear in Table 4.

Table 4

DIMINISHING INCREMENT SORT (7, 5, 3, 1)



Proof. Exercise 20 shows that this theorem comes from the following fact:

Lemma L. *Let m, n, r be nonnegative integers, and let $x_1, \ldots, x_{m+r}$ and $y_1, \ldots, y_{n+r}$ be any sequences of numbers such that*

$$y_1 \le x_{m+1}, \qquad y_2 \le x_{m+2}, \qquad \ldots, \qquad y_r \le x_{m+r}. \tag{7}$$

If the x's and y's are sorted independently, so that $x_1 \le \cdots \le x_{m+r}$ and $y_1 \le \cdots \le y_{n+r}$, the relations (7) will still be valid.

Proof. All but m of the x's are known to dominate (i.e., to be greater than or equal to) some y, where distinct x's dominate distinct y's. Let $1 \le j \le r$. Since x_{m+j} after sorting dominates $m + j$ of the x's, it dominates at least j of the y's; so it dominates the *smallest* j of the y's; hence $x_{m+j} \ge y_j$ after sorting. ∎ ∎

Theorem K shows us that it is desirable to sort with relatively prime increments, but it does not lead directly to exact estimates of the number of moves made in Algorithm D. Since the number of permutations of $\{1, 2, \ldots, n\}$ which are both h-ordered and k-ordered is not always a divisor of $n!$, we can see that Theorem K does not tell the whole story; some k- and h-ordered files are obtained more often than others after k- and h-sorting. Moreover, there is no obvious way to find the "worst case" of Algorithm D for general increments $h_t, \ldots, h_1$. Therefore the analysis of this algorithm in the general case has baffled everyone so far; essentially all that is known is the approximate asymptotic form of the maximum running time in certain cases:

Theorem P. *The running time for Algorithm D is $O(N^{3/2})$ when $h_s = 2^s - 1$, $1 \le s \le t = \lfloor \log_2 N \rfloor$.*

Proof. It suffices to bound B_s, the number of moves in pass s, in such a way that $B_t + \cdots + B_1 = O(N^{3/2})$. During the first $t/2$ passes, for $t \geq s \geq t/2$, we may use the obvious bound $B_s = O(h_s(N/h_s)^2)$, and for subsequent passes we may use the result of exercise 23, $B_s = O(Nh_{s+2}h_{s+1}/h_s)$; hence $B_t + \cdots + B_1 = O(N(2 + 2^2 + \cdots + 2^{t/2} + 2^{t/2} + \cdots + 2)) = O(N^{3/2})$. ∎

This theorem is due to A. A. Papernov and G. V. Stasevich, *Problemy Peredachi Informatsii* **1**, 3 (1965), 81–98. It gives an upper bound on the *maximum* running time of the algorithm, not merely a bound on the *average* running time. The result is not trivial since the maximum running time when the h's satisfy the divisibility constraint (5) is of order N^2; and exercise 24 shows that the exponent 3/2 cannot be lowered.

An interesting improvement of Theorem P was discovered in 1969 by Vaughan Pratt: *If the increments are chosen to be the set of all numbers of the form 2^p3^q which are less than N, the running time of Algorithm D is of order $N(\log N)^2$.* In this case we can also make several important simplifications to the algorithm. Unfortunately, Pratt's method requires a comparatively large number of passes, so it is not the best way to choose the increments unless N is quite large; see exercises 30 and 31.

Let us consider the *total* running time of Program D, namely $(9B + 10NT + 13T - 10S - 3A + 1)u$. Table 5 shows the average running time for various

Table 5

ANALYSIS OF ALGORITHM D WHEN $N = 8$

Increments	A_{ave}	B_{ave}	S	T	MIX time
1	1.718	14.000	1	1	$204.85u$
2 1	2.667	9.657	3	2	$235.91u$
3 1	2.917	9.100	4	2	$220.16u$
4 1	3.083	10.000	5	2	$217.75u$
5 1	2.601	10.000	6	2	$210.00u$
6 1	2.135	10.667	7	2	$206.60u$
7 1	1.718	12.000	8	2	$208.85u$
4 2 1	3.500	8.324	7	3	$272.32u$
5 3 1	3.301	8.167	9	3	$251.60u$
3 2 1	3.320	7.829	6	3	$278.50u$

sequences of increments when $N = 8$; each of the entries in this table can be calculated from the formulas above or in exercise 19, except when the increments are 5 3 1 or 3 2 1; in the latter cases an exhaustive study of all 8! permutations was made. Note that for this small value of N the bookkeeping operations dominate the running time, so that the best results are obtained when $t = 1$; hence for $N = 8$ we are better off using simple straight insertion. (The average

running time of Program S when $N = 8$ is only $191.85u$.) Curiously, the best two-pass algorithm occurs when $h_2 = 6$, since a large value of S is more important here than a small value of B. Similarly, the three increments 3 2 1 minimize the average number of moves, but they do not lead to the best three-pass sequence. It may be of interest to record here some "worst case" permutations which maximize the number of moves, since the general construction of such permutations is still unknown:

$h_3 = 5, \quad h_2 = 3, \quad h_1 = 1:$ 8 5 2 6 3 7 4 1 (19 moves)

$h_3 = 3, \quad h_2 = 2, \quad h_1 = 1:$ 8 3 5 7 2 4 6 1 (17 moves)

As N grows larger we have a slightly different picture. Table 6 shows the approximate number of moves for various sequences of increments when $N = 1000$. The first few entries satisfy the divisibility constraints (5), so that formula (6) can be used; empirical tests were used to get approximate average values for other increment sequences. Five random files of 1000 elements were generated, and they were each sorted with each of the sequences of increments.

Some patterns are evident in this data, but the behavior of Algorithm D still remains very obscure. Shell originally suggested using the increments $\lfloor N/2 \rfloor, \lfloor N/4 \rfloor, \lfloor N/8 \rfloor, \ldots$, but this is undesirable when the binary representation of N contains a long string of zeros. Lazarus and Frank [*CACM* **3** (1960), 20–22] suggested using essentially the same sequence, but adding 1 when necessary, to make all increments odd. Hibbard [*CACM* **6** (1963), 206–213] suggested using increments of the form $2^k - 1$; Papernov and Stasevich suggested the form $2^k + 1$. Other natural sequences investigated in Table 6 involve the numbers $(2^k - (-1)^k)/3$, $(3^k - 1)/2$, and the Fibonacci numbers.

The minimum number of moves, about 6600, was observed for increments of the form $2^k + 1$; but it is important to realize that the number of moves is not the only consideration, even though it asymptotically dominates the running time. Since Program D takes $9B + 10NT + \cdots$ units of time, we see that saving one pass is about as desirable as saving $\frac{10}{9}N$ moves; we are willing to add 1100 moves if we can save one pass. Therefore it seems unwise to start with h_t greater than, say, $\frac{1}{3}N$, since a large increment will not decrease the subsequent number of moves enough to justify the first pass.

A large number of experiments on Algorithm D were conducted by James Peterson and David L. Russell at Stanford University in 1971; they found that the average number of moves, B, can be reasonably well approximated by the following formulas for $100 \leq N \leq 60000$:

$1.09N^{1.27}$ or $.30N(\ln N)^2 - 1.35N \ln N$ for increments $2^k + 1, \ldots, 9, 5, 3, 1$;

$1.22N^{1.26}$ or $.29N(\ln N)^2 - 1.26N \ln N$ for increments $2^k - 1, \ldots, 15, 7, 3, 1$;

$1.12N^{1.28}$ or $.36N(\ln N)^2 - 1.73N \ln N$ for increments $(2^k \pm 1)/3, \ldots, 11, 5, 3, 1$;

$1.66N^{1.25}$ or $.33N(\ln N)^2 - 1.26N \ln N$ for increments $(3^k - 1)/2, \ldots, 40, 13, 4, 1$.

Table 6

ANALYSIS OF ALGORITHM D WHEN $N = 1000$

Increments	A_{ave}	B_{ave}	T
1	6.485	249750	1
17 1	64.66	41666.5	2
60 6 1	158.1	26360.6	3
140 20 4 1	262.1	21912.5	4
256 64 16 4 1	361.9	20458.9	5
576 192 48 16 4 1	419.2	20087.6	6
729 243 81 27 9 3 1	378.3	18533.2	7
512 256 128 64 32 16 8 4 2 1	492.5	16434.7	10
500 250 125 62 31 15 7 3 1	525	7600	9
501 251 125 63 31 15 7 3 1	575	7200	9
511 255 127 63 31 15 7 3 1	550	7100	9
255 127 63 31 15 7 3 1	450	7300	8
127 63 31 15 7 3 1	300	8150	7
63 31 15 7 3 1	200	9700	6
31 15 7 3 1	125	13600	5
513 257 129 65 33 17 9 5 3 1	575	6600	10
257 129 65 33 17 9 5 3 1	450	6800	9
129 65 33 17 9 5 3 1	300	7600	8
65 33 17 9 5 3 1	200	9400	7
33 17 9 5 3 1	125	13000	6
683 341 171 85 43 21 11 5 3 1	500	7050	10
341 171 85 43 21 11 5 3 1	500	7300	9
255 63 15 7 3 1	400	8600	6
257 65 17 5 3 1	400	8700	6
341 85 21 5 3 1	425	9300	6
610 377 233 144 89 55 34 21 13 8 5 3 2 1	550	7400	14
377 144 55 21 8 3 1	475	8800	7
365 122 41 14 5 2 1	450	8300	7
364 121 40 13 4 1	450	9200	6
121 40 13 4 1	275	9900	5

For example when $N = 20000$ we get $B \approx 31000, 33000, 35000, 39000$, with these respective types of increments. Table 7 shows typical breakdowns of

Table 7

MOVES PER PASS; EXAMPLES WITH $N = 20000$

h_s	B_s	h_s	B_s	h_s	B_s
4095	19460	4097	19550	3280	25210
2047	15115	2049	14944	1093	28324
1023	15869	1025	15731	364	35477
511	18891	513	18548	121	47158
255	22306	257	21827	40	62110
127	27400	129	27814	13	88524
63	35053	65	33751	4	74599
31	34677	33	34303	1	34666
15	51054	17	46044		
7	40382	9	35817		
3	24044	5	19961		
1	16789	3	9628		
		1	13277		

moves per pass obtained in three of these experiments. Curiously, *both* of the functional forms $\alpha N(\ln N)^2 + \beta N \ln N$ and βN^α seem to give a pretty good fit to the observed data, although the exponential form was noticeably better than the logarithmic for smaller values of N. Further experiments were made for the increments $2^k - 1$, letting N range as high as 250,000; forty-five trials for $N = 250{,}000$ gave $B \approx 7{,}900{,}000$ with an observed standard deviation of about 50,000. The "best fit" formulas for the range $100 \leq N \leq 250{,}000$ turned out to be $1.21N^{1.26}$ and $.39N(\ln N)^2 - 2.33\ N \ln N$, respectively. Since the coefficients of the exponential form remained about the same, while the coefficients of the logarithmic form changed rather drastically, it is reasonable to conjecture that the exponential form gives the true asymptotic behavior of Shell's method.

This empirical data by no means exhausts the possibilities, and we have found no grounds for making strong assertions about what sequences of increments are best in Algorithm D. Since the increments $(3^k - 1)/2$ do not take substantially more moves, and since they require only about five-eighths as many passes as the increments of other forms, the evidence indicates that *it is reasonable to choose the increments in the following way:*

$$\textit{Let } h_1 = 1,\ h_{s+1} = 3h_s + 1, \textit{ and stop with } h_t \textit{ when } h_{t+2} \geq N. \qquad (8)$$

List insertion. Let us now leave Shell's method and consider other types of improvements over straight insertion. One of the most important general ways

to improve on a given algorithm is to examine its data structures carefully, since a reorganization of data structures to avoid unnecessary operations often leads to substantial improvements. Further discussion of this general idea appears in Section 2.4, where a rather complex algorithm is studied; let us consider how it applies to a very simple algorithm like straight insertion. What is the most appropriate data structure for Algorithm S?

Straight insertion involves two basic operations:

i) scanning an ordered file to find the largest key less than or equal to a given key; and

ii) inserting a new record into a specified part of the ordered file.

The file is obviously a linear list, and Algorithm S handles this list by using sequential allocation (Section 2.2.2); therefore it is necessary to move roughly half of the records in order to accomplish each insertion operation. On the other hand, we know that linked allocation (Section 2.2.3) is ideally suited to insertion, since only a few links need to be changed; and the other operation, sequential scanning, is about as easy with linked allocation as with sequential allocation. Only one-way linkage is needed, since we always scan the list in the same direction. Therefore we conclude that the "right" data structure for straight insertion is a one-way, linked linear list. It also becomes convenient to revise Algorithm S so that the list is scanned in increasing order:

Algorithm L (*List insertion*). Records $R_1, \ldots, R_N$ are assumed to contain keys $K_1, \ldots, K_N$, and "link fields" $L_1, \ldots, L_N$ capable of holding the numbers 0 through N; and there is another link field L_0 in an artificial record R_0 at the beginning of the file. This algorithm sets the link fields so that the records are linked together in ascending order. Thus, if $p(1) \ldots p(N)$ is the "stable" permutation which makes $K_{p(1)} \le \cdots \le K_{p(N)}$, this algorithm will yield

$$L_0 = p(1); \qquad L_{p(i)} = p(i+1), \quad \text{for} \quad 1 \le i \le N; \qquad L_{p(N)} = 0. \tag{9}$$

L1. [Loop on j.] Set $L_0 \leftarrow N$, $L_N \leftarrow 0$. (L_0 acts as the "head" of the list, and 0 acts as a null link; hence the list is essentially circular.) Perform steps L2 through L5 for $j = N-1, N-2, \ldots, 1$; then terminate the algorithm.

L2. [Set up p, q, K.] Set $p \leftarrow L_0$, $q \leftarrow 0$, $K \leftarrow K_j$. (In the following steps we will insert R_j into its proper place in the linked list, by comparing K with the previous keys in ascending order. The variables p and q act as pointers to the current place in the list, with $p = L_q$ so that q is one step behind p.)

L3. [Compare K, K_p.] If $K \le K_p$, go to step L5. (We have found the desired position for record R, between R_q and R_p in the list.)

L4. [Bump p, q.] Set $q \leftarrow p$, $p \leftarrow L_q$. If $p > 0$, go back to step L3. (If $p = 0$, K is the largest key found so far; hence record R belongs at the end of the list, between R_q and R_0.)

L5. [Insert into list.] Set $L_q \leftarrow j$, $L_j \leftarrow p$. ∎

This algorithm is important not only because it is a simple sorting method, but also because it occurs frequently as part of other list-processing algorithms. Table 8 shows the first few steps which occur when our sixteen example numbers are sorted.

Table 8

EXAMPLE OF LIST INSERTION

j:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
K_j:	——	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
L_j:	16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0
L_j:	16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0	15
L_j:	14	—	—	—	—	—	—	—	—	—	—	—	—	—	16	0	15

Program L (*List insertion*). We assume that K_j is stored in `INPUT` $+ j$(0:3), and L_j is stored in `INPUT+`j(4:5). rI1 $\equiv j$; rI2 $\equiv p$; rI3 $\equiv q$; rA(0:3) $\equiv K$.

```
01    KEY    EQU   0:3
02    LINK   EQU   4:5
03    START  ENT1  N                    1          L1. Loop on j. j ← N.
04           ST1   INPUT(LINK)          1          L0 ← N.
05           STZ   INPUT+N(LINK)        1          L_N ← 0.
06           JMP   6F                   1          Go to decrease j.
07    2H     LD2   INPUT(LINK)          N − 1      L2. Set up p, q, K.  p ← L0.
08           ENT3  0                    N − 1      q ← 0.
09           LDA   INPUT,1              N − 1      K ← K_j.
10    3H     CMPA  INPUT,2(KEY)         B+N−1−A    L3. Compare K, K_p.
11           JLE   5F                   B+N−1−A    To L5 if K ≤ K_p.
12    4H     ENT3  0,2                  B          L4. Bump p, q.  q ← p.
13           LD2   INPUT,3(LINK)        B          p ← L_q.
14           J2P   3B                   B          To L3 if p > 0.
15    5H     ST1   INPUT,3(LINK)        N − 1      L5. Insert into list.  L_q ← j.
16           ST2   INPUT,1(LINK)        N − 1      L_j ← p.
17    6H     DEC1  1                    N
18           J1P   2B                   N          N > j ≥ 1. ∎
```

The running time of this program is $7B + 14N - 3A - 6$ units, where N is the length of the file, A is the number of right-to-left maxima, and B is the number of inversions in the original permutation. (Cf. the analysis of Program S. Note that Program L does not rearrange the records in memory; this can be done as in exercise 5.2–12, at a cost of about $20N$ additional units of time.) Program S requires $(9B + 10N - 3A - 9)u$, and since B is about $\frac{1}{4}N^2$, we can see that the extra memory space used for the link fields has saved about 22 percent of the execution time. Another 22 percent can be saved by careful programming (see exercise 33), but the running time remains proportional to N^2.

To summarize what we have done so far: We started with Algorithm S, a simple and natural sorting algorithm which does about $\frac{1}{4}N^2$ comparisons and $\frac{1}{4}N^2$ moves. We improved it in one direction by considering binary insertion, which does about $N \log_2 N$ comparisons and $\frac{1}{4}N^2$ moves. Changing the data structure slightly with "two-way insertion" cuts the number of moves down to about $\frac{1}{8}N^2$. Shell's "diminishing increment" sorting technique cuts the number of comparisons and moves to about $N^{1.3}$, for N in a practical range; as $N \to \infty$ this number can be lowered to order $N(\log N)^2$. Another way to improve on Algorithm S, using a linked data structure, gave us the list insertion method, which does about $\frac{1}{4}N^2$ comparisons, 0 moves, and $2N$ changes of links.

Is it possible to marry the best features of these methods, reducing the number of comparisons to order $N \log N$ as in binary insertion, yet reducing the number of moves as in list insertion? The answer is yes, by going to a tree-structured arrangement. This possibility was first explored about 1957 by D. J. Wheeler, who suggested using two-way insertion until it becomes necessary to move some data; then instead of moving the data, a pointer to another area of memory is inserted, and the same technique is applied recursively to all items which are to be inserted into this new area of memory. Wheeler's original method [see A. S. Douglas, *Comp. J.* **2** (1959), 5] is a complicated combination of sequential and linked memory, with nodes of varying size; for our 16 example numbers the tree of Fig. 13 would be formed. A similar but simpler tree-insertion

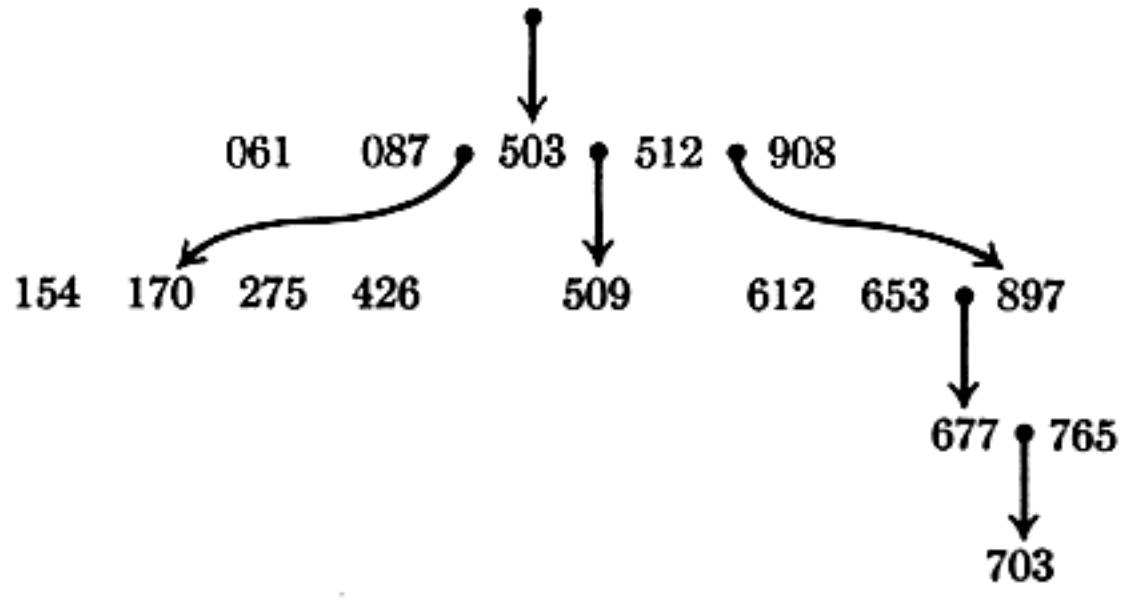


Fig. 13. Example of Wheeler's tree insertion scheme.

scheme, using binary trees, was independently devised by C. M. Berners-Lee about 1958 [see *Comp. J.* **3** (1960), 174, 184]. Since this method and its refinements are quite important for searching as well as sorting, they are discussed at length in Chapter 6.

Still another way to improve on straight insertion is to consider inserting several things at a time. For example, if we have a file of 1000 items, and if 998 of them have already been sorted, Algorithm S makes two more passes through the file (first inserting R_{999}, then R_{1000}). We can obviously save time if we first compare K_{999} with K_{1000}, to see which is larger, then insert them *both* with one look at the file. A combined operation of this kind involves about

$\frac{2}{3}N$ comparisons and moves (cf. exercise 3.4.2–5), instead of two passes each with about $\frac{1}{2}N$ comparisons and moves.

In other words, it is generally a good idea to "batch" operations that require long searches, so that multiple operations can be done together. If we carry this idea to its natural conclusion, we rediscover the method of sorting by merging, which is so important it is discussed below in a separate section.

Address calculation sorting. Surely by now we have exhausted all possible ways to improve on the simple method of straight insertion; but let's look again! Suppose someone gives you a blank sheet of paper and says that he is going to read you some words. Your job is to alphabetize the words, and to give him the sorted list on the paper. Then if he tells you a word starting with A, you will tend to write it near the top of the page, while a word starting with Z will probably be placed near the bottom, and so on. A similar technique is used when putting books into a bookshelf in order by the authors' names, when the books are given to you in random order on the floor: you estimate the final position of each book as you put it in place, thereby reducing the number of comparisons and moves you have to make. (The whole process is somewhat more efficient if you start with a little more shelf space than is absolutely necessary.) This method was first suggested for computer sorting by Isaac and Singleton, *JACM* **3** (1956), 169–174, and it has been further developed by Kronmal and Tartar, *Proc. ACM Nat'l Conf.* **21** (1966), 331–337.

Address calculation sorting usually requires additional storage space proportional to N, either to leave enough room so that excessive moving is not required, or to maintain auxiliary tables which help account for irregularities in the distribution of keys. (See the "distribution counting" sort, Algorithm 5.2D, which is a form of address calculation.) We can probably make best use of this additional memory space if we devote it to link fields, as in the list insertion method. In this way we can also avoid having separate areas for input and output; everything can be done in the same area of memory.

The basic idea is to generalize list insertion so that *several* lists are kept, not just one. Each list is used for certain ranges of keys. We make the important assumption that the keys are pretty evenly distributed, not "bunched up" irregularly: the set of all possible values of the keys is partitioned into M parts, and we assume a probability of $1/M$ that a given key falls into a given part. Then we provide additional storage for M list heads, and each list is maintained as in simple list insertion.

It is not necessary to give the algorithm in great detail here; the method is simply to begin with all list heads set to Λ. As each new item enters, we first decide which of the M parts its key falls into, then we insert it into the corresponding list as in Algorithm L.

To illustrate the method, suppose that the 16 keys used in our examples are divided into the $M = 4$ ranges 0–249, 250–499, 500–749, 750–999. We obtain the following configurations as sorting proceeds:

	4 items entered	8 items entered	12 items entered	Final state
List 1:	061, 087	061, 087, 170	061, 087, 154, 170	061, 087, 154, 170
List 2:		275	275, 426	275, 426
List 3:	503, 512	503, 512	503, 509, 512, 653	503, 509, 512, 612, 653, 677, 703
List 4:		897, 908	897, 908	765, 897, 908

Because linked memory is used, there is no storage allocation problem caused by varying-length lists.

Program M (*Multiple list insertion*). In this program we make the same assumptions as in Program L, except that the keys must be *nonnegative*, thus

$$0 \le \texttt{KEY} < (\texttt{BYTESIZE})^3.$$

The program divides this range into M equal parts by multiplying each key by a suitable constant. The list heads are in locations HEAD+1 through HEAD+M.

```
01    KEY     EQU   1:3
02    LINK    EQU   4:5
03    START   ENT2  M                    1
04            STZ   HEAD,2                M      HEAD[p] ← Λ.
05            DEC2  1                     M
06            J2P   *-2                   M      M ≥ p ≥ 1.
07            ENT1  N                     1      j ← N.
08    2H      LDA   INPUT,1(KEY)          N
09            MUL   =M(1:3)=              N      rA ← ⌊M · K_j/BYTESIZE^3⌋
10            STA   *+1(1:2)              N
11            ENT3  0                     N      q ← rA.
12            INC3  HEAD+1-INPUT          N      q ← LOC(HEAD[q]).
13            LDA   INPUT,1               N      K ← K_j.
14            JMP   4F                    N      To set p.
15    3H      CMPA  INPUT,2(KEY)          B+N-A
16            JLE   5F                    B+N-A  To insert, if K ≤ K_p.
17            ENT3  0,2                   B      q ← p.
18    4H      LD2   INPUT,3(LINK)         B+N    p ← LINK(q).
19            J2P   3B                    B+N    Jump if not end of list.
20    5H      ST1   INPUT,3(LINK)         N      LINK(q) ← LOC(R_j).
21            ST2   INPUT,1(LINK)         N      LINK(LOC(R_j)) ← p.
22    6H      DEC1  1                     N
23            J1P   2B                    N      N ≥ j ≥ 1. ∎
```

This program is written for general M, but it would be better to fix M at some convenient value; for example, we might choose $M = \texttt{BYTESIZE}$, so that the list heads could be cleared with a single MOVE instruction and the multiplication sequence of lines 08–11 could be replaced by the single instruction

`LD3 INPUT,1 (1:1)`. The most notable contrast between Program L and Program M is the fact that Program M must consider the case of an empty list, when no comparisons are to be made.

How much time do we save by having M lists? The total running time of Program M is $7B + 31N - 3A + 4M + 2$ units, where M is the number of lists, N is the number of records sorted, and A, B respectively count the right-to-left maxima and the inversions present among the keys belonging to each list. (In contrast to other time analyses of this section, we are here treating the rightmost element of a nonempty permutation as a "right-to-left maximum," instead of ignoring it.) We have already studied A and B for $M = 1$, when their average values are respectively H_N and $\frac{1}{2}\binom{N}{2}$. By our assumption about the distribution of keys, the probability that a given list contains precisely n items at the conclusion of sorting is the "binomial" probability

$$\binom{N}{n}\left(\frac{1}{M}\right)^n\left(1-\frac{1}{M}\right)^{N-n} \tag{10}$$

Therefore the average values of A and B in the general case are

$$A_{\text{ave}} = M\sum_n \binom{N}{n}\left(\frac{1}{M}\right)^n\left(1-\frac{1}{M}\right)^{N-n} H_n; \tag{11}$$

$$B_{\text{ave}} = M\sum_n \binom{N}{n}\left(\frac{1}{M}\right)^n\left(1-\frac{1}{M}\right)^{N-n} \binom{n}{2}. \tag{12}$$

By Theorem 1.2.7A,

$$\sum_n \binom{N}{n}(M-1)^{-n}H_n = \left(1-\frac{1}{M}\right)^{-N}(H_N - \ln M) + \epsilon,$$

$$0 < \epsilon = \sum_{n>N} \frac{1}{n}\left(1-\frac{1}{M}\right)^{n-N} < \frac{M-1}{N+1};$$

hence

$$A_{\text{ave}} = M(H_N - \ln M) + \delta, \qquad 0 < \delta < \frac{M^2}{N+1}\left(1-\frac{1}{M}\right)^{N+1}. \tag{13}$$

(This formula is practically useless when $M \approx N$. A more detailed discussion of the asymptotic behavior of A_{ave}, when $M = N/\alpha$, appears in exercise 5.2.2–57.) Using the identity

$$\binom{N}{n}\binom{n}{2} = \binom{N}{2}\binom{N-2}{n-2},$$

which is a special case of Eq. 1.2.6–20, the sum in (11) is easily evaluated, and we obtain

$$B_{\text{ave}} = \frac{1}{2M}\binom{N}{2}. \tag{14}$$

(For the standard deviation of B, see exercise 37.) Hence the total running time of Program M, for fixed M as $N \to \infty$, is

$$\begin{aligned} \min &\quad 31N + M + 2, \\ \text{ave} &\quad 1.75N^2/M + 31N - 3MH_N + 3M \ln M + 4M - 3 - 1.75N/M + 2, \\ \max &\quad 3.50N^2 + 24.5N + 4M + 2. \end{aligned} \tag{15}$$

Note that when M is not too large *we are speeding up the average time by a factor of* M; $M = 10$ will sort about ten times as fast as $M = 1$! However, the maximum time is much larger than the average time; this reiterates the assumption we have made about a fairly equal distribution of keys, since the worst case occurs when all records pile onto the same list.

If we set $M = N$, the average running time is approximately $34.36N$; when $M = \frac{1}{2}N$ it is slightly more, approximately $34.52N$; and when $M = \frac{1}{10}N$ it is approximately $48.04N$. (Note that $10N$ of these MIX time units are spent in the multiplication instruction alone!) *We have achieved a sorting method of order* N, *provided only that the keys are reasonably well spread out over their range.*

EXERCISES

1. [*10*] Is Algorithm S a "stable" sorting algorithm?

2. [*11*] Would Algorithm S still sort numbers correctly if the relation "$K \geq K_i$" in step S3 were replaced by "$K > K_i$"?

▶ **3.** [*30*] Is Program S the shortest possible sorting program that can be written for MIX, or is there a shorter program which achieves the same effect?

▶ **4.** [*M20*] Find the minimum and maximum running times for Program S, as a function of N.

▶ **5.** [*M27*] Find the generating function $g_N(z) = \sum_{k\geq 0} p_{Nk}z^k$ for the total running time of Program S, where p_{Nk} is the probability that Program S takes exactly k units of time, given a random permutation of $\{1, 2, \ldots, N\}$ as input. Also calculate the standard deviation of the running time, given N.

6. [*23*] The two-way insertion method illustrated in Table 2 seems to imply that there is an output area capable of holding up to $2N + 1$ records, in addition to the input area containing N records. Show that two-way insertion can be done using only enough space for $N + 1$ records, including both input and output.

7. [*M20*] If $a_1\, a_2 \ldots a_n$ is a random permutation of $\{1, 2, \ldots, n\}$, what is the average value of $|a_1 - 1| + |a_2 - 2| + \cdots + |a_n - n|$? (This is n times the average net distance traveled by a record during a sorting process.)

8. [*10*] Is Algorithm D a "stable" sorting algorithm?

9. [*20*] What are the quantities A and B, and the total running time of Program D, corresponding to Tables 3 and 4? Discuss the relative merits of Shell's method versus straight insertion in this case.

▶ **10.** [*22*] If $K_j \geq K_{j-h}$ in step D3, Algorithm D specifies a lot of actions that accomplish nothing. Show how to modify Program D so that this redundant computation can be avoided, and discuss the merits of such a modification.

11. [*M10*] What path in a lattice like Fig. 11 corresponds to the permutation 1 2 5 3 7 4 8 6 9 11 10 12?

12. [*M20*] Prove that the sum of the vertical weights on a lattice path equals the number of inversions in the corresponding 2-ordered permutation.

▸ **13.** [*M22*] Explain how to put weights on the *horizontal* line segments of a lattice, instead of the vertical segments, so that the sum of the horizontal weights on a lattice path is the number of inversions in the corresponding 2-ordered permutation.

14. [*M24*] (a) Show that, in the sums defined by Eq. (2), $A_{2n+1} = 2A_{2n}$. (b) The general identity of exercise 1.2.6–26 simplifies to

$$\sum_k \binom{2k+s}{k} z^k = \frac{1}{\sqrt{1-4z}} \left(\frac{1-\sqrt{1-4z}}{2z} \right)^s$$

if we set $r = -s-1$, $t = 1$. By considering the sum $\sum_n A_{2n} z^n$, show that

$$A_{2n} = n \cdot 4^{n-1}.$$

▸ **15.** [*HM33*] Let $g_n(z)$, $\bar{g}_n(z)$, $h_n(z)$, $\bar{h}_n(z)$ be $\sum z^{\text{total weight of path}}$ summed over all lattice paths of length $2n$ from $(0, 0)$ to (n, n), subject to certain restrictions on the vertices on the paths: For $h_n(z)$, there is no restriction, but for $g_n(z)$ the path must avoid all vertices (i, j) with $i > j$; $\bar{h}_n(z)$ and $\bar{g}_n(z)$ are defined similarly, except that all vertices (i, i) are also excluded, for $0 < i < n$. Thus

$$g_0(z) = 1, \quad g_1(z) = z, \quad g_2(z) = z^3 + z^2; \quad \bar{g}_1(z) = z, \quad \bar{g}_2(z) = z^3;$$
$$h_0(z) = 1, \quad h_1(z) = z+1, \quad h_2(z) = z^3 + z^2 + 3z + 1;$$
$$\bar{h}_1(z) = z+1, \quad \bar{h}_2(z) = z^3 + z.$$

Find recurrence relations defining these functions, and use the recurrence relations to prove that

$$h_n''(1) + h_n'(1) = \frac{7n^3 + 4n^2 + 4n}{30} \binom{2n}{n}.$$

(The exact formula for the variance of the number of inversions in a random 2-ordered permutation of $\{1, 2, \ldots, 2n\}$ is therefore easily found; it is asymptotically $(\frac{7}{30} - \frac{\pi}{16})n^3$.)

16. [*M24*] Find a formula for the maximum number of inversions in an h-ordered permutation of $\{1, 2, \ldots, n\}$. What is the maximum possible number of moves in Algorithm D when the increments satisfy the divisibility condition (5)?

17. [*M21*] Show that, when $N = 2^t$ and $h_s = 2^{s-1}$ for $t \geq s \geq 1$, there is a unique permutation of $\{1, 2, \ldots, N\}$ which maximizes the number of move operations performed by Algorithm D. Find a simple way to describe this permutation.

18. [*HM24*] For large N the sum (6) can be estimated as

$$\frac{1}{4} \frac{N^2}{h_t} + \frac{\sqrt{\pi}}{8} \left(\frac{N^{3/2} h_t^{1/2}}{h_{t-1}} + \cdots + \frac{N^{3/2} h_2^{1/2}}{h_1} \right).$$

What real values of $h_t, \ldots, h_1$ minimize this expression when N and t are fixed and $h_1 = 1$?

▸ **19.** [*M25*] What is the average value of the quantity A in the timing analysis of Program D, when the increments satisfy the divisibility condition (5)?

20. [*M20*] Show that Theorem K follows from Lemma L.

21. [*M25*] Let h and k be relatively prime positive integers. Show that the largest integer which cannot be represented in the form $ah + bk$, when a and b are *nonnegative* integers, is $hk - h - k$. Hence if a file is both h-ordered and k-ordered, we have $K_i \leq K_j$ whenever $j - i \geq (h - 1)(k - 1)$.

22. [*M30*] Prove that all integers $\geq 2^h(2^h - 1)$ can be represented in the form $a_0(2^h - 1) + a_1(2^{h+1} - 1) + a_2(2^{h+2} - 1) + \cdots$, where the a_j's are nonnegative integers; but $2^h(2^h - 1) - 1$ cannot be so represented. Furthermore, exactly $2^{h-1}(2^h + h - 3)$ positive integers are unrepresentable in this form.

Find analogous formulas when the quantities $2^k - 1$ are replaced by $2^k + 1$ in the representations.

▸ **23.** [*M22*] Prove that if h_{s+2} and h_{s+1} are relatively prime, the number of moves which occur while Algorithm D is using the increment h_s is $O(Nh_{s+2}h_{s+1}/h_s)$. *Hint:* See exercise 21.

24. [*M42*] Prove that Theorem P is best possible, in the sense that the exponent 3/2 cannot be lowered.

▸ **25.** [*M22*] How many permutations of $\{1, 2, \ldots, n\}$ are both 3-ordered and 2-ordered? What is the maximum number of inversions in such a permutation? What is the total number of inversions among all such permutations?

26. [*M35*] Can a file of N elements have more than N inversions if it is 3-, 5-, and 7-ordered?

27. [*M50*] Find an efficient algorithm to construct permutations which maximize the number of moves in Algorithm D, given N, h_t, h_{t-1}, $\ldots$, h_1.

28. [*15*] Which sequence of increments shown in Table 6 is best from the standpoint of Program D, considering the total running time?

29. [*M42*] For $N = 1000$ and various values of t, find empirical values of $h_t, \ldots, h_1$ which seem to minimize the average number of moves made by Algorithm D, in the sense that, if any one of the h's is varied while the others are fixed, the average number of moves increases.

30. [*M23*] (V. Pratt.) If the set of increments is $\{2^p3^q \mid 2^p3^q < N\}$, show that the number of passes is approximately $\frac{1}{2}(\log_2 N)(\log_3 N)$, and the number of moves per pass is at most $N/2$. In fact, if $K_{j-h} > K_j$ on any pass, we will always have K_{j-3h}, $K_{j-2h} \leq K_j < K_{j-h} \leq K_{j+h}$, K_{j+2h}; so we may simply interchange K_{j-h} and K_j and increase j by $2h$, saving two of the comparisons of Algorithm D. *Hint:* See exercise 25.

▸ **31.** [*25*] Write a MIX program for Pratt's sorting algorithm (exercise 30). Express its running time in terms of quantities A, B, S, T, N analogous to those in Program D.

32. [*10*] What would be the final contents of $L_0\, L_1 \ldots L_{16}$ if the list insertion sort in Table 8 were carried through to completion?

▸ **33.** [*25*] Find a way to improve on Program L so that its running time is dominated by $5B$ instead of $7B$, where B is the number of inversions.

34. [*M10*] Verify formula (10).

35. [*18*] Assume that the byte size of MIX is 100, and that the sixteen example keys in Table 8 are actually 503000, 087000, 512000, . . . , 703000. Determine the running time of Programs L and M on this data, when $M = 4$.

36. [*HM16*] If a program runs in approximately $A/M + B$ units of time and uses $C + M$ locations in memory, what choice of M gives the optimum product of space times time?

37. [*M25*] Let $g_n(z)$ be the generating function for inversions in a random permutation of n objects (cf. Eq. 5.1.1–11). Let $g_{NM}(z)$ be the corresponding generating function for the quantity B in Program M. Show that

$$\sum_{N\geq 0} g_{NM}(z)M^N w^N/N! = \left(\sum_{n\geq 0} g_n(z)w^n/n!\right)^M,$$

and use this formula to derive the variance of B.

38. [*HM23*] (R. M. Karp.) Let $F(x)$ be a distribution function for a probability distribution, with $F(0) = 0$ and $F(1) = 1$. Given that the keys $K_1, K_2, \ldots, K_N$ are independently chosen at random from this distribution, and that $M = cN$, where c is constant and $N \to \infty$, prove that the running time of Program M is $O(N)$ when F is sufficiently smooth. (A key K is inserted into list j when $\lfloor MK \rfloor = j - 1$; this occurs with probability $F(j/M) - F((j-1)/M)$. Only the case $F(x) = x, 0 \leq x \leq 1$, is treated in the text.)

39. [*M25*] (*Disorder in a library.*) Casual users of a library often put books back on the shelves in the wrong place. One way to measure the amount of disorder present in a library is to consider the minimum number of times we would have to take a book out of one place and insert it in another, before all books are restored to the correct order.

Thus let $\pi = a_1\, a_2 \ldots a_n$ be a permutation of $\{1, 2, \ldots, n\}$. A "deletion-insertion operation" changes π to

$$a_1 \ldots a_{i-1} a_{i+1} \ldots a_j\, a_i\, a_{j+1} \ldots a_n \quad \text{or to} \quad a_1 \ldots a_j\, a_i\, a_{j+1} \ldots a_{i-1}\, a_{i+1} \ldots a_n,$$

for some i and j. Let dis (π) be the minimum number of deletion-insertion operations which will sort π into order. Can dis (π) be expressed in terms of simpler characteristics of π?

40. [*40*] Experiment with the following variation of the diminishing increment sort, intended to speed up the "inner loop": For $s = t, t-1, \ldots, 2$, and for $j = h_s + 1, h_s + 2, \ldots, N$, interchange $R_{j-h_s} \leftrightarrow R_j$ if $K_{j-h_s} > K_j$. (Thus, the result of interchanges is not propagated; no comparison of K_{j-h_s}: K_{j-2h_s} is made. The records will not always be h_s-sorted, but these preliminary passes will help to decrease the number of inversions.) Finally apply straight insertion to complete the sort.

5.2.2. Sorting by Exchanging

We come now to the second family of sorting algorithms mentioned near the beginning of Section 5.2: "exchange" or "transposition" methods which systematically interchange pairs of elements that are out of order until no more such pairs exist.

The process of straight insertion, Algorithm 5.2.1S, can be viewed as an exchange method: We take each new record R_j and essentially exchange it with its neighbors to the left until it has been inserted into the proper place. So the classification of sorting methods into various families such as "insertion," "exchange," "selection," etc., is not always clear-cut. In this section, we shall discuss four types of sorting methods for which exchanging is a dominant characteristic: *exchange selection* (the "bubble sort"); *merge exchange* (Batcher's parallel sort); *partition exchange* (Hoare's "quicksort"); and *radix exchange*.

The bubble sort. Perhaps the most obvious way to sort by exchanges is to compare K_1 with K_2, interchanging R_1 and R_2 if the keys are out of order; then do the same to records R_2 and R_3, R_3 and R_4, etc. During this sequence of operations, records with large keys tend to move to the right, and in fact the record with the largest key will move up to become R_N. Repetitions of the process will get the appropriate records into positions R_{N-1}, R_{N-2}, etc., so that all records will ultimately be sorted.

Figure 14 shows this sorting method in action on the sixteen keys 503 087 512 . . . 703; it is convenient to represent the file of numbers vertically instead of horizontally, with R_N at the top and R_1 at the bottom. The method is called

	Pass 1	Pass 2	Pass 3	Pass 4	Pass 5	Pass 6	Pass 7	Pass 8	Pass 9
703	908	908	908	908	908	908	908	908	908
765	703	897	897	897	897	897	897	897	897
677	765	703	765	765	765	765	765	765	765
612	677	765	703	703	703	703	703	703	703
509	612	677	677	677	677	677	677	677	677
154	509	612	653	653	653	653	653	653	653
426	154	509	612	612	612	612	612	612	612
653	426	154	509	512	512	512	512	512	512
275	653	426	154	509	509	509	509	509	509
897	275	653	426	154	503	503	503	503	503
170	897	275	512	426	154	426	426	426	426
908	170	512	275	503	426	154	275	275	275
061	512	170	503	275	275	275	154	170	170
512	061	503	170	170	170	170	170	154	154
087	503	061	087	087	087	087	087	087	087
503	087	087	061	061	061	061	061	061	061

Fig. 14. The bubble sort in action.

"bubble sorting" because large elements "bubble up" to their proper position, by contrast with the "sinking sort" (i.e., straight insertion) in which elements sink down to an appropriate level. The bubble sort is also known by more prosaic names such as "exchange selection" or "propagation."

After each pass through the file, it is not hard to see that all records above and including the last one to be exchanged must be in their final position, so they need not be examined on subsequent passes. Horizontal lines in Fig. 14 show the progress of the sorting from this standpoint; note, for example, that five more elements are known to be in final position as a result of Pass 4. On the final pass, no exchanges are performed at all. With these observations we are ready to formulate the algorithm.

Algorithm B (*Bubble sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete their keys will be in order, $K_1 \le \cdots \le K_N$.

B1. [Initialize BOUND.] Set BOUND $\leftarrow N$. (BOUND is the highest index for which the record is not known to be in its final position; thus we are indicating that nothing is known at this point.)

B2. [Loop on j.] Set $t \leftarrow 0$. Perform step B3 for $j = 1, 2, \ldots,$ BOUND-1, and then go to step B4.

B3. [Compare/exchange R_j: R_{j+1}.] If $K_j > K_{j+1}$, interchange $R_j \leftrightarrow R_{j+1}$ and set $t \leftarrow j$.

B4. [Any exchanges?] If $t = 0$, the algorithm terminates. Otherwise set BOUND $\leftarrow t$ and return to step B2. ∎

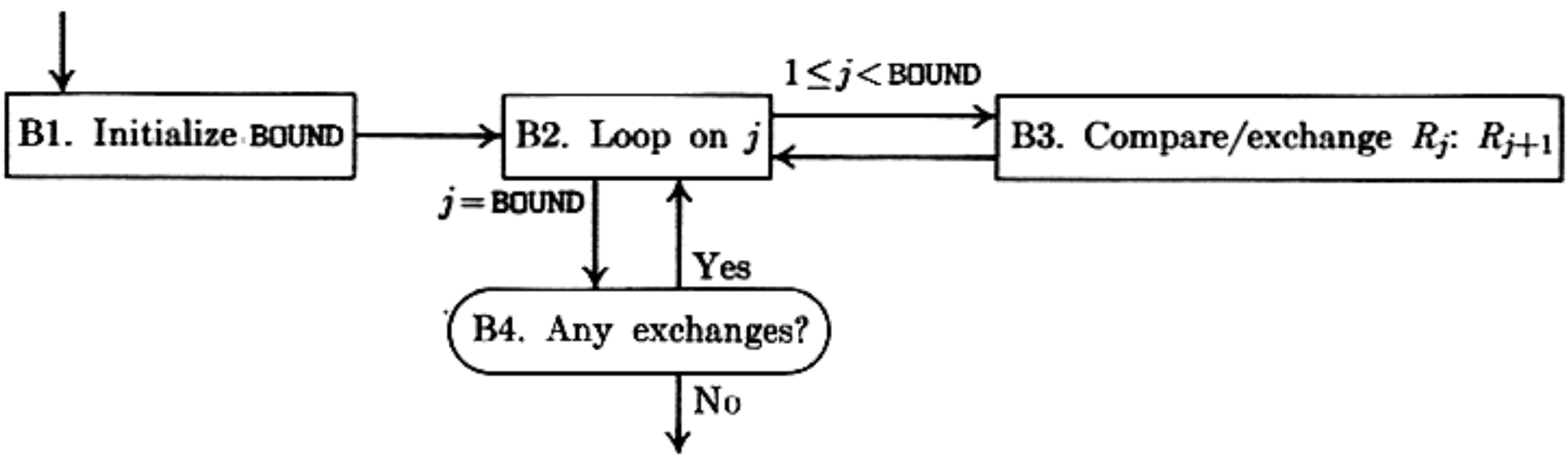


Fig. 15. Flowchart for bubble sorting.

Program B (*Bubble sort*). As in previous MIX programs of this chapter, we assume that the items to be sorted are in locations INPUT+1 through INPUT+N. rI1 $\equiv t$; rI2 $\equiv j$.

```
01  START  ENT1  N              1   B1. Initialize BOUND. t ← N.
02  1H     ST1   BOUND(1:2)     A   BOUND ← t.
03         ENT2  1              A   B2. Loop on j.
04         ENT1  0              A   t ← 0.
05         JMP   BOUND          A   Exit if j ≥ BOUND.
```

06	3H	LDA	INPUT,2	C	*B3. Compare/exchange R_j:R_{j+1}.*
07		CMPA	INPUT+1,2	C	
08		JLE	2F	C	No exchange if $K_j \leq K_{j+1}$.
09		LDX	INPUT+1,2	B	R_{j+1}
10		STX	INPUT,2	B	$\rightarrow R_j$.
11		STA	INPUT+1,2	B	(old R_j) $\rightarrow R_{j+1}$.
12		ENT1	0,2	B	$t \leftarrow j$.
13	2H	INC2	1	C	$j \leftarrow j + 1$.
14	BOUND	ENTX	-*,2	$A + C$	rX $\leftarrow j -$ BOUND. (Instruction modified)
15		JXN	3B	$A + C$	$1 \leq j <$ BOUND.
16	4H	J1P	1B	A	*B4. Any exchanges?* To B2 if $t > 0$. ∎

Analysis of the bubble sort. It is quite instructive to analyze the running time of Algorithm B. Three quantities are involved in the timing: the number of passes, A; the number of exchanges, B; and the number of comparisons, C. If the input keys are distinct and in random order, we may assume that they form a random permutation of $1, 2, \ldots, n$. The idea of *inversion tables* (Section 5.1.1) leads to an easy way to describe the effect of each pass in a bubble sort.

Theorem I. *Let $a_1\ a_2 \ldots a_n$ be a permutation of $\{1, 2, \ldots, n\}$, and let $b_1\ b_2 \ldots b_n$ be the corresponding inversion table. If one pass of the bubble sort, Algorithm B, changes $a_1\ a_2 \ldots a_n$ to the permutation $a'_1\ a'_2 \ldots a'_n$, the corresponding inversion table $b'_1\ b'_2 \ldots b'_n$ is obtained from $b_1\ b_2 \ldots b_n$ by decreasing each nonzero entry by* 1.

Proof. If a_i is preceded by a larger element, the largest preceding element is exchanged with it, so b_{a_i} decreases by 1. On the other hand if a_i is not preceded by a larger element, it is never exchanged with a larger element, so b_{a_i} remains 0. ∎

Thus we can see what happens during a bubble sort by studying the sequence of inversion tables betweeen passes. For example, the successive inversion tables corresponding to Fig. 14 are

	3	1	8	3	4	5	0	4	0	3	2	2	3	2	1	0	
Pass 1																	
	2	0	7	2	3	4	0	3	0	2	1	1	2	1	0	0	
Pass 2																	
	1	0	6	1	2	3	0	2	0	1	0	0	1	0	0	0	
Pass 3																	
	0	0	5	0	1	2	0	1	0	0	0	0	0	0	0	0	(1)

and so on. If $b_1\ b_2 \ldots b_n$ is the inversion table of the input permutation, we must therefore have

$$A = 1 + \max(b_1, b_2, \ldots, b_n); \tag{2}$$

$$B = b_1 + b_2 + \cdots + b_n; \tag{3}$$

$$C = c_1 + c_2 + \cdots + c_A, \tag{4}$$

where c_j is the value of BOUND–1 at the beginning of pass j. In terms of the inversion table,

$$c_j = \max \{b_i + i \mid b_i \geq j - 1\} - j \tag{5}$$

(see exercise 5). In example (1) we therefore have $A = 9$, $B = 41$, $C = 15 + 14 + 13 + 12 + 7 + 5 + 4 + 3 + 2 = 75$. The total MIX sorting time for Fig. 14 is $1030u$.

The distribution of B (the total number of inversions in a random permutation) is very well-known to us by now; so we are left with A and C to be analyzed.

The probability that $A \leq k$ is $1/n!$ times the number of inversion tables having no components $\geq k$, namely $k^{n-k}k!$, when $1 \leq k \leq n$. Hence the probability that exactly k passes are required is

$$A_k = \frac{1}{n!}(k^{n-k}k! - (k-1)^{n-k+1}(k-1)!). \tag{6}$$

The mean value $\sum kA_k$ can therefore be calculated; summing by parts, it is

$$A_{\text{ave}} = n + 1 - \sum_{0 \leq k \leq n} \frac{k^{n-k}k!}{n!} = n + 1 - P(n), \tag{7}$$

where $P(n)$ is the function whose asymptotic value was found to be $\sqrt{\pi n/2} - \frac{2}{3} + O(1/\sqrt{n})$ in Section 1.2.11, Eq. (24). Formula (7) was stated without proof by E. H. Friend in *JACM* **3** (1956), 150; a proof was given by Howard B. Demuth [Ph. D. thesis (Stanford University: October, 1956), 64–68]. For the standard deviation of A, see exercise 7.

The total number of comparisons, C, is somewhat harder to handle, and we will consider only C_{ave}. For fixed n, let $f_j(k)$ be the number of inversion tables $b_1 \ldots b_n$ such that for $1 \leq i \leq n$ we have either $b_i < j - 1$ or $b_i + i - j \leq k$; then

$$f_j(k) = (j + k)!(j - 1)^{n-j-k}, \qquad \text{for} \qquad 0 \leq k \leq n - j. \tag{8}$$

(See exercise 8.) The average value of c_j in (5) is $(\sum k(f_j(k) - f_j(k-1)))/n!$; summing by parts and then summing on j leads to the formula

$$C_{\text{ave}} = \binom{n+1}{2} - \frac{1}{n!} \sum_{\substack{1 \leq j \leq n \\ 0 \leq k \leq n-j}} f_j(k) = \binom{n+1}{2} - \frac{1}{n!} \sum_{0 \leq r < s \leq n} s!r^{n-s}. \tag{9}$$

Here the asymptotic value is not easy to determine, and we shall return to it at the end of this section.

To summarize our analysis of the bubble sort, the formulas derived above and below may be written as follows:

$$A = (\min 1, \quad \text{ave } N - \sqrt{\pi N/2} + O(1), \quad \max N); \tag{10}$$

$$B = (\min 0, \quad \text{ave } \tfrac{1}{4}(N^2 - N), \quad \max \tfrac{1}{2}(N^2 - N)); \tag{11}$$

$$C = (\min N - 1, \quad \text{ave } \tfrac{1}{2}(N^2 - N \ln N - (\gamma + \ln 2 - 1)N) + O(\sqrt{N}), \quad \max \tfrac{1}{2}(N^2 - N)). \tag{12}$$

In each case the minimum occurs when the input is already in order, and the maximum occurs when it is in reverse order; so the MIX running time is $8A + 7B + 8C + 1 = (\min 8N + 1, \text{ave } 5.75N^2 + O(N \ln N), \max 7.5N^2 + 0.5N + 1)$.

Refinements of the bubble sort. It took a good deal of work to analyze the bubble sort; and although the techniques used in the calculations are instructive, the results are disappointing since they tell us that the bubble sort isn't really very good at all. Compared to straight insertion (Algorithm 5.2.1S), bubble sorting requires a more complicated program and takes about twice as long!

Some suggestions can be given for improving the bubble sort. For example, in Fig. 14, the first comparison in Pass 4 is redundant, as are the first two in Pass 5 and the first three in Passes 6 and 7. Note also that elements can never move to the left more than one step per pass; so if the smallest item happens to be initially at the far right we are forced to make the maximum number of comparisons. This suggests the "cocktail shaker sort," in which alternate passes go in opposite directions (see Fig. 16). The average number of comparisons is slightly reduced by this approach. K. E. Iverson [*A Programming Language* (Wiley, 1962), 218–219] has made an interesting observation in this regard: If j is an index such that R_j and R_{j+1} are not exchanged with each

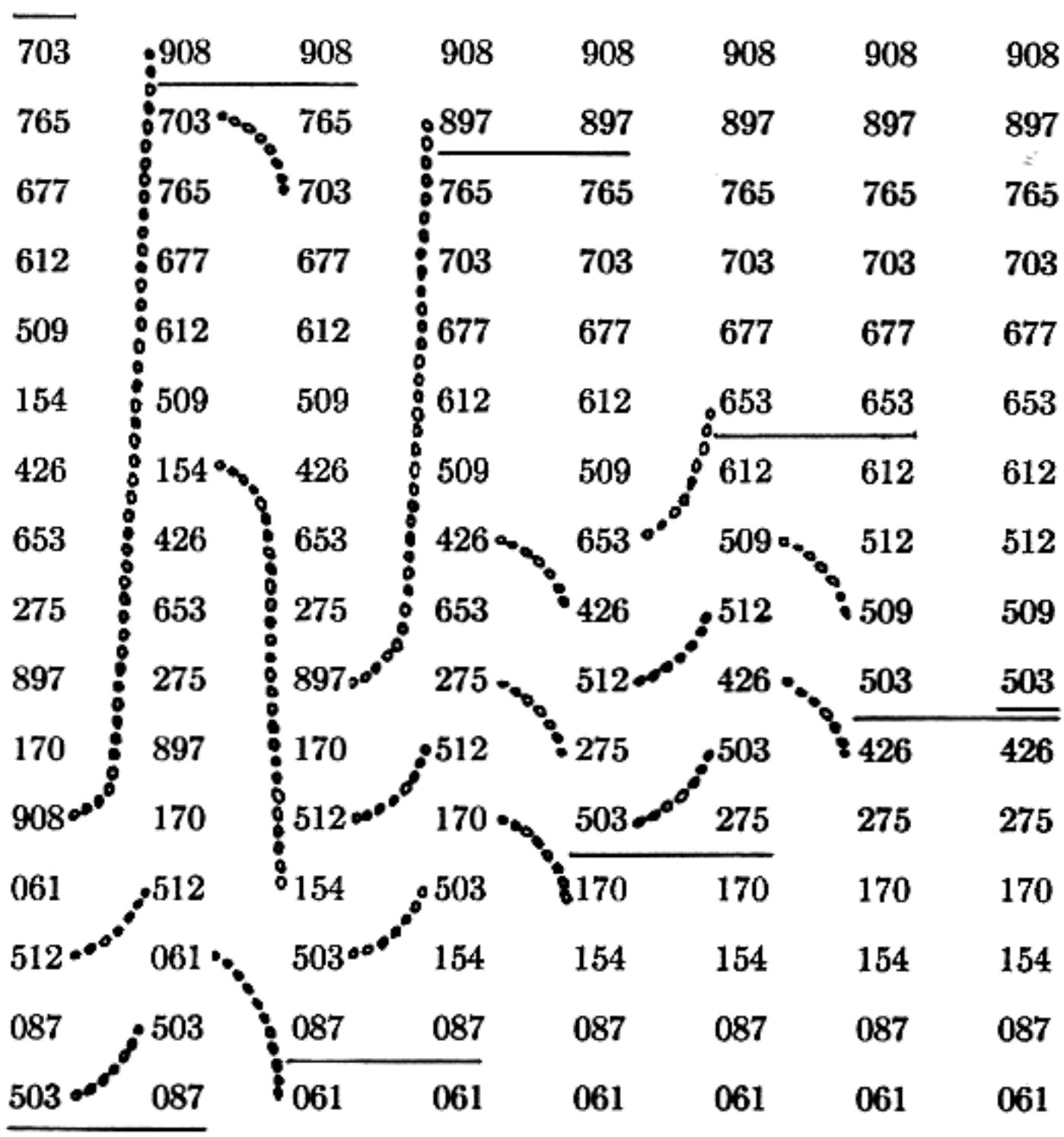


Fig. 16. The "cocktail-shaker sort."

other on two consecutive passes in opposite directions, then R_j and R_{j+1} must be in their final position, and they need not enter into any subsequent comparisons. For example, traversing 4 3 2 1 8 6 9 7 5 from left to right yields 3 2 1 4 6 8 7 5 9; no interchange occurred between R_4 and R_5. Traversing the latter permutation from right to left, R_4 is still less than (the new) R_5, so we may immediately conclude that R_4 and R_5 need not participate in any further comparisons.

But none of these refinements lead to an algorithm better than straight insertion (and we already know that straight insertion isn't suitable for large N). Another idea is to eliminate most of the exchanges; since most elements simply shift left one step during an exchange, we could achieve the same effect by viewing the array differently, shifting the origin of indexing! But the resulting algorithm is no better than straight *selection*, Algorithm 5.2.3S, which we shall study later.

In short, the bubble sort seems to have nothing to recommend it, except a catchy name and the fact that it leads to some interesting theoretical problems.

Batcher's parallel method. If we are going to have an exchange algorithm whose running time is faster than order N^2, we need to select some *nonadjacent* pairs of keys (K_i, K_j) for comparisons; otherwise we will need as many exchanges as the original permutation has inversions, and the average number of inversions is $\frac{1}{4}(N^2 - N)$. An ingenious way to program a sequence of comparisons, looking for potential exchanges, was discovered in 1964 by K. E. Batcher [*Proc. AFIPS Spring Joint Computer Conference* **32** (1968), 307–314]. His method is not at all obvious; in fact, it requires a fairly intricate proof just to show that it is valid, since comparatively few comparisons are made. We shall discuss two proofs, one in this section and another in Section 5.3.4.

Batcher's sorting scheme is somewhat like Shell's sort, but the comparisons are done in a novel way so that no propagation of exchanges is necessary. Thus we can compare Table 1 to Table 5.2.1–3; Batcher's method achieves the

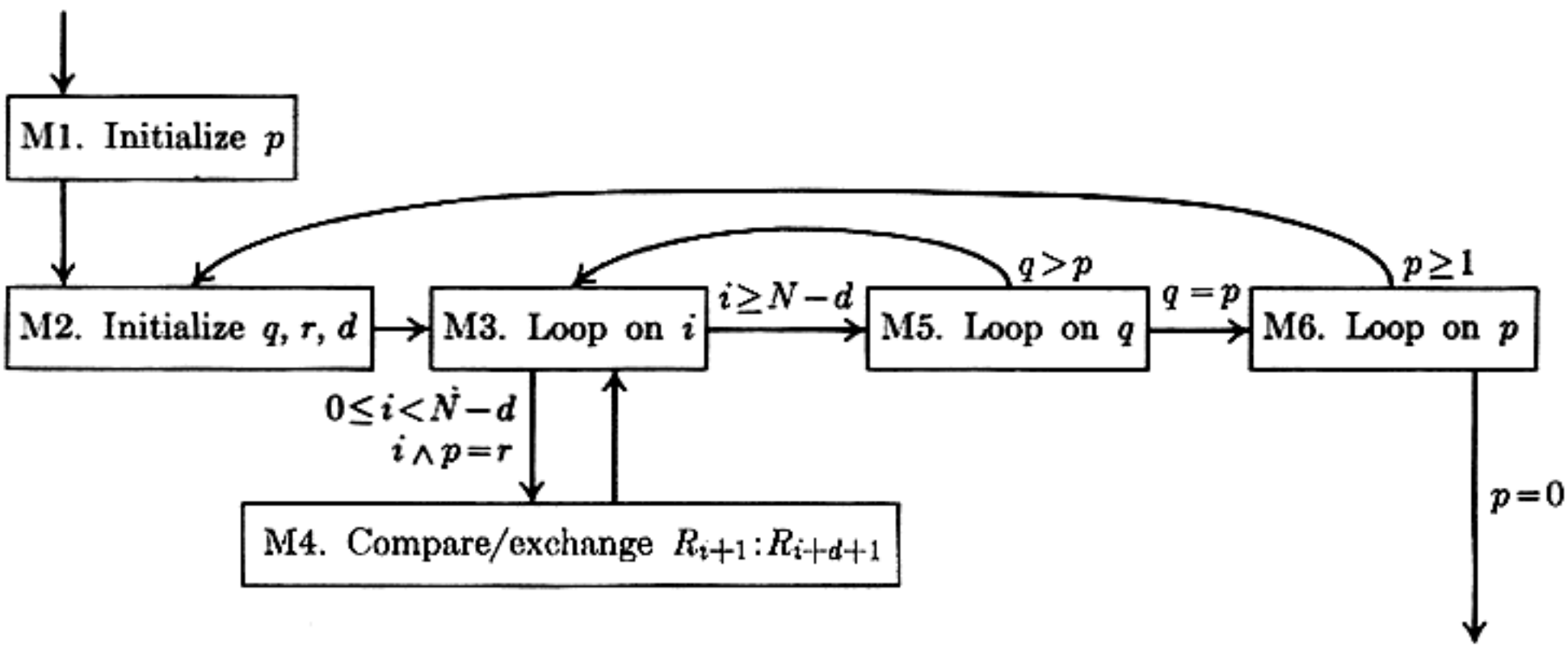


Fig 17. Algorithm M.

effect of 8-sorting, 4-sorting, 2-sorting, and 1-sorting, but the comparisons do not overlap. Since Batcher's algorithm essentially merges pairs of sorted subsequences, it may be called the "merge exchange sort."

Algorithm M (*Merge exchange*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete their keys will be in order, $K_1 \le \cdots \le K_N$. We assume that $N \ge 2$.

M1. [Initialize p.] Set $p \leftarrow 2^{t-1}$, where $t = \lceil \log_2 N \rceil$ is the least integer such that $2^t \ge N$. (Steps M2 through M5 will be performed for $p = 2^{t-1}$, $2^{t-2}, \ldots, 1$.)

M2. [Initialize q, r, d.] Set $q \leftarrow 2^{t-1}$, $r \leftarrow 0$, $d \leftarrow p$.

M3. [Loop on i.] For all i such that $0 \le i < N - d$ and $i \wedge p = r$, do step M4. Then go to step M5. (Here $i \wedge p$ means the "logical and" of the binary representations of i and p; each bit of the result is zero except where both i and p have 1 bits in corresponding positions. Thus $13 \wedge 21 = (1101)_2 \wedge (10101)_2 = (00101)_2 = 5$. At this point, d is an odd multiple of p, and p is a power of 2, so that $i \wedge p \ne (i + d) \wedge p$; it follows that the actions of step M4 can be done for all relevant i in any order, even simultaneously.)

M4. [Compare/exchange $R_{i+1} : R_{i+d+1}$.] If $K_{i+1} > K_{i+d+1}$, interchange $R_{i+1} \leftrightarrow R_{i+d+1}$.

M5. [Loop on q.] If $q \ne p$, set $d \leftarrow q - p$, $q \leftarrow q/2$, $r \leftarrow p$, and return to M3.

M6. [Loop on p.] (At this point the permutation $K_1\, K_2 \ldots K_N$ is p-ordered.) Set $p \leftarrow \lfloor p/2 \rfloor$. If $p > 0$, go back to M2. ∎

Table 1 illustrates the method for $N = 16$. Note that the algorithm sorts N elements essentially by sorting R_1, R_3, $R_5, \ldots$ and R_2, R_4, $R_6, \ldots$ independently; then we perform steps M2 through M5 for $p = 1$, in order to merge the two sorted sequences together.

In order to prove that the magic sequence of comparison/exchanges specified in Algorithm M actually will sort all possible input files $R_1\, R_2 \ldots R_N$, we must show only that steps M2 through M5 will merge all 2-ordered files $R_1\, R_2 \ldots R_N$ when $p = 1$. For this purpose we can use the lattice-path method of Section 5.2.1 (cf. Fig. 11); each 2-ordered permutation of $\{1, 2, \ldots, N\}$ corresponds uniquely to a path from $(0, 0)$ to $(\lceil N/2 \rceil, \lfloor N/2 \rfloor)$ in a lattice diagram. Figure 18(a) shows an example for $N = 16$, corresponding to the permutation 1 3 2 4 10 5 11 6 13 7 14 8 15 9 16 12. When we perform step M3 with $p = 1$, $q = 2^{t-1}$, $r = 0$, $d = 1$, the effect is to compare (and possibly exchange) $R_1 : R_2$, $R_3 : R_4$, etc. This operation corresponds to a simple transformation of the lattice path, "folding" it about the diagonal if necessary so that it never goes above the diagonal. (See Fig. 18(b) and the proof in exercise 10.) The next iterations of step M3 have $p = r = 1$, and $d = 2^{t-1} - 1, 2^{t-2} - 1, \ldots, 1$; their effect is to compare/exchange $R_2 : R_{2+d}$, $R_4 : R_{4+d}$, etc., and again there is a simple lattice interpretation: the path is "folded" about a line $\frac{1}{2}(d + 1)$ units below the diagonal. See Fig. 18(c) and (d); eventually we get to the path

in Fig. 18(e), which corresponds to a completely sorted permutation. This completes a "geometric proof" that Batcher's algorithm is valid; we might call it sorting by folding!

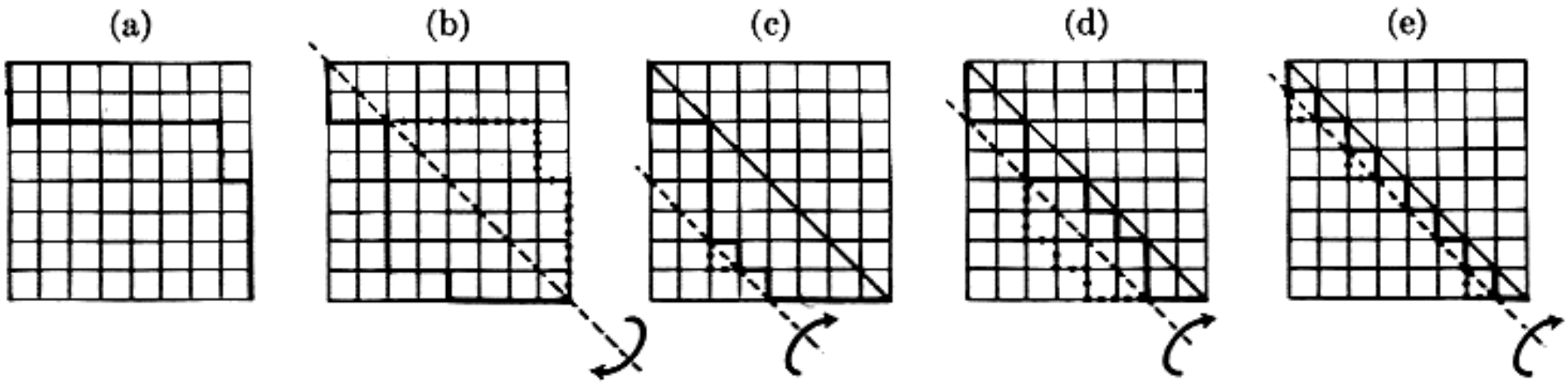


Fig. 18. A geometric interpretation of Batcher's method, $N = 16$.

Table 1

MERGE-EXCHANGE SORTING (BATCHER'S METHOD)

p	q	r	d	
				503 087 512 061 908 170 897 275 653 426 154 509 612 677 765 703
8	8	0	8	
				503 087 154 061 612 170 765 275 653 426 512 509 908 677 897 703
4	8	0	4	
				503 087 154 061 612 170 765 275 653 426 512 509 908 677 897 703
4	4	4	4	
				503 087 154 061 612 170 512 275 653 426 765 509 908 677 897 703
2	8	0	2	
				154 061 503 087 512 170 612 275 653 426 765 509 897 677 908 703
2	4	2	6	
				154 061 503 087 512 170 612 275 653 426 765 509 897 677 908 703
2	2	2	2	
				154 061 503 087 512 170 612 275 653 426 765 509 897 677 908 703
1	8	0	1	
				061 154 087 503 170 512 275 612 426 653 509 765 677 897 703 908
1	4	1	7	
				061 154 087 503 170 512 275 612 426 653 509 765 677 897 703 908
1	2	1	3	
				061 154 087 275 170 426 503 509 512 653 612 703 677 897 765 908
1	1	1	1	
				061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908

A MIX program for Algorithm M appears in exercise 12. Unfortunately the amount of bookkeeping needed to control the sequence of comparisons is rather large, so the program is less efficient than other methods we have seen. But it has one important redeeming feature: All comparison/exchanges specified by a given iteration of step M3 can be done *simultaneously*, on computers or logical networks which allow parallel computations. With such parallel operations, sorting is completed in $\frac{1}{2}\lceil \log_2 N \rceil (\lceil \log_2 N \rceil + 1)$ steps, and this is about as fast as any general method known. For example, *1024 elements can be sorted in only 55 parallel steps by Batcher's method.* The nearest competitor is Pratt's method (see exercise 5.2.1–30), which uses either 40 or 73 steps, depending on how we count; if we are willing to allow overlapping comparisons as long as no overlapping exchanges are necessary, Pratt's method requires only 40 comparison/exchange cycles to sort 1024 elements. For further comments, see Section 5.3.4.

"Quicksort." The sequence of comparisons in Batcher's method is predetermined; we compare the same pairs of keys each time, regardless of what we may have learned about the file from previous comparisons. The same is largely true of the bubble sort, although Algorithm B does make limited use of previous knowledge in order to reduce its work at the right end of the file. Let us now turn to a quite different strategy, which uses the result of each comparison to determine what keys are to be compared next. Such a strategy is inappropriate for parallel computations, but on computers which work serially it can be quite fruitful.

Consider therefore the following comparison/exchange scheme: Keep two pointers, i and j, with $i = 1$ and $j = N$ initially. Compare $K_i{:}K_j$, and if no exchange is necessary decrease j by 1 and repeat the process. After an exchange first occurs, increase i by 1, and continue comparing and increasing i until another exchange occurs. Then decrease j again, and so on, "burning the candle at both ends," until $i = j$. For example, consider what happens to our file of sixteen numbers.

Given:	**503** 087 512 061 908 170 897 275 653 426 154 509 612 677 765 **703**	
		Decrease j
1st exchange:	**154** 087 512 061 908 170 897 275 653 426 **503** 509 612 677 765 703	
		Increase i
2nd exchange:	154 087 **503** 061 908 170 897 275 653 426 **512** 509 612 677 765 703	
		Decrease j
3rd exchange:	154 087 **426** 061 908 170 897 275 653 **503** 512 509 612 677 765 703	
		Increase i
4th exchange:	154 087 426 061 **503** 170 897 275 653 **908** 512 509 612 677 765 703	
		Decrease j
5th exchange:	154 087 426 061 **275** 170 897 **503** 653 908 512 509 612 677 765 703	
		Increase i
6th exchange:	154 087 426 061 275 170 **503** **897** 653 908 512 509 612 677 765 703	
		Decrease j

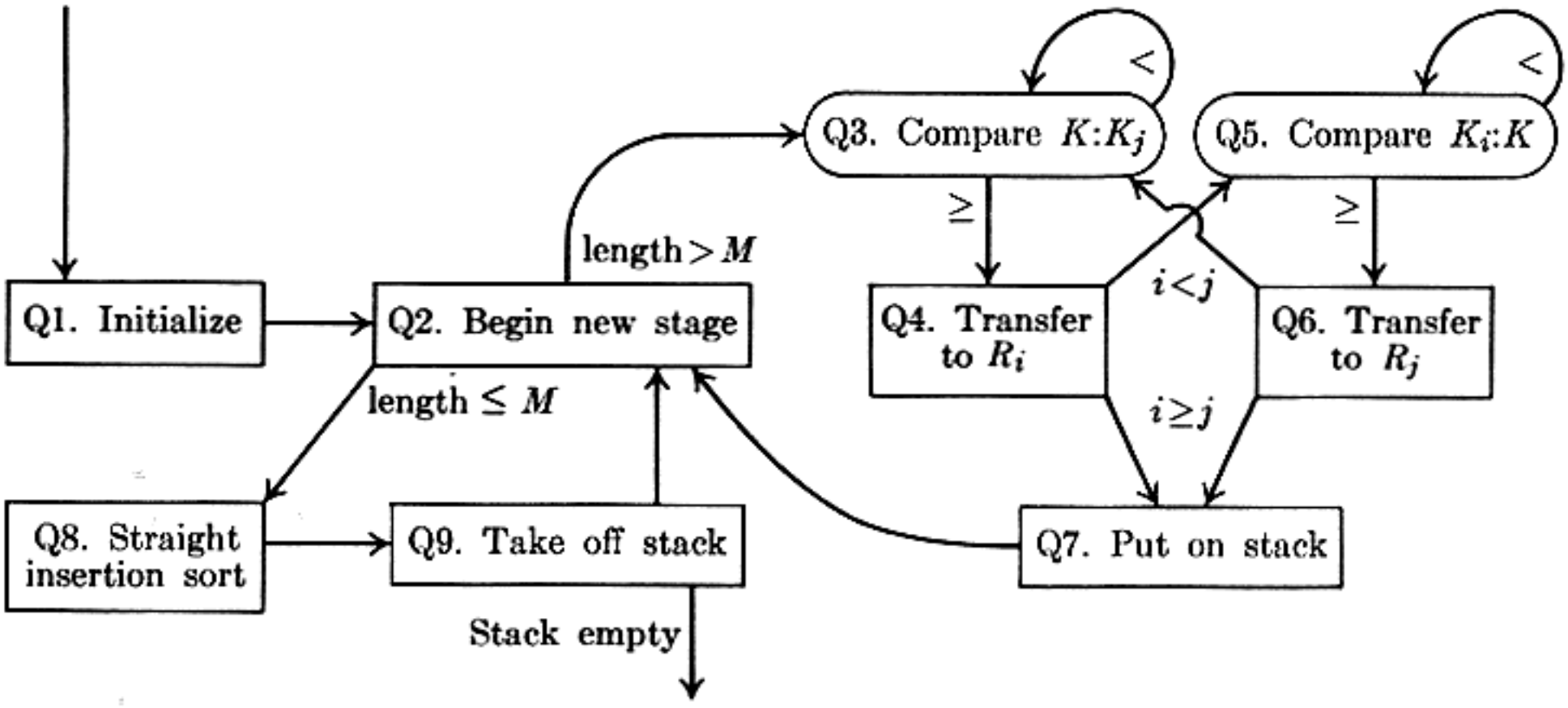


Fig. 19. Partition-exchange sorting ("quicksort").

(In order to indicate the position of i and j, K_i and K_j have been shown in boldface type.) Note that each of the comparisons in this example involves the key 503; in general, each comparison will involve the original value of K_1, because it keeps getting exchanged every time we switch directions. By the time that $i = j$, the original record R_1 will have moved into its final position, since it is easy to see that there will be no greater keys to its left and no smaller keys to its right. The original file will have been partitioned in such a way that the original problem is reduced to two simpler problems, sorting $R_1 \ldots R_{i-1}$ and (independently) sorting $R_{i+1} \ldots R_N$. We can apply the same technique to each of these subfiles.

Table 2 shows how our example file gets completely sorted by this approach, in 11 stages. Brackets indicate subfiles which still need to be sorted; inside a computer, these subfiles can be represented by two variables l and r (the

Table 2

"QUICKSORTING"

																(l, r)	Stack
[503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703]	(1, 16)	—
[154	087	426	061	275	170]	503	[897	653	908	512	509	612	677	765	703]	(1, 6)	(8, 16)
[061	087]	154	[426	275	170]	503	[897	653	908	512	509	612	677	765	703]	(1, 2)	(4, 6)(8, 16)
061	087	154	[426	275	170]	503	[897	653	908	512	509	612	677	765	703]	(4, 6)	(8, 16)
061	087	154	[170	275]	426	503	[897	653	908	512	509	612	677	765	703]	(4, 5)	(8, 16)
061	087	154	170	275	426	503	[897	653	908	512	509	612	677	765	703]	(8, 16)	—
061	087	154	170	275	426	503	[703	653	765	512	509	612	677]	897	908	(8, 14)	—
061	087	154	170	275	426	503	[677	653	612	512	509]	703	765	897	908	(8, 12)	—
061	087	154	170	275	426	503	[509	653	612	512]	677	703	765	897	908	(8, 11)	—
061	087	154	170	275	426	503	509	[653	612	512]	677	703	765	897	908	(9, 11)	—
061	087	154	170	275	426	503	509	[512	612]	653	677	703	765	897	908	(9, 10)	—
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908	—	—

boundaries of the subfile currently under examination) and a stack of additional pairs (l_k, r_k). Each time the file is subdivided, we put the *largest* subfile on the stack and commence work on the other one, until we reach trivially short files; this procedure assures that the stack will never contain more than about $\log_2 N$ entries, as shown in exercise 20.

The sorting procedure just described may be called *partition-exchange sorting;* it is due to C. A. R. Hoare, whose interesting paper [*Comp. J.* **5** (1962), 10–15] contains one of the most comprehensive accounts of a sorting method that has ever been published. Hoare dubbed his method "quicksort"; such a name is not inappropriate, since, for example, the entire sorting process in Table 2 requires only 48 comparisons (the least we have seen so far, except for binary insertion which takes 47). All comparisons in a given stage are made against the same key, so this key may be kept in a register. Furthermore, the amount of data movement is quite reasonable; the computation in Table 2 makes only 17 exchanges, most of which are simply "half-exchanges" (simple transfers) because one key stays in the register and it need not be stored until the end of a stage.

The extra bookkeeping (required to control i, j, and the stack) is not difficult, but it makes the quicksort partitioning procedure most suitable for large N; therefore it is desirable to sort short subfiles in a special manner as in the following algorithm.

Algorithm Q (*Partition-exchange sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete their keys will be in order, $K_1 \le \cdots \le K_N$. An auxiliary stack with at most $\log_2 N$ entries is needed for temporary storage. This algorithm follows the "quicksort" partitioning procedure described in the text above, with slight modifications for extra efficiency:

a) We assume the presence of artificial keys $K_0 = -\infty$ and $K_{N+1} = +\infty$ such that

$$K_0 \le K_i \le K_{N+1} \quad \text{for} \quad 1 \le i \le N. \tag{13}$$

(Equality is allowed.)

b) Subfiles of M or fewer elements are sorted by straight insertion, where $M \ge 1$ is a parameter which should be chosen as described in the text below.

c) One or two extra comparisons are made during particular stages (allowing the pointers i, j to cross), so that the main comparison loops can be as fast as possible.

d) Records with equal keys are exchanged, although it is not strictly necessary to do so. (This idea, due to R. C. Singleton, helps to split subfiles nearly in half when equal elements are present; see exercise 18.)

Q1. [Initialize.] Set the stack empty, and set $l \leftarrow 1$, $r \leftarrow N$.

Q2. [Begin new stage.] (We now wish to sort the subfile $R_l \ldots R_r$; from the nature of the algorithm, we have $r \geq l-1$, and $K_{l-1} \leq K_i \leq K_{r+1}$ for $l \leq i \leq r$.) If $r - l < M$, go to step Q8. Otherwise set $i \leftarrow l$, $j \leftarrow r$, $K \leftarrow K_l$, $R \leftarrow R_l$. (For a discussion of better choices for R and K, see the text below.)

Q3. [Compare K:K_j.] If $K < K_j$, decrease j by 1 and repeat this step.

Q4. [Transfer to R_i.] (At this point K_i is an irrelevant key $\geq K$, and $K \geq K_j$.) If $j \leq i$, set $R_i \leftarrow R$ and go to Q7. Otherwise set $R_i \leftarrow R_j$ and increase i by 1.

Q5. [Compare K_i:K.] If $K_i < K$, increase i by 1 and repeat this step.

Q6. [Transfer to R_j.] (At this point K_j is an irrelevant key $\leq K$, and $K \leq K_i$.] If $j \leq i$, set $R_j \leftarrow R$ and $i \leftarrow j$. Otherwise set $R_j \leftarrow R_i$, decrease j by 1, and go to Q3.

Q7. [Put on stack.] (Now the subfile $R_l \ldots R_i \ldots R_r$ has been partitioned so that $K_k \leq K_i$ for $l \leq k \leq i$ and $K_i \leq K_k$ for $i \leq k \leq r$.) If $r - i \geq i - l$, insert $(i+1, r)$ on top of the stack and set $r \leftarrow i-1$. Otherwise insert $(l, i-1)$ on top of the stack and set $l \leftarrow i+1$. (Each entry (a, b) on the stack is a request to sort the subfile $R_a \ldots R_b$ at some future time.) Now go back to step Q2.

Q8. [Straight insertion sort.] For $j = l+1, l+2, \ldots$ until $j > r$ do the following operations: Set $K \leftarrow K_j$, $R \leftarrow R_j$, $i \leftarrow j-1$; then set $R_{i+1} \leftarrow R_i$, $i \leftarrow i-1$ zero or more times until $K_i \leq K$; then set $R_{i+1} \leftarrow R$. (This is essentially Algorithm 5.2.1S, applied to a subfile of M or fewer elements.)

Q9. [Take off stack.] If the stack is empty, we are done sorting; otherwise remove its top entry (l', r'), set $l \leftarrow l'$, $r \leftarrow r'$, and return to step Q2. ∎

The corresponding MIX program is rather long, but not complicated; in fact, a large part of the coding is devoted to step Q7, which just fools around with the variables in a very straightforward way.

Program Q (*Partition-exchange sort*). Records to be sorted appear in locations INPUT+1 through INPUT+N; assume that locations INPUT and INPUT+N+1 contain, respectively, the smallest and largest values possible in MIX. The stack is kept in locations STACK+1, STACK+2, . . . ; see exercise 20 for the exact number of locations to set aside for the stack. rI1 $\equiv l$, rI2 $\equiv r$, rI3 $\equiv i$, rI4 $\equiv j$, rI6 $\equiv$ size of stack, rA $\equiv K \equiv R$.

```
      A       EQU     2:3                     First component of stack entry
      B       EQU     4:5                     Second component of stack entry
01    START   ENT1    1               1       Q1. Initialize. l ← 1.
02            ENT2    N               1       r ← N.
03            ENT6    0               1       Set stack empty.
04    2H      ENTX    0,2             2A + 1  Q2. Begin new stage.
05            DECX    M,1             2A + 1  rX ← r − l − M.
06            JXN     8F              2A + 1  To Q8 if subfile size ≤ M.
07            ENT3    0,1             A       i ← l.
08            ENT4    0,2             A       j ← r.
09            LDA     INPUT,3         A       K ← K_i.
10            JMP     3F              A       To Q3.
11    0H      LDX     INPUT,3         B
12            STX     INPUT,4         B       R_j ← R_i.
13            DEC4    1               C' − A  j ← j − 1.
14    3H      CMPA    INPUT,4         C'      Q3. Compare K:K_j.
15            JL      *-2             C'      If <, decrease j and repeat.
16    4H      ENTX    0,3             B + A   Q4. Transfer to R_i.
17            DECX    0,4             B + A
18            JXNN    7F              B + A   To Q7 if i ≥ j.
19            LDX     INPUT,4         B + X
20            STX     INPUT,3         B + X   R_i ← R_j.
21            INC3    1               C''     i ← i + 1.
22    5H      CMPA    INPUT,3         C''     Q5. Compare K_i:K.
23            JG      *-2             C''     If <, increase i and repeat.
24    6H      ENTX    0,3             B + X   Q6. Transfer to R_j.
25            DECX    0,4             B + X
26            JXN     0B              B + X   To Q3 if i < j.
27            ENT3    0,4             X       i ← j.
28    7H      STA     INPUT,3         A       Q7. Put on stack. R_i ← R.
29            ENTA    0,2             A
30            DECA    0,3             A
31            DECA    0,3             A
32            INCA    0,1             A
33            INC6    1               A       rI6 ← rI6 + 1.
34            JANN    1F              A       Jump if r − i ≥ i − l.
35            ST1     STACK,6(A)      A'
36            DEC3    1               A'
37            ST3     STACK,6(B)      A'      (l, i − 1) ⇒ stack.
38            ENT1    2,3             A'      l ← i + 1.
39            JMP     2B              A'      To Q2.
40    1H      INC3    1               A − A'
41            ST3     STACK,6(A)      A − A'
42            ST2     STACK,6(B)      A − A'  (i + 1, r) ⇒ stack.
43            ENT2    -2,3            A − A'  r ← i − 1.
44            JMP     2B              A − A'  To Q2.
45    8H      DEC2    0,1             A + 1   Q8. Straight insertion sort.
46            J2NP    9F              A + 1   To Q9 if r ≤ l.
47            ENT4    1,1             A + 1 − L  j ← l + 1.
```

48	1H	LDA	INPUT,4	D	$R \leftarrow R_j$.
49		ENT3	-1,4	D	$i \leftarrow j - 1$.
50		JMP	*+4	D	Jump into the loop.
51		LDX	INPUT,3	E	
52		STX	INPUT+1,3	E	$R_{i+1} \leftarrow R_i$.
53		DEC3	1	E	$i \leftarrow i - 1$.
54		CMPA	INPUT,3	$D+E$	
55		JL	*-4	$D+E$	Jump if $K < K_i$.
56		STA	INPUT+1,3	D	$R_{i+1} \leftarrow R$.
57		INC4	1	D	$j \leftarrow j + 1$.
58		DEC2	1	D	
59		J2P	1B	D	Repeat $r - l$ times.
60	9H	LD1	STACK,6(A)	$A+1$	*Q9. Take off stack.*
61		LD2	STACK,6(B)	$A+1$	$(l, r) \Leftarrow$ stack.
62		DEC6	1	$A+1$	rI6 $\leftarrow$ rI6 $-$ 1.
63		J6NN	2B	$A+1$	To Q2 if stack wasn't empty. ∎

Analysis of "quicksort." The timing information shown with Program Q is not hard to derive using Kirchhoff's conservation law (cf. Section 1.3.3) and the fact that everything put on the stack is eventually removed again. The total running time depends on the following quantities:

$$\begin{aligned}
A &= \text{number of stages in which the given subfile has more than } M \text{ elements;}\\
B &= \text{number of times } R_j \leftarrow R_i \text{ in step Q6;}\\
C' &= \text{number of times step Q3 is performed;}\\
C'' &= \text{number of times step Q5 is performed;}\\
D &= \text{number of times } R \leftarrow R_j \text{ in step Q8;}\\
E &= \text{number of times } R_{i+1} \leftarrow R_i \text{ in step Q8;}\\
L &= \text{number of times } r \le l \text{ in step Q8;}\\
X &= \text{number of times } i \leftarrow j \text{ in step Q6.}
\end{aligned} \tag{14}$$

Instead of analyzing C' and C'' independently, we shall consider only

$$C = C' + C'' = \text{ total number of comparisons,} \tag{15}$$

since the running time for C' and C'' is identical. By analyzing these seven quantities A, B, C, D, E, L, and X, we will be able to make an intelligent choice of the parameter M which specifies the "threshold" between straight insertion sorting and partitioning. (Nonmathematical readers, please skip to Eq. (25).)

As in most other analyses of this chapter, we shall assume that the keys to be sorted are distinct; exercise 18 indicates that equalities between keys do not seriously harm the efficiency of Algorithm Q, and in fact they seem to help it. Since the method depends only on the relative order of the keys, we may as well assume that they are simply $\{1, 2, \ldots, N\}$ in some order.

When we first get to step Q2, the sorting reduces to straight insertion if $N \le M$, and we have already analyzed straight insertion; so let us assume that $N > M$. We need to study only the computations which take us to step Q7

for the first time; once the partitioning has been achieved, it is easy to see that both subfiles $R_1 \ldots R_{i-1}$ and $R_{i+1} \ldots R_N$ are in random order if the original file was in random order, so the contribution of subsequent computations can be determined by induction on N.

Let s be the value of the first key, K_1, and assume that exactly t of the keys $K_1, \ldots, K_s$ are greater than s. Let

$$h = \begin{cases} 1, & \text{if } K_s < s; \\ 0, & \text{if } K_s \geq s. \end{cases} \tag{16}$$

In other words, h is 1 if the key which initially occupies position s (where K_1 will be placed) will be moved to the left. Remember that the keys being sorted are the integers $\{1, 2, \ldots, N\}$.

If $s = 1$, it is easy to see what happens during the first stage of partitioning: Step Q3 is performed N times, and then step Q4 takes us to Q7. So the contributions of the first stage in this case are $A = 1$, $B = 0$, $C = N$, $X = 0$. A similar but slightly more complicated argument when $s > 1$ (see exercise 21) shows that the contributions of the first stage to the total running time are, in general,

$$A = 1, B = t, C = N + 1 - \delta_{s1}, X = h, \qquad \text{for} \qquad 1 \leq s \leq N. \tag{17}$$

To this we must add the contributions of the later stages, which sort subfiles of $s - 1$ and $N - s$ elements, respectively.

If we assume that the original file is in random order, it is now possible to write down formulas which define the generating functions for the probability distributions of $A, B, \ldots, X$ (see exercise 22). But for simplicity we shall consider here only the *average* values of these quantities, $A_N, B_N, \ldots, X_N$, as functions of N. Consider, for example, the average number of comparisons, C_N, which occur during the partitioning process. When $N \leq M$, $C_N = 0$. Since any given value of s occurs with probability $1/N$, we have

$$\begin{aligned} C_N &= \frac{1}{N} \sum_{1 \leq s \leq N} (N + 1 - \delta_{s1} + C_{s-1} + C_{N-s}) \\ &= N + 1 - \frac{1}{N} + \frac{2}{N} \sum_{0 \leq k < N} C_k, \qquad \text{for} \qquad N > M. \end{aligned} \tag{18}$$

Similar formulas hold for the other quantities $A_N, B_N, \ldots, X_N$ (see exercise 23).

There is a simple way to solve recurrence relations of the form

$$x_n = f_n + \frac{2}{n} \sum_{0 \leq k < n} x_k, \qquad \text{for} \qquad n \geq m. \tag{19}$$

The first step is to get rid of the summation sign: Since

$$(n+1)x_{n+1} = (n+1)f_{n+1} + 2\sum_{0\le k\le n} x_k,$$

$$nx_n = nf_n + 2\sum_{0\le k<n} x_k,$$

we may subtract, obtaining

$$(n+1)x_{n+1} - nx_n = g_n + 2x_n, \qquad \text{where} \qquad g_n = (n+1)f_{n+1} - nf_n.$$

Now the recurrence takes the much simpler form

$$(n+1)x_{n+1} = (n+2)x_n + g_n, \qquad \text{for} \qquad n \ge m. \tag{20}$$

Any recurrence relation which has the general form

$$a_n x_{n+1} = b_n x_n + g_n \tag{21}$$

can be reduced to a summation if we multiply both sides by the "summing factor" $a_0\, a_1 \ldots a_{n-1}/b_0\, b_1 \ldots b_n$; we obtain

$$y_{n+1} = y_n + c_n, \quad \text{where} \quad y_n = \frac{a_0 \ldots a_{n-1}}{b_0 \ldots b_{n-1}}\, x_n, \quad c_n = \frac{a_0 \ldots a_{n-1}}{b_0\, b_1 \ldots b_n}\, g_n. \tag{22}$$

In our case (20), the summing factor is simply $n!/(n+2)! = 1/(n+1)(n+2)$, so we find that the simple relation

$$\frac{x_{n+1}}{n+2} = \frac{x_n}{n+1} + \frac{(n+1)f_{n+1} - nf_n}{(n+1)(n+2)}, \qquad n \ge m, \tag{23}$$

is a consequence of (19).

For example, if we set $f_n = 1/n$, we get the unexpected result $x_n/(n+1) = x_m/(m+1)$ for all $n \ge m$. If we set $f_n = n+1$, we get

$$\begin{aligned} x_n/(n+1) &= 2/(n+1) + 2/n + \cdots + 2/(m+2) + x_m/(m+1) \\ &= 2(H_{n+1} - H_{m+1}) + x_m/(m+1), \end{aligned}$$

for all $n \ge m$. Combining these two solutions, and setting $m = M+1$ and $x_n = 0$ for $n \le M$, gives the formula

$$\begin{aligned} C_N &= (N+1)\left(2H_{N+1} - 2H_{M+2} + 1 - \frac{1}{(M+1)(M+2)}\right) \\ &\approx 2(N+1)\ln\left(\frac{N+1}{M+2}\right), \qquad \text{for} \qquad N > M. \end{aligned} \tag{24}$$

In Section 6.2.2, we shall prove that the standard deviation of C_N is asymptotically $\sqrt{(21 - 2\pi^2)/3}\, N$; this is reasonably small compared to (24).

The other quantities can be found in a similar way (see exercise 23); we have

$$\begin{aligned}
A_N &= 2(N+1)/(M+2) - 1,\\
B_N &= \frac{1}{6}(N+1)\left(2H_{N+1} - 2H_{M+2} + 1 - \frac{6}{M+2}\right) + \frac{1}{2},\\
D_N &= (N+1)M(M-1)/(M+2)(M+1),\\
E_N &= \tfrac{1}{6}(N+1)M(M-1)/(M+2),\\
L_N &= 4(N+1)/(M+2)(M+1),\\
X_N &= (N+1)/(M+2) - \tfrac{1}{2}, \qquad \text{for} \qquad N > M.
\end{aligned} \tag{25}$$

The above discussion shows that it is possible to carry out an exact analysis of the average running time of a fairly complex program, by using techniques that we have previously applied only to simpler cases.

Formulas (24) and (25) can be used to determine the "best" value of M on a particular computer. In MIX's case, Program Q requires $37A + 14B + 4C + 12D + 8E - L + 8X + 15$ units of time; this comes to $\frac{1}{3}\,(38(N+1)H_N + (N+1)f(M)) - 19$ units on the average, for $N > M$, where

$$f(M) = 4M - 38H_{M+2} + 43 + \frac{84}{M+2} + \frac{48}{(M+2)(M+1)}. \tag{26}$$

We want to choose M so that $f(M)$ is a minimum. In this case

$$f(M) - f(M-1) = 4 - \frac{38}{M+2} - \frac{84}{(M+2)(M+1)} - \frac{96}{(M+2)(M+1)M},$$

and we want to find M such that $f(M) - f(M-1) \le 0$, $f(M+1) - f(M) \ge 0$; the solution, $M = 9$, is easily found. When $M = 9$ the average running time of Program Q is approximately $12.67(N+1)\ln N - 1.92N - 14.59$ for large N.

So Program Q is quite fast, on the average, considering that it requires very little memory space. But what is the *worst* case of Algorithm Q? Are there some inputs which it does not handle efficiently? The answer to this question is quite embarrassing: If the original file is already in order, with $K_1 < K_2 < \cdots < K_N$, each "partitioning" operation is almost useless, since it reduces the size of the subfile by only one element! So this situation (which ought to be easiest of all to sort) makes quicksort anything but quick; its running time becomes proportional to N^2 instead of $N \log N$. (See exercise 25.) Unlike the other sorting methods we have seen, Algorithm Q likes a disordered file!

Hoare suggested two ways to remedy the situation, in his original paper, by choosing a better value of the test key K which governs the partitioning. One of his recommendations is to choose a *random* integer q between l and r in the last part of step Q2; we can change the instructions "$K \leftarrow K_l$, $R \leftarrow R_l$" to

$$K \leftarrow K_q, \qquad R \leftarrow R_q, \qquad R_q \leftarrow R_l \tag{27}$$

in that step. According to Eqs. (25), such random integers need to be calculated only $2(N+1)/(M+2) - 1$ times on the average, so the additional running

time is not substantial; and the random choice gives good protection against the occurrence of the worst case.

Hoare's second suggestion is to look at a small sample of the file and to choose a median value of the sample. This approach has been followed by R. C. Singleton [*CACM* **12** (1969), 185–187], who suggests letting K_q be the median of the three values

$$K_l, \qquad K_{\lfloor (l+r)/2 \rfloor}, \qquad K_r. \tag{28}$$

Singleton's procedure cuts the number of comparisons down from $2N \ln N$ to about $\frac{12}{7}N \ln N$ (see exercise 29). It can be shown that B_N is asymptotically $C_N/5$ instead of $C_N/6$ in this case, so the median method slightly increases the amount of time spent in transferring the data; the total running time therefore decreases by roughly 8 percent. (See exercise 56 for a detailed analysis.) The worst case is still of order N^2, but such slow behavior will hardly ever occur.

W. D. Frazer and A. C. McKellar [*JACM* **17** (1970), 496–507] have suggested taking a much larger sample consisting of $2^k - 1$ records, where k is chosen so that $2^k \approx N/\ln N$. The sample can be sorted by the usual quicksort method, then inserted among the remaining records by taking k passes over the file (partitioning it into 2^k subfiles, bounded by the elements of the sample). Finally the subfiles are sorted. The average number of comparisons required by such a "samplesort" procedure is about the same as in Singleton's median method, when N is in a practical range, but it decreases to the asymptotic value $N \log_2 N$ as $N \to \infty$.

Radix exchange. We come now to a method which is quite different from any of the sorting schemes we have seen before; it makes use of the *binary representation* of the keys, so it is intended only for binary computers. Instead of comparing two keys with each other, this method inspects individual bits of the keys, to see if they are 0 or 1. In other respects it has the characteristics of exchange sorting, and, in fact, it is rather similar to quicksort. Since it depends on radix 2 representations, we call it "radix exchange sorting." The algorithm can be described roughly as follows:

1) Sort the sequence on its *most significant binary bit*, so that all keys which have a leading 0 come before all keys which have a leading 1. This sorting is done by finding the leftmost key K_i which has a leading 1, and the rightmost key K_j with a leading 0. Then R_i and R_j are exchanged and the process is repeated until $i > j$.

2) Let F_0 be the elements with leading bit 0, and let F_1 be the others. Apply the radix exchange sorting method to F_0 (starting now at the *second* bit from the left instead of the most significant bit), until F_0 is completely sorted; then do the same for F_1.

For example, Table 3 shows how the radix exchange sort acts on our 16 random numbers, which have been converted to octal notation. Stage 1 in the table shows the initial input, and after exchanging on the first bit we get to

Table 3

RADIX-EXCHANGE SORTING

Stage																	l	r	b	Stack
1	[1][*0767*	*0127*	*1000*	*0075*	*1614*	*0252*	*1601*	*0423*	*1215*	*0652*	*0232*	*0775*	*1144*	*1245*	*1375*	*1277*]	1	16	1	—
2	[2][*0767*	*0127*	*0775*	*0075*	*0232*	*0252*	*0652*	*0423*]	[2][*1215*	*1601*	*1614*	*1000*	*1144*	*1245*	*1375*	*1277*]	1	8	2	(16, 2)
3	[3][*0252*	*0127*	*0232*	*0075*]	[3][*0775*	*0767*	*0652*	*0423*]	[2][*1215*	*1601*	*1614*	*1000*	*1144*	*1245*	*1375*	*1277*]	1	4	3	(8, 3)(16, 2)
4	[4][*0075*	*0127*]	[4][*0232*	*0252*]	[3][*0775*	*0767*	*0652*	*0423*]	[2][*1215*	*1601*	*1614*	*1000*	*1144*	*1245*	*1375*	*1277*]	1	2	4	(4, 4)(8, 3)(16, 2)
5	*0075*	*0127*	[4][*0232*	*0252*]	[3][*0775*	*0767*	*0652*	*0423*]	[2][*1215*	*1601*	*1614*	*1000*	*1144*	*1245*	*1375*	*1277*]	3	4	4	(8, 3)(16, 2)
6	*0075*	*0127*	[5][*0232*	*0252*]	[3][*0775*	*0767*	*0652*	*0423*]	[2][*1215*	*1601*	*1614*	*1000*	*1144*	*1245*	*1375*	*1277*]	3	4	5	(8, 3)(16, 2)
7	*0075*	*0127*	*0232*	*0252*	[3][*0775*	*0767*	*0652*	*0423*]	[2][*1215*	*1601*	*1614*	*1000*	*1144*	*1245*	*1375*	*1277*]	5	8	3	(16, 2)
8	*0075*	*0127*	*0232*	*0252*	*0423*	[4][*0767*	*0652*	*0775*]	[2][*1215*	*1601*	*1614*	*1000*	*1144*	*1245*	*1375*	*1277*]	6	8	4	(16, 2)
9	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	[5][*0767*	*0775*]	[2][*1215*	*1601*	*1614*	*1000*	*1144*	*1245*	*1375*	*1277*]	7	8	5	(16, 2)
10	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	[6][*0767*	*0775*]	[2][*1215*	*1601*	*1614*	*1000*	*1144*	*1245*	*1375*	*1277*]	7	8	6	(16, 2)
11	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	[7][*0767*	*0775*]	[2][*1215*	*1601*	*1614*	*1000*	*1144*	*1245*	*1375*	*1277*]	7	8	7	(16, 2)
12	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	[2][*1215*	*1601*	*1614*	*1000*	*1144*	*1245*	*1375*	*1277*]	9	16	2	—
13	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	[3][*1215*	*1277*	*1375*	*1000*	*1144*	*1245*]	[3][*1614*	*1601*]	9	14	3	(16, 3)
14	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	[4][*1144*	*1000*]	[4][*1375*	*1277*	*1215*	*1245*]	[3][*1614*	*1601*]	9	10	4	(14, 4)(16, 3)
15	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	*1000*	*1144*	[4][*1375*	*1277*	*1215*	*1245*]	[3][*1614*	*1601*]	11	14	4	(16, 3)
16	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	*1000*	*1144*	[5][*1245*	*1277*	*1215*]	[5][*1375*]	[3][*1614*	*1601*]	11	13	5	(14, 5)(16, 3)
17	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	*1000*	*1144*	*1215*	[6][*1277*	*1245*]	[5][*1375*]	[3][*1614*	*1601*]	12	13	6	(14, 5)(16, 3)
18	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	*1000*	*1144*	*1215*	*1245*	*1277*	*1375*	[3][*1614*	*1601*]	15	16	3	—
19	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	*1000*	*1144*	*1215*	*1245*	*1277*	*1375*	[4][*1614*	*1601*]	15	16	4	—
20	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	*1000*	*1144*	*1215*	*1245*	*1277*	*1375*	[5][*1614*	*1601*]	15	16	5	—
21	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	*1000*	*1144*	*1215*	*1245*	*1277*	*1375*	[6][*1614*	*1601*]	15	16	6	—
22	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	*1000*	*1144*	*1215*	*1245*	*1277*	*1375*	[7][*1614*	*1601*]	15	16	7	—
23	*0075*	*0127*	*0232*	*0252*	*0423*	*0652*	*0767*	*0775*	*1000*	*1144*	*1215*	*1245*	*1277*	*1375*	*1614*	*1601*	17	—	—	—

stage 2. Stage 2 sorts the first group on bit 2, and stage 3 works on bit 3. (The reader should mentally convert the octal notation to 10-bit binary numbers.) When we reach stage 5, after sorting on bit 4, we find that each group remaining has but a single element, so this part of the file need not be further examined. The notation "$^4[0232\ 0252]$" means that the subfile *0232 0252* is waiting to be sorted on bit 4 from the left. In this particular case, no progress occurs when sorting on bit 4; we need to go to bit 5 before the items are separated.

The complete sorting process shown in Table 3 takes 22 stages, somewhat more than the comparable number for quicksort (Table 2). Similarly, the number of bit inspections, 82, is rather high; but we shall see that the number of bit inspections for large N is actually less than the number of comparisons made by quicksort, assuming a uniform distribution of keys. The total number of exchanges in Table 3 is 17, which is quite reasonable. Note that bit inspections never have to go past bit 7 here, although 10-bit numbers are being sorted.

As in quicksort, we can use a stack to keep track of the "boundary line information" for waiting subfiles. Instead of sorting the smallest subfile first, it is convenient simply to go from left to right, since the stack size in this case can never exceed the number of bits in the keys being sorted. In the following algorithm the stack entry (r, b) is used to indicate the right boundary r of a subfile waiting to be sorted on bit b; the left boundary need not be recorded in the stack, it is implicit because of the left-to-right nature of the procedure.

Algorithm R (*Radix-exchange sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \le \cdots \le K_N$. Each key is assumed to be an m-bit binary number, e.g. $(a_1\, a_2 \ldots a_m)_2$; the ith most significant bit, a_i, is called "bit i" of the key. An auxiliary stack with room for at most $m - 1$ entries is needed for temporary storage. This algorithm essentially follows the radix-exchange partitioning procedure described in the text above; certain improvements in its efficiency are possible, as described in the text and exercises below.

R1. [Initialize.] Set the stack empty, and set $l \leftarrow 1$, $r \leftarrow N$, $b \leftarrow 1$.

R2. [Begin new stage.] (We now wish to sort the subfile $R_l \le \cdots \le R_r$ on bit b; from the nature of the algorithm, we have $l \le r$.) If $l = r$, go to step R10 (since a one-word file is already sorted). Otherwise set $i \leftarrow l$, $j \leftarrow r$.

R3. [Inspect K_i for 1.] Examine bit b of K_i. If it is a 1, go to step R6.

R4. [Increase i.] Increase i by 1. If $i \le j$, return to step R3; otherwise go to step R8.

R5. [Inspect K_{j+1} for 0.] Examine bit b of K_{j+1}. If it is a 0, go to step R7.

R6. [Decrease j.] Decrease j by 1. If $i \le j$, go to step R5; otherwise go to step R8.

R7. [Exchange R_i, R_{j+1}.] Interchange records $R_i \leftrightarrow R_{j+1}$; then go to step R4.

R8. [Test special cases.] (At this point a partitioning stage has been completed; $i = j + 1$, bit b of keys $K_l, \ldots, K_j$ is 0, and bit b of keys $K_i, \ldots, K_r$ is 1.) Increase b by 1. If $b > m$, where m is the total number of bits in the keys, go to step R10. (In such a case, the subfile $R_l \ldots R_r$ has been sorted. This test need not be made if there is no chance of having equal keys present in the file.) Otherwise if $j < l$ or $j = r$, go back to step R2 (all bits examined were 1 or 0, respectively). Otherwise if $j = l$, increase l by 1 and go to step R2 (there was only one 0 bit).

R9. [Put on stack.] Insert the entry (r, b) on top of the stack; then set $r \leftarrow j$ and go to step R2.

R10. [Take off stack.] If the stack is empty, we are done sorting; otherwise set $l \leftarrow r + 1$, remove the top entry (r', b') of the stack, set $r \leftarrow r'$, $b \leftarrow b'$, and return to step R2. ∎

Program R (*Radix-exchange sort*). The following MIX code uses essentially the same conventions as Program Q. We have rI1 $\equiv l - r$, rI2 $\equiv r$, rI3 $\equiv i$, rI4 $\equiv j$, rI5 $\equiv m - b$, rI6 $\equiv$ size of stack, except that it proves convenient for certain instructions (designated below) to leave $rI3 = i - j$ or $rI4 = j - i$. Because of the binary nature of radix-exchange, this program uses the operations SRB (shift right AX binary), JAE (jump A even), and JAO (jump A odd) which are defined in Section 4.5.2. We assume that $N \geq 2$.

```
01    START  ENT6  0                1     R1. Initialize. Set stack empty.
02           ENT1  1-N              1     l ← 1.
03           ENT2  N                1     r ← N.
04           ENT5  M-1              1     b ← 1.
05           JMP   1F               1     To R2 (omit testing l = r).
06    9H     INC6  1                S     R9. Put on stack.          [rI4 = l − j]
07           ST2   STACK,6(A)       S
08           ST5   STACK,6(B)       S     (r, b) ⇒ stack.
09           ENN1  0,4              S     rI1 ← l − j.
10           ENT2  -1,3             S     r ← j.
11    1H     ENT3  0,1              A     R2. Begin new stage.       [rI3 = i − j]
12           ENT4  0,2              A     i ← l, j ← r.              [rI3 = i − j]
13    3H     INC3  0,4              C'    R3. Inspect K_i for 1.
14           LDA   INPUT,3          C'
15           SRB   0,5              C'    units bit of rA ← bit b of K_i.
16           JAE   4F               C'    To R4 if it is 0.
17    6H     DEC4  1,3              C''+X R6. Decrease j. j ← j − 1.  [rI4 = j − i]
18           J4N   8F               C''+X To R8 if j < i.            [rI4 = j − i]
19    5H     INC4  0,3              C''   R5. Inspect K_{j+1} for 0.
20           LDA   INPUT+1,4        C''
21           SRB   0,5              C''   units bit of rA ← bit b of K_{j+1}.
22           JAO   6B               C''   To R6 if it is 1.
23    7H     LDA   INPUT+1,4        B     R7. Exchange R_i, R_{j+1}.
24           LDX   INPUT,3          B
25           STX   INPUT+1,4        B
26           STA   INPUT,3          B
```

```
27    4H    DEC3  -1,4          C' - X      R4. Increase i. i ← i + 1.  [rI3 = i - j]
28          J3NP  3B            C' - X      To R3 if i ≤ j.             [rI3 = i - j]
29          INC3  0,4           A - X       rI3 ← i.
30    8H    J5Z   0F              A         R8. Test special cases.  [rI4 = unknown]
31          DEC5  1             A - G       To R10 if b = m, else b ← b - 1.
32          ENT4  -1,3          A - G       rI4 ← j.
33          DEC4  0,2           A - G       rI4 ← j - r.
34          J4Z   1B            A - G       To R2 if j = r.
35          DEC4  0,1         A - G - R     rI4 ← j - l.
36          J4N   1B          A - G - R     To R2 if j < l.
37          J4NZ  9B        A - G - L - R   To R9 if j ≠ l.
38          INC1  1             K           l → l + 1.
39    2H    J1NZ  1B            K + S       Jump if l ≠ r.
40    0H    ENT1  1,2           S + 1       R10. Take off stack.
41          LD2   STACK,6(A)    S + 1
42          DEC1  0,2           S + 1
43          LD5   STACK,6(B)    S + 1       stack ⇒ (r, b).
44          DEC6  1             S + 1
45          J6NN  2B            S + 1       To R2 if stack was nonempty. ∎
```

The running time of this radix exchange program depends on

$$\begin{aligned}
A &= \text{number of stages encountered with } l < r;\\
B &= \text{number of exchanges;}\\
C &= C' + C'' = \text{number of bit inspections;}\\
G &= \text{number of times } b > m \text{ in step R8;}\\
K &= \text{number of times } b \le m,\ j = l \text{ in step R8;}\\
L &= \text{number of times } b \le m,\ j < l \text{ in step R8;}\\
R &= \text{number of times } b \le m,\ j = r \text{ in step R8;}\\
S &= \text{number of times things are entered onto the stack;}\\
X &= \text{number of times } j < i \text{ in step R6.}
\end{aligned} \tag{29}$$

By Kirchhoff's law, $S = A - G - K - L - R$; so the total running time comes to $27A + 8B + 8C - 23G - 14K - 17L - 19R - X + 13$ units. The bit-inspection loops can be made somewhat faster, as shown in exercise 34, at the expense of a more complicated program. It is also possible to increase the speed of radix exchange by using straight insertion whenever $r - l$ is sufficiently small, as we did in Algorithm Q; but we shall not dwell on these refinements.

In order to analyze the running time of radix exchange, two kinds of input data suggest themselves:

i) Assume that $N = 2^m$ and that the keys to be sorted are simply the integers $0, 1, 2, \ldots, 2^m - 1$ in random order; or

ii) Assume that $m = \infty$ (unlimited precision) and that the keys to be sorted are independent uniformly distributed real numbers in $[0, 1)$.

The analysis of case (i) is relatively easy, so it has been left as an exercise for the reader (see exercise 35). Case (ii) is comparatively difficult, so it has *also* been left as an exercise. The following table shows crude approximations to the results of these analyses:

Quantity	Case (i)	Case (ii)
A	N	αN
B	$\frac{1}{4}N \log_2 N$	$\frac{1}{4}N \log_2 N$
C	$N \log_2 N$	$N \log_2 N$
G	$\frac{1}{2}N$	0
K	0	$\frac{1}{2}N$
L	0	$\frac{1}{2}(\alpha - 1)N$
R	0	$\frac{1}{2}(\alpha - 1)N$
S	$\frac{1}{2}N$	$\frac{1}{2}N$
X	$\frac{1}{2}N$	$\frac{1}{4}(\alpha + 1)N$

(30)

Here $\alpha = 1/(\ln 2) = 1.4427$. Note that the average number of exchanges, bit inspections, and stack accesses is essentially the same for both cases, even though case (ii) takes about 44 percent more stages. Our MIX program takes approximately $14.4\, N \ln N$ units of time, on the average, to sort N items in case (ii), and this could be cut to about $11.5\, N \ln N$ using the suggestion of exercise 34; the corresponding figure for Program Q is $12.7\, N \ln N$, which can be decreased to about $11.7\, N \ln N$ using Singleton's median-of-three suggestion.

Thus radix exchange sorting takes about as long as quicksort, on the average, when sorting uniformly distributed data; on MIX it is actually a little quicker than quicksort. Exercise 53 indicates to what extent the process slows down for a nonuniform distribution. It is important to note that our entire analysis is predicated on the assumption that keys are distinct; *radix exchange as defined above is not especially efficient when equal keys are present*, since it goes through several time-consuming stages trying to separate sets of identical keys before b becomes $> m$. One plausible way to remedy this defect is suggested in the answer to exercise 40.

Both radix exchange and quicksort are essentially based on the idea of partitioning. Records are exchanged until the file is split into two parts: a left-hand subfile, in which all keys are $\leq K$, for some K, and a right-hand subfile in which all keys are $\geq K$. Quicksort chooses K to be an actual key in the file, while radix exchange essentially chooses an artificial key K based on binary representations. From a historical standpoint, radix exchange was discovered by P. Hildebrandt, H. Isbitz, H. Rising, and J. Schwartz [*JACM* **6** (1959), 156–163], about a year earlier than quicksort. Other partitioning schemes are also possible; for example, John McCarthy has suggested setting $K \approx \frac{1}{2}(u + v)$, if all keys are known to lie between u and v.

Still another partitioning strategy has been proposed by M. H. van Emden [*CACM* **13** (1970), 563–567]: Instead of choosing K in advance, we "learn"

what a good K might be, by keeping track of $K' = \max(K_l, \ldots, K_i)$ and $K'' = \min(K_j, \ldots, K_r)$ as partitioning proceeds. We may increase i until encountering a key greater than K', then decrease j until encountering a key less than K'', then exchange and/or adjust K' and K''. Empirical tests on this "interval-exchange sort" method indicate that it requires about $1.64\,N \ln N = 1.14\,N \log_2 N$ comparisons; it is the only sorting method discussed in this book for which no adequate theoretical explanation of the behavior has yet been found.

A generalization of radix exchange to radices higher than 2 is discussed in Section 5.2.5.

***Asymptotic methods.** The analysis of exchange sorting algorithms leads to some particularly instructive mathematical problems which enable us to learn more about how to find the asymptotic behavior of functions. For example, we came across the function

$$W_n = \frac{1}{n!} \sum_{0 \le r < s \le n} s! r^{n-s} \tag{31}$$

in our analysis of the bubble sort (Eq. 9); what is its asymptotic value?

We can proceed as in our study of the number of involutions, Eq. 5.1.4–41; the reader will find it helpful to review the discussion at the end of Section 5.1.4 before reading further.

Inspection of (31) shows that the contribution for $s = n$ is larger than that for $s = n - 1$, etc; this suggests replacing s by $n - s$. In fact, we soon discover that it is most convenient to use the substitutions $t = n - s + 1$, $m = n + 1$, so that (31) becomes

$$\frac{1}{m} W_{m-1} = \frac{1}{m!} \sum_{1 \le t < m} (m-t)! \sum_{0 \le r < m-t} r^{t-1}. \tag{32}$$

The inner sum has a well-known asymptotic series obtained from Euler's summation formula,

$$\begin{aligned} \sum_{0 \le r < N} r^{t-1} &= \frac{N^t}{t} - \frac{1}{2}(N^{t-1} - \delta_{t1}) + \frac{B_2}{2!}(t-1)(N^{t-2} - \delta_{t2}) + \cdots \\ &= \frac{1}{t} \sum_{0 \le j \le k} \binom{t}{j} B_j (N^{t-j} - \delta_{tj}) + O(N^{t-k}) \end{aligned} \tag{33}$$

(see exercise 1.2.11.2–4), hence our problem reduces to studying sums of the form

$$\frac{1}{m!} \sum_{1 \le t < m} (m-t)!(m-t)^t t^k, \qquad k \ge -1. \tag{34}$$

As in Section 5.1.4 we can show that the value of this summand is negligible, $O(\exp(-n^\delta))$, whenever t is greater than $m^{1/2+\epsilon}$; hence we may put $t =$

$O(m^{1/2+\epsilon})$ and replace the factorials by Stirling's approximation:

$$\frac{(m-t)!(m-t)^t}{m!}$$
$$= \sqrt{1-\frac{t}{m}}\exp\left(\frac{t}{12m^2}-\left(\frac{t^2}{2m}+\frac{t^3}{3m^2}+\frac{t^4}{4m^3}+\frac{t^5}{5m^4}\right)+O(m^{-2+6\epsilon})\right).$$

We are therefore interested in the asymptotic value of

$$r_k(m)=\sum_{1\le t<m} e^{-t^2/2m}t^k, \qquad k\ge -1. \tag{35}$$

(The sum could also be extended to the full range $1\le t<\infty$ without changing its asymptotic value, since the values for $t>m^{1/2+\epsilon}$ are negligible as stated above.)

Let $g_k(x)=x^ke^{-x^2}$ and $f_k(x)=g_k(x/\sqrt{2m})$. When $k\ge 0$, Euler's summation formula tells us that

$$\sum_{0\le t<m} f_k(t)=\int_0^m f_k(x)\,dx+\sum_{1\le j\le p}\frac{B_j}{j!}\left(f_k^{(j-1)}(m)-f_k^{(j-1)}(0)\right)+R_p,$$

$$R_p=\frac{(-1)^{p+1}}{p!}\int_0^m B_p(\{x\})f_k^{(p)}(x)\,dx$$
$$=\left(\frac{1}{\sqrt{2m}}\right)^p O\left(\int_0^\infty |g_k^{(p)}(y)|\,dy\right)=O(m^{-p/2}); \tag{36}$$

hence we can get an asymptotic series for $r_k(m)$ whenever $k\ge 0$ by using essentially the same ideas we have used before. But when $k=-1$ the method breaks down, since $f_{-1}(0)$ is undefined; we can't merely sum from 1 to m either, because the remainders don't give smaller and smaller powers of m when the lower limit is 1. (This is the crux of the matter, and the reader should make sure he appreciates the problem before reading further.)

To resolve the dilemma we can define $g_{-1}(x)=(e^{-x^2}-1)/x$, $f_{-1}(x)=g_{-1}(x/\sqrt{2m})$; then $f_{-1}(0)=0$ and $r_{-1}(m)$ can be obtained from $\sum_{0\le t<m}f_{-1}(t)$ in a simple way. Equation (36) is now valid for $k=-1$, and the remaining integral is "well known,"

$$\frac{2}{\sqrt{2m}}\int_0^m f_{-1}(x)\,dx=2\int_0^m\frac{e^{-x^2/2m}-1}{x}\,dx=\int_0^{m/2}\frac{e^{-y}-1}{y}\,dy$$
$$=\int_0^1\frac{e^{-y}-1}{y}\,dy+\int_1^{m/2}\frac{e^{-y}}{y}\,dy-\ln(m/2)$$
$$=-\gamma-\ln(m/2)+O(e^{-m/2})$$

by exercise 43.

Now we have enough facts and formulas to grind out the answer,

$$W_n = \tfrac{1}{2}m \ln m + \tfrac{1}{2}(\gamma + \ln 2)m - \tfrac{2}{3}\sqrt{2\pi m} + \tfrac{31}{36} + O(n^{-1/2}), \qquad m = n + 1, \tag{37}$$

as shown in exercise 44. This completes our analysis of the bubble sort.

For the analysis of radix exchange sorting, we need to know the asymptotic value of the finite sum

$$U_n = \sum_{k\ge 2} \binom{n}{k} (-1)^k \frac{1}{2^{k-1} - 1} \tag{38}$$

as $n \to \infty$. This question turns out to be harder than any of the other asymptotic problems we have met so far; the elementary methods of power series expansions, Euler's summation formula, etc., turn out to be inadequate. The following derivation has been suggested by N. G. de Bruijn.

To get rid of the cancellation effects of the large factors $\binom{n}{k}(-1)^k$ in (38), we start by rewriting the sum as an infinite series

$$U_n = \sum_{k\ge 2} \binom{n}{k} (-1)^k \sum_{j\ge 1} \left(\frac{1}{2^{k-1}}\right)^j = \sum_{j\ge 1} (2^j(1 - 2^{-j})^n - 2^j + n). \tag{39}$$

If we set $x = n/2^j$, the summand is

$$2^j(1 - 2^{-j})^n - 2^j + n = \frac{n}{x}\left(\left(1 - \frac{x}{n}\right)^n - 1 + x\right).$$

When $x \le n^\epsilon$, we have

$$\left(1 - \frac{x}{n}\right)^n = \exp\left(n \ln\left(1 - \frac{x}{n}\right)\right) = \exp\left(-x + x^2 O(n^{-1})\right), \tag{40}$$

and this suggests approximating (39) by

$$T_n = \sum_{j\ge 1} (2^j e^{-n/2^j} - 2^j + n). \tag{41}$$

To justify this approximation, we have $U_n - T_n = X_n + Y_n$, where

$$X_n = \sum_{\substack{j\ge 1 \\ 2^j < n^{1-\epsilon}}} (2^j(1 - 2^{-j})^n - 2^j e^{-n/2^j}) \qquad \text{[i.e., the terms for } x > n^\epsilon\text{]}$$

$$= \sum_{\substack{j\ge 1 \\ 2^j < n^{1-\epsilon}}} O(ne^{-n/2^j}) \qquad \text{[since } 0 < 1 - 2^{-j} < e^{-2^{-j}}\text{]}$$

$$= O(n \log n\, e^{-n^\epsilon}) \qquad \text{[since there are } O(\log n) \text{ terms]};$$

and

$$Y_n = \sum_{\substack{j\ge 1\\ 2^j\ge n^{1-\epsilon}}} (2^j(1-2^{-j})^n - 2^j e^{-n/2^j}) \qquad \text{[the terms for } x \le n^{-\epsilon}\text{]}$$

$$= \sum_{\substack{j\ge 1\\ 2^j\ge n^{1-\epsilon}}} \left(e^{-n/2^j}\left(\frac{n}{2^j}\right)O(1)\right) \qquad \text{[by (40)].}$$

Our discussion below will demonstrate that the latter sum is $O(1)$; hence $U_n - T_n = O(1)$. (See exercise 47.)

So far we haven't applied any techniques that are really different from those we have used before; but the study of T_n requires a new idea, based on simple principles of complex variable theory. If x is any positive number, we have

$$e^{-x} = \frac{1}{2\pi i}\int_{1/2-i\infty}^{1/2+i\infty} \Gamma(z)x^{-z}\,dz = \frac{1}{2\pi}\int_{-\infty}^{\infty} \Gamma(\tfrac{1}{2}+it)x^{-(1/2+it)}\,dt. \qquad (42)$$

To prove this identity, consider the path of integration shown in Fig 20(a), where N, N', and M are large. The value of the integral along this contour is the sum of the residues inside, namely,

$$\sum_{0\le k<M} x^{-k} \lim_{z\to -k}(z+k)\Gamma(z) = \sum_{0\le k<M} x^{-k}\frac{(-1)^k}{k!}.$$

The integral on the top line is $O(\int_{-\infty}^{1/2} |\Gamma(t+iN)|x^{-t}\,dt)$, and we have the well-known bound

$$\Gamma(t+iN) = O(N^{t-1/2}e^{-\pi N/2}) \qquad \text{as} \qquad N\to\infty.$$

[For properties of the gamma function see, for example, A. Erdélyi et al., *Higher Transcendental Functions* **1** (New York: McGraw-Hill, 1953), Chapter 1.] Therefore the top line integral is quite negligible, $O(e^{-\pi N/2}\int_{-\infty}^{1/2}(N/x)^t\,dt)$. The bottom line integral has a similar innocuous behavior. For the integral along the left line we use the fact that

$$\Gamma(\tfrac{1}{2}+it-M) = \Gamma(\tfrac{1}{2}+it)/(-M+\tfrac{1}{2}+it)\ldots(-1+\tfrac{1}{2}+it)$$
$$= \Gamma(\tfrac{1}{2}+it)O(1/(M-1)!);$$

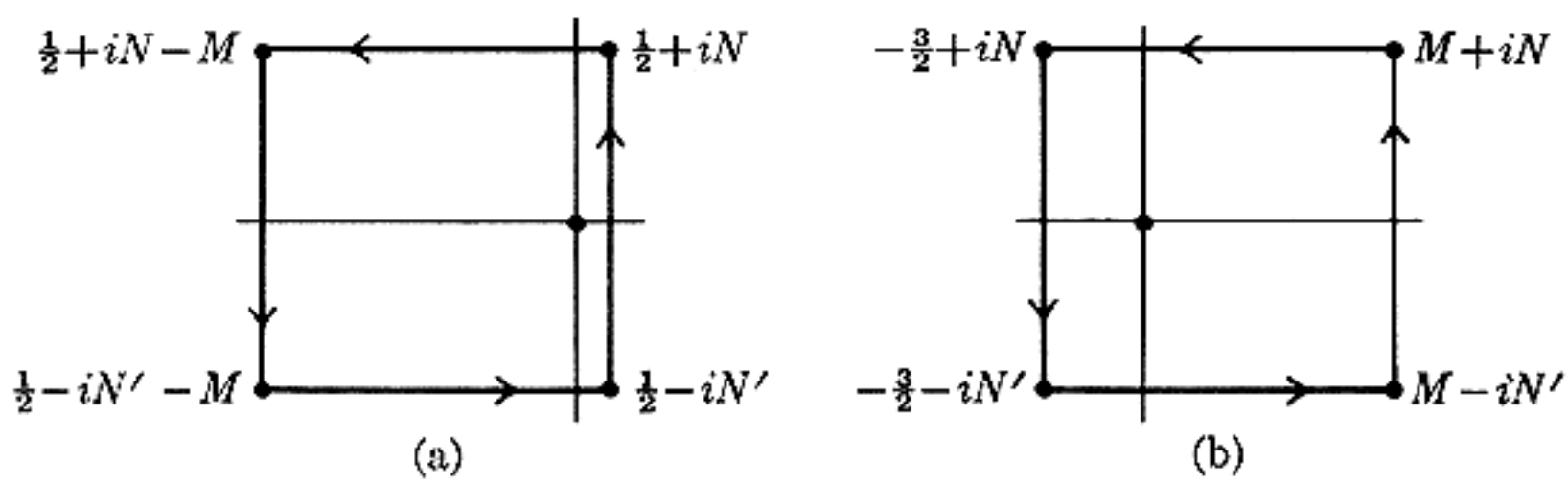


Fig. 20. Contours of integration for gamma-function identities.

hence the left-hand integral is $O(x^{M-1/2}/(M-1)!)\int_{-\infty}^{\infty}|\Gamma(\frac{1}{2}+it)|\,dt$. Therefore as $M, N, N' \to \infty$, only the right-hand integral survives, and this proves (42). In fact, (42) remains valid if we replace "$\frac{1}{2}$" by any positive number.

The same argument can be used to derive many other useful relations involving the gamma function. We can replace x^{-z} by other functions of z; or we can replace the constant $\frac{1}{2}$ by other quantities. For example,

$$\frac{1}{2\pi i}\int_{-3/2-i\infty}^{-3/2+i\infty}\Gamma(z)x^{-z}\,dz = e^{-x} - 1 + x, \tag{43}$$

and this is the critical quantity in our formula (41) for T_n:

$$T_n = n\sum_{j\geq 1}\frac{1}{2\pi i}\int_{-3/2-i\infty}^{-3/2+i\infty}\Gamma(z)(n/2^j)^{-1-z}\,dz. \tag{44}$$

The sum may be placed inside the integrals, since convergence is absolutely well-behaved; we have

$$\sum_{j\leq 1}(n/2^j)^w = n^w\sum_{j\leq 1}(1/2^w)^j = n^w/(2^w-1), \qquad \text{when} \qquad \Re(w) > 0,$$

since $|2^w| = 2^{\Re(w)} > 1$. Therefore

$$T_n = \frac{n}{2\pi i}\int_{-3/2-i\infty}^{-3/2+i\infty}\frac{\Gamma(z)n^{-1-z}}{2^{-1-z}-1}\,dz, \tag{45}$$

and it remains to evaluate the latter integral.

This time we integrate along a path which extends far to the *right*, as in Fig. 20(b). The top line integral is $O(n^{1/2}e^{-\pi M/2}\int_{-3/2}^{M}N^t\,dt)$, if $2^{iN} \neq 1$, and the bottom line integral is equally negligible. The right-hand line integral is $O(n^{-1-M}\int_{-\infty}^{\infty}|\Gamma(M+it)|\,dt)$. Fixing M and letting $N, N' \to \infty$ shows that $-T_n/n$ is $O(n^{-1-M})$ plus the sum of the residues in the region $-3/2 < \mathrm{Re}\,(z) < M$. Let $M \to \infty$; $\Gamma(z)$ has simple poles at $z = -1$ and 0, n^{-1-z} has no poles, and $1/(2^{-1-z}-1)$ has simple poles when $z = -1 + 2\pi ik/\ln 2$.

The double pole at $z = -1$ is the hardest to handle. We can use the well-known relation

$$\Gamma(z+1) = \exp\left(-\gamma z + \zeta(2)z^2/2 - \zeta(3)z^3/3 + \zeta(4)z^4/4 - \cdots\right),$$

where $\zeta(s) = 1^{-s} + 2^{-s} + 3^{-s} + \cdots = H_\infty^{(s)}$,

to deduce the following expansions when $w = z + 1$ is small:

$$\Gamma(z) = \frac{\Gamma(w+1)}{w(w-1)} = -w^{-1} + (\gamma - 1) + O(w),$$

$$n^{-1-z} = 1 - (\ln n)w + O(w^2),$$

$$1/(2^{-1-z}-1) = -w^{-1}/\ln 2 - \tfrac{1}{2} + O(w).$$

The residue at $z = -1$ is the coefficient of w^{-1} in the product of these three formulas, namely $\frac{1}{2} - (\ln n + \gamma - 1)/\ln 2$. Adding the other residues gives the formula

$$T_n/n = \frac{\ln n + \gamma - 1}{\ln 2} - \frac{1}{2} + f(n) + \frac{2}{n}, \tag{46}$$

where $f(n)$ is a rather strange function,

$$f(n) = \frac{2}{\ln 2} \sum_{k \geq 1} \Re\left(\Gamma(-1 - 2\pi ik/\ln 2) \exp\left(2\pi ik \log_2 n\right)\right). \tag{47}$$

Note that $f(n) = f(2n)$. The average value of $f(n)$ is zero, since the average value of each term is zero. (We may assume that $(\log_2 n) \bmod 1$ is uniformly distributed, in view of the results about floating-point numbers in Section 4.2.4.) Furthermore since $|\Gamma(-1 + it)| = |\pi/(t(1 + t^2) \sinh \pi t)|^{1/2}$, it is not difficult to show that

$$f(n) < 0.0000001725;$$

thus we may safely ignore $f(n)$ for practical purposes. For theoretical purposes, however, we can't obtain an asymptotic expansion of U_n without it; that is why U_n is a comparatively difficult function to analyze.

In summary, we have proved that

$$U_n = n \log_2 n + n\left(\frac{\gamma - 1}{\ln 2} - \frac{1}{2} + f(n)\right) + O(1). \tag{48}$$

Other examples of this gamma-function method appear in exercises 51–53 and in Section 6.3.

EXERCISES

1. [*M20*] Let $a_1 \ldots a_n$ be a permutation of $\{1, \ldots, n\}$, and let i and j be indices such that $i < j$ and $a_i > a_j$. Let $a'_1 \ldots a'_n$ be the permutation obtained from $a_1 \ldots a_n$ by interchanging a_i and a_j. Can $a'_1 \ldots a'_n$ have more inversions than $a_1 \ldots a_n$?

▶ **2.** [*M25*] (a) What is the minimum number of exchanges which will sort the permutation 3 7 6 9 8 1 4 5 2? (b) In general, given any permutation $\pi = a_1 \ldots a_n$ of $\{1, \ldots, n\}$, let xch (π) be the minimum number of exchanges which will sort π into increasing order. Express xch (π) in terms of "simpler" characteristics of π. (Cf. exercise 5.2.1–39.)

3. [*10*] Is the bubble sort Algorithm B a "stable" sorting algorithm?

4. [*M23*] If $t = 1$ in step B4, we could actually terminate Algorithm B immediately, because the subsequent step B2 will do nothing useful. What is the probability that $t = 1$ will occur in step B4 when sorting a random permutation?

5. [*M25*] Let $b_1\, b_2 \ldots b_n$ be the inversion table for the permutation $a_1\, a_2 \ldots a_n$. Show that the value of BOUND after r passes of the bubble sort is $\max \{b_i + i \mid b_i \geq r\} - r$, for $0 \leq r \leq \max (b_1, \ldots, b_n)$.

6. [*M22*] Let $a_1 \ldots a_n$ be a permutation of $\{1, \ldots, n\}$ and let $a'_1 \ldots a'_n$ be its inverse. Show that the number of passes to bubble-sort $a_1 \ldots a_n$ is $1 + \max(a'_1 - 1, a'_2 - 2, \ldots, a'_n - n)$.

7. [*M28*] Calculate the standard deviation of the number of passes for the bubble sort, and express it in terms of n and the function $P(n)$. [Cf. Eqs. (6) and (7).]

8. [*M24*] Derive Eq. (8).

9. [*M48*] Analyze the number of passes and the number of comparisons in the cocktail-shaker shorting [shic] algorithm. *Note:* See exercise 5.4.8–9 for partial information.

10. [*M26*] Let $a_1\, a_2 \ldots a_n$ be a 2-ordered permutation of $\{1, 2, \ldots, n\}$. (a) What are the coordinates of the endpoints of the a_i-th step of the corresponding lattice path? (Cf. Fig. 11.) (b) Prove that comparison/exchange of $a_1{:}a_2$, $a_3{:}a_4$, ..., corresponds to folding the path about the diagonal, as in Fig. 18(b). (c) Prove that comparison/exchange of $a_2{:}a_{2+d}$, $a_4{:}a_{4+d}$, ..., corresponds to folding the path about a line m units below the diagonal, as in Figs. 18(c), (d), and (e), when $d = 2m - 1$.

▸ **11.** [*M25*] What permutation of $\{1, 2, \ldots, 16\}$ maximizes the number of exchanges done by Batcher's algorithm?

12. [*24*] Write a MIX program for Algorithm M, assuming that MIX is a binary computer with the operations AND, SRB. How much time does your program take to sort the sixteen records in Table 1?

13. [*10*] Is Batcher's method a "stable" sorting algorithm?

14. [*M21*] Let $c(N)$ be the number of key comparisons used to sort N elements by Batcher's method; this is the number of times step M4 is performed. (a) Show that $c(2^t) = 2c(2^{t-1}) + (t-1)2^{t-1} + 1$, for $t \geq 1$. (b) Find a simple expression for $c(2^t)$ as a function of t. *Hint:* Consider the sequence $x_t = c(2^t)/2^t$.

15. [*M38*] The object of this exercise is to analyze the function $c(N)$ of exercise 13, and to find a formula for $c(N)$ when $N = 2^{e_1} + 2^{e_2} + \cdots + 2^{e_r}$, $e_1 > e_2 > \cdots > e_r \geq 0$. (a) Let $a(N) = c(N+1) - c(N)$. Prove that $a(2n) = a(n) + \lfloor \log_2 (2n) \rfloor$, and $a(2n+1) = a(n) + 1$; hence

$$a(N) = \binom{e_1 + 1}{2} - r(e_1 - 1) + (e_1 + e_2 + \cdots + e_r).$$

(b) Let $x(n) = a(n) - a(\lfloor n/2 \rfloor)$, so that $a(n) = x(n) + x(\lfloor n/2 \rfloor) + x(\lfloor n/4 \rfloor) + \cdots$. Let $y(n) = x(1) + x(2) + \cdots + x(n)$; and let $z(2n) = y(2n) - a(n)$, $z(2n+1) = y(2n+1)$. Prove that $c(N+1) = z(N) + 2z(\lfloor N/2 \rfloor) + 4z(\lfloor N/4 \rfloor) + \cdots$. (c) Prove that $y(N) = N + (\lfloor N/2 \rfloor + 1)(e_1 - 1) - 2^{e_1} + 2$. (d) Now put everything together and find a formula for $c(N)$ in terms of the exponents e_j, holding r fixed.

16. [*HM46*] Find the asymptotic value of the *average* number of exchanges occurring when Batcher's method is applied to $N = 2^t$ distinct elements, randomly ordered.

▸ **17.** [*20*] Where in Algorithm Q do we use the fact that K_0 and K_{N+1} have the values postulated in (13)?

▸ **18.** [*20*] Explain how the computation proceeds in Algorithm Q when all of the input keys are equal. What would happen if the "$<$" signs in steps Q3 and Q5 were changed to "$\leq$" instead?

19. [*15*] Would Algorithm Q still work properly if a queue (first-in-first-out) were used instead of a stack (last-in-first-out)?

20. [*M20*] What is the largest possible number of elements that will ever be on the stack at once in Algorithm Q, as a function of M and N?

21. [*20*] Explain why the partitioning phase of Algorithm Q takes the number of comparisons, transfers, etc., specified in (17).

22. [*M25*] Let p_{kN} be the probability that the quantity A in (14) will equal k, when Algorithm Q is applied to a random permutation of $\{1, 2, \ldots, N\}$, and let $A_N(z) = \sum_k p_{kN} z^k$ be the corresponding generating function. Prove that $A_N(z) = 1$ for $N \leq M$, and $A_N(z) = z\bigl(\sum_{1\leq s\leq N} A_{s-1}(z) A_{N-s}(z)\bigr)/N$ for $N > M$. Find similar recurrence relations defining the other probability distributions $B_N(z)$, $C_N(z)$, $D_N(z)$, $E_N(z)$, $L_N(z)$, $X_N(z)$.

23. [*M24*] Let A_N, B_N, D_N, E_N, L_N, X_N be the average values of the corresponding quantities in (14), when sorting a random permutation of $\{1, 2, \ldots, N\}$. Find recurrence relations for these quantities, analogous to (18); and solve these recurrences to obtain (25).

24. [*M21*] Algorithm Q obviously does a few more comparisons than it needs to, since we can have $i = j$ and even $i > j$ in steps Q3 and Q5. How many comparisons C_N would be done on the average if we avoided all comparisons when $i \geq j$?

25. [*M20*] When the input keys are the numbers $1\ 2 \ldots N$ in order, what are the exact values of the quantities A, B, C, D, E, L, and X in the timing of Program Q? (Assume that $N > M$.)

▸ **26.** [*M21*] Construct an input file which makes Program Q go even more slowly than it does in exercise 25. (Try to find a really bad case.)

27. [*M23*] What input leads to the *best* case of Algorithm Q? Give approximate values of A, B, C, D, E, X in this case.

28. [*M26*] Find the recurrence relation analogous to (20) which is satisfied by the average number of comparisons in Singleton's modification of Algorithm Q (choosing s as the median of $\{K_1, K_{\lfloor (N+1)/2 \rfloor}, K_N\}$ instead of $s = K_1$).

29. [*HM40*] Find the asymptotic value of the number of comparisons in Singleton's "median of three" method (cf. exercise 28).

30. [*25*] (P. Shackleton.) When *multiword keys* are being sorted, many sorting methods become progressively slower as the file gets closer to its final order, since equal and nearly-equal keys require an inspection of several words to determine the proper lexicographic order. (See exercise 5–5.) Files which arise in practice often involve nearly-equal keys, so this phenomenon can have a significant impact on the sorting time. Algorithm Q has this difficulty; Algorithm R does not, since it inspects only one bit each time (although the presence of equal and nearly-equal keys can greatly increase the running time of Algorithm R for other reasons).

Explain how Algorithm Q can be extended to avoid this difficulty; within a subfile in which the leading k words are known to have constant values for all keys, only the $(k + 1)$st words of the keys should be inspected.

▸ **31.** [*20*] (C. A. R. Hoare.) Suppose that, instead of sorting an entire file, we only want to determine the mth smallest of a given set of n elements. Show that "quicksort" can be adapted to this purpose, avoiding many of the computations required to do a complete sort.

32. [*M40*] Find a simple "closed form" expression for C_{nm}, the average number of key comparisons required to find the mth smallest of n elements by the method of

exercise 31. (For simplicity, avoid all comparisons with $i \geq j$ as in exercise 24.) What is the asymptotic behavior of $C_{(2m-1)m}$, the average number of comparisons needed to find the median of $2m - 1$ elements by Hoare's method?

▶ **33.** [*20*] Design an algorithm which rearranges all the numbers in a given table so that all *negative* values precede all positive ones. (The items need not be sorted completely, just separated between negative and nonnegative.) Your algorithm should use the minimum possible number of exchanges.

34. [*20*] How can the bit-inspection loops of radix exchange (in steps R3 through R6) be speeded up?

35. [*M23*] Analyze the values of the frequencies A, B, C, G, K, L, R, S, and X which arise in radix exchange sorting using "case (i) input."

36. [*M27*] Given a sequence of numbers $\langle a_n \rangle = a_0, a_1, a_2, \ldots$, define its *binomial transform* $\langle \hat{a}_n \rangle = \hat{a}_0, \hat{a}_1, \hat{a}_2, \ldots$ by the rule

$$\hat{a}_n = \sum_k \binom{n}{k} (-1)^k a_k.$$

(a) Prove that $\langle \hat{\hat{a}}_n \rangle = \langle a_n \rangle$. (b) Find the binomial transform of the sequences $\langle 1 \rangle$; $\langle n \rangle$; $\langle \binom{n}{m} \rangle$, for fixed m; $\langle a^n \rangle$, for fixed a; $\langle \binom{n}{m} a^n \rangle$, for fixed a and m. (c) Given that a sequence $\langle x_n \rangle$ satisfies the relation

$$x_n = a_n + 2^{1-n} \sum_{k \geq 2} \binom{n}{k} x_k, \qquad \text{for} \qquad n \geq 2; \qquad x_0 = x_1 = a_0 = a_1 = 0;$$

prove that

$$x_n = \sum_{k \geq 2} \binom{n}{k} (-1)^k \frac{2^{k-1} \hat{a}_k}{2^{k-1} - 1} = a_n + \sum_{k \geq 2} \binom{n}{k} (-1)^k \frac{\hat{a}_k}{2^{k-1} - 1}.$$

37. [*M28*] Determine all sequences $\langle a_n \rangle$ such that $\langle \hat{a}_n \rangle = \langle a_n \rangle$, in the sense of exercise 36.

▶ **38.** [*M30*] Find A_N, B_N, C_N, G_N, K_N, L_N, R_N, and X_N, the average values of the quantities in (29) when radix exchange is applied to "case (ii) input." Express your answers in terms of N and the functions

$$U_n = \sum_{k \geq 2} \binom{n}{k} \frac{(-1)^k}{2^{k-1} - 1} \qquad V_n = \sum_{k \geq 2} \binom{n}{k} \frac{(-1)^k k}{2^{k-1} - 1} = n(U_n - U_{n-1}).$$

[*Hint:* See exercise 36.]

39. [*20*] The results shown in (30) indicate that radix exchange takes about $1.44N$ stages when it is applied to random input. Prove that quicksort will never require more than N stages; and explain why radix exchange often does.

40. [*21*] Explain how to modify Algorithm R so that it works with reasonable efficiency when sorting files containing numerous equal keys.

41. [*23*] Formulate van Emden's "interval exchange sort," obtaining a precise description of an algorithm.

42. [*M43*] Analyze the interval exchange algorithm of exercise 41.

43. [*HM21*] Prove that $\int_0^1 y^{-1}(e^{-y} - 1)\,dy + \int_1^\infty y^{-1}e^{-y}\,dy = -\gamma$. [*Hint:* Consider $\lim_{a\to 0+} y^{a-1}$.]

44. [*HM24*] Derive (37) as suggested in the text.

45. [*HM20*] Explain why (43) is true, when $x > 0$.

46. [*HM20*] What is the value of $(1/2\pi i)\int_{a+i\infty}^{a+i\infty} \Gamma(z)n^{s-z}\,dz/(2^{s-z} - 1)$, given that s is a positive integer and $0 < a < s$?

47. [*HM21*] Prove that $\sum_{j\geq 1} (n/2^j)e^{-n/2^j}$ is a bounded function of n.

48. [*HM24*] Find the asymptotic value of the quantity V_n defined in exercise 38, using a method analogous to the text's study of U_n, obtaining terms up to $O(1)$.

49. [*HM24*] Extend the asymptotic formula (47) for U_n to $O(n^{-1})$.

50. [*HM24*] Find the asymptotic value of the function

$$U_{mn} = \sum_{k\geq 2} \binom{n}{k} (-1)^k \frac{1}{m^{k-1} - 1},$$

when m is any fixed number greater than 1. (When m is an integer greater than 2, this quantity arises in the study of generalizations of radix exchange, as well as the "trie memory" search algorithms of Section 6.3.)

▸ **51.** [*HM28*] Show that the gamma-function approach to asymptotic problems can be used instead of Euler's summation formula to derive the asymptotic expansion of $r_k(m)$. (Cf. Eq. 35). This gives us a uniform method for studying $r_k(m)$ for all k, without relying on "tricks" such as the text's introduction of $g_{-1}(x) = (e^{-x^2} - 1)/x$.)

52. [*HM35*] (N. G. de Bruijn.) What is the asymptotic behavior of the sum

$$S_n = \sum_{t\geq 1} \left(\frac{2n}{n+t}\right) d(t),$$

where $d(t)$ is the number of divisors of t? (Thus, $d(1) = 1$, $d(2) = d(3) = 2$, $d(4) = 3$, $d(5) = 2$, etc. This question arises in connection with the analysis of a tree traversal algorithm, exercise 2.3.1–11.) Find the value of $S_n/\binom{2n}{n}$ to terms of $O(n^{-1})$.

53. [*HM42*] Analyze the average number of bit inspections and exchanges done by radix exchange when the input data consists of infinite-precision binary numbers in [0, 1), each of whose bits is independently equal to 1 with probability p. (Only the case $p = \frac{1}{2}$ is discussed in the text; the methods we have used can be generalized to arbitrary p.) Consider in particular the case $p = 1/\phi = .61803\ldots$.

54. [*HM24*] (S. O. Rice.) Show that U_n can be written

$$U_n = (-1)^n \frac{n!}{2\pi i} \oint_C \frac{dz}{z(z-1)\ldots(z-n)} \frac{1}{2^{z-1} - 1},$$

where C is a skinny closed curve encircling the points $2, 3, \ldots, n$. Changing C to an arbitrarily large circle centered at the origin, derive the convergent series

$$U_n = n(H_{n-1} - 1)/(\ln 2) - \tfrac{1}{2}n + 2 + \frac{2}{\ln 2} \sum_{m\geq 1} \Re\,(B(n+1, -1 + ibm)),$$

where $b = 2\pi/(\ln 2)$, and $B(n+1, -1+ibm) = \Gamma(n+1)\Gamma(-1+ibm)/\Gamma(n+ibm) = n!/\Pi_{0\leq k\leq n}(k - 1 + ibm)$.

▸ **55.** [*22*] Show how Program Q should be modified so that the partitioning element is the median of the three keys (28).

56. [*M44*] Analyze the average behavior of the quantities which occur in the running time of Algorithm Q when the program has been modified to take the median of three elements as in exercise 55. [Cf. exercise 29.]

▸ **57.** [*HM24*] Find the asymptotic value of the number of right-to-left maxima occurring in multiple list insertion, Eq. 5.2.1–11, when $M = N/\alpha$, for fixed α as $N \to \infty$. Carry out the expansion to terms of $O(N^{-1})$, expressing your answer in terms of the "exponential integral" function $E_1(z) = \int_z^\infty e^{-t}\, dt/t$.

5.2.3. Sorting by Selection

Another important family of sorting techniques is based on the idea of repeated selection. The simplest selection method is perhaps the following:

i) Find the smallest key; transfer the corresponding record to the output area; then replace the key by the value "∞" (which is assumed to be higher than any actual key).
ii) Repeat step (i). This time the second smallest key will be selected, since the smallest key has been replaced by ∞.
iii) Continue repeating step (i) until N records have been selected.

Note that such a selection method requires all the input items to be present before sorting may proceed, and it generates the final outputs one by one in sequence. This is essentially the opposite of insertion, where the inputs are received sequentially but we do not know any of the final outputs until sorting is completed.

Some computers (e.g., those with a cyclic drum memory) have a built-in "find the smallest" instruction which operates at high speed. This makes selection sorting by the above method especially attractive when N is not too large.

The above method involves $N - 1$ comparisons each time a new record is selected, and it also requires a separate output area in memory. There is an obvious way to improve upon the situation, avoiding the use of ∞: We can take the selected value and move it into its proper position by exchanging it with the record currently occupying that position. Then we need not consider that position again in future selections. This idea yields our first selection sorting algorithm.

Algorithm S (*Straight selection sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \le \cdots \le K_N$. Sorting is based on the method indicated above, except that it proves to be more convenient to select the *largest* element first, then the second largest, etc.

S1. [Loop on j.] Perform steps S2 and S3 for $j = N, N - 1, \ldots, 2$.

S2. [Find max $(K_1, \ldots, K_j)$.] Search through keys $K_j, K_{j-1}, \ldots, K_1$ to find a maximal one; let it be K_i.

S3. [Exchange with R_j.] Interchange records $R_i \leftrightarrow R_j$. (Now records $R_j, \ldots, R_N$ are in their final position.) ∎

Table 1 shows this algorithm in action on our sixteen example keys; elements which are candidates for the maximum during the search in step S2 are shown in boldface type.

Table 1

STRAIGHT SELECTION SORTING

503	087	512	061	**908**	170	**897**	275	653	426	154	509	612	677	**765**	**703**
503	087	512	061	703	170	**897**	275	653	426	154	509	612	677	765	908
503	087	512	061	703	170	**765**	275	653	426	154	509	612	**677**	897	908
503	087	512	061	**703**	170	**677**	275	**653**	426	154	509	**612**	765	897	908
503	087	512	061	612	170	**677**	275	**653**	426	154	**509**	703	765	897	908
503	087	512	061	612	170	509	275	**653**	**426**	**154**	677	703	765	897	908
...															
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

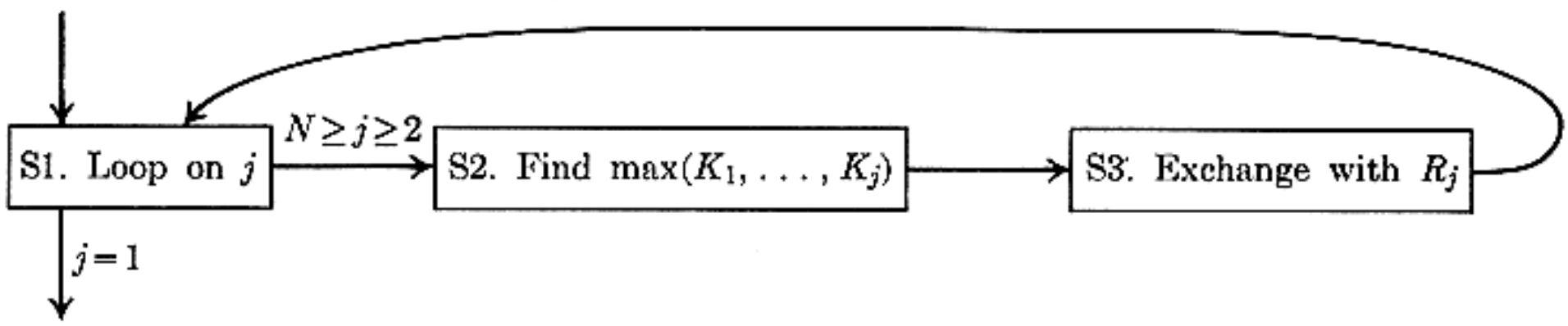


Fig. 21. Straight selection sorting.

The corresponding MIX program is quite simple:

Program 5.2.3S (*Straight selection sort*). As in previous programs of this chapter, the records in locations INPUT+1 through INPUT+N are sorted in place, on a full-word key. rA ≡ current maximum, rI1 ≡ $j - 1$, rI2 ≡ k (index for searching), rI3 ≡ i. Assume that $N \geq 2$.

```
01     START  ENT1  N-1          1        S1. Loop on j. j ← N.
02     2H     ENT2  0,1          N - 1    S2. Find max (K_1, ..., K_j). k ← j - 1.
03            ENT3  1,1          N - 1    i ← j.
04            LDA   INPUT,3      N - 1    rA ← K_i.
05     8H     CMPA  INPUT,2      A
06            JGE   *+3          A        Jump if K_i ≥ K_k.
07            ENT3  0,2          B        Otherwise set i ← k,
08            LDA   INPUT,3      B        rA ← K_i.
09            DEC2  1            A        k ← k - 1.
10            J2P   8B           A        Repeat if k > 0.
11            LDX   INPUT+1,1    N - 1    S3. Exchange with R_j.
12            STX   INPUT,3      N - 1    R_i ← R_j.
13            STA   INPUT+1,1    N - 1    R_j ← rA.
14            DEC1  1            N - 1
15            J1P   2B           N - 1    N ≥ j ≥ 2. ∎
```

The running time of this program depends on the number of items; N; the number of comparisons, A; and the number of "right-to-left maxima," B. It is easy to see that

$$A = \binom{N}{2} = \frac{1}{2}N(N-1), \tag{1}$$

regardless of the values of the input keys; hence only B is variable. In spite of the simplicity of straight selection, this quantity B is not easy to analyze precisely. Exercises 3 through 6 show that

$$B = (\min 0,\ \text{ave}\ (N+1)H_N - 2N,\ \max \lfloor N^2/4 \rfloor); \tag{2}$$

in this case the maximum value turns out to be particularly interesting. (The standard deviation of B has not yet been determined.)

Thus the average running time of Program S is $2.5N^2 + 3(N+1)H_N + 3.5N - 11$ units, just slightly slower than straight insertion (Program 5.2.1S). It is interesting to compare Algorithm S to the bubble sort (Algorithm 5.2.2B), since bubble sorting may be regarded as a selection algorithm which sometimes selects more than one element at a time. For this reason bubble sorting usually does less comparisons than straight selection and it may seem to be preferable; but in fact Program 5.2.2B is more than twice as slow as Program S! Bubble sorting is handicapped by the fact that it does so many exchanges, while straight selection involves very little data movement.

Refinements of straight selection. Is there any way to improve on the selection method used in Algorithm S? For example, take the search for a maximum in step S2; is there a substantially faster way to find a maximum? The answer to the latter question is *no*!

Lemma M. *Every algorithm for finding the maximum of n elements, based on comparing pairs of elements, must make at least $n - 1$ comparisons.*

Proof. If we have made less than $n - 1$ comparisons, there will be at least two elements which have never been found to be less than any others. Therefore we will not know which of these two elements is larger, and we cannot have determined the maximum. ∎

Thus a selection process which finds the largest element must take at least $n - 1$ steps; perhaps all sorting methods based on n repeated selections are doomed to require order n^2 steps? Fortunately Lemma M applies only to the *first* selection step; subsequent selections can make use of previously-gained information. For example, exercise 8 shows that a comparatively simple change to Algorithm S cuts the average number of comparisons in half.

Consider the 16 numbers in Table 1; one way to save time on repeated selections is to regard them as four groups of four. We can start by determining the largest of each group, namely the respective keys

512, 908, 653, 765;

the largest of these four elements, 908, is then the largest of the entire file. To get the second largest we need only look at the other three elements of the group containing 908; the largest of {170, 897, 275} is 897, and the largest of

512, 897, 653, 765

is 897. Similarly to get the third largest element we determine the largest of {170, 275} and then the largest of

512, 275, 653, 765,

and so on. Each selection after the first takes at most 6 additional comparisons. In general, if N is a perfect square, we can divide the file into $\sqrt{N}$ groups of $\sqrt{N}$ elements each; each selection after the first takes at most $\sqrt{N} - 1$ comparisons within the group of the previously selected item, plus $\sqrt{N} - 1$ comparisons among the "group-leaders." This idea is called *quadratic selection*; its total execution time is $O(N\sqrt{N})$, which is substantially better than order N^2.

Quadratic selection was first published by E. H. Friend [*JACM* **3** (1956), 152–154], who pointed out that the same idea can be generalized to cubic, quartic, etc. selection. For example, cubic selection divides the file into $\sqrt[3]{N}$ large groups, each containing $\sqrt[3]{N}$ small groups, each containing $\sqrt[3]{N}$ records; the execution time is proportional to $N\sqrt[3]{N}$. If we carry this idea to its ultimate conclusion we arrive at what Friend called "nth degree selecting," based on a binary tree structure. This method has an execution time proportional to $N \log N$; we shall call it *tree selection*.

Tree selection. The principles of tree selection sorting are easy to understand in terms of matches in a typical "knock-out tournament." Consider, for example, the results of the ping-pong contest shown in Fig. 22; Jim beats Don and Joe beats Jack, then in the next round Joe beats Jim, etc.

Figure 22 shows that Joe is the champion of the eight players, and $8 - 1 = 7$ matches (i.e., comparisons) were required to determine this fact. Dick is not necessarily the second-best player; any of the people defeated by Joe, including the first-round loser Jack, might possibly be second best. We can determine the second-best player by having Jack play Jim, and the winner of that match plays Dick; only two additional matches are required to find the second-best man, because of the structure we have remembered from the earlier games.

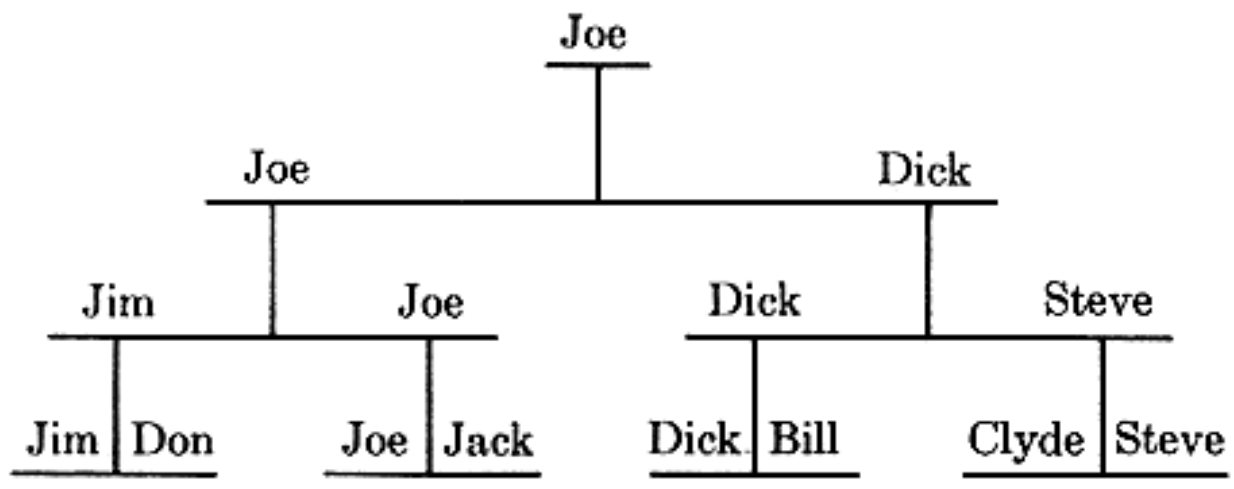
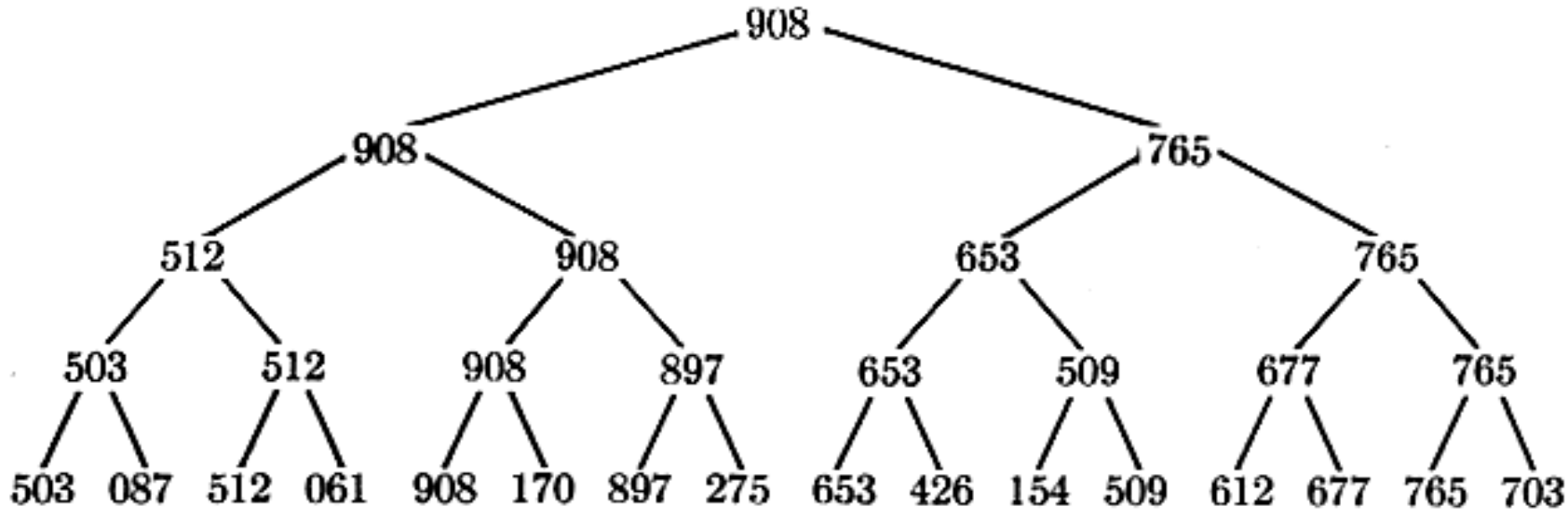


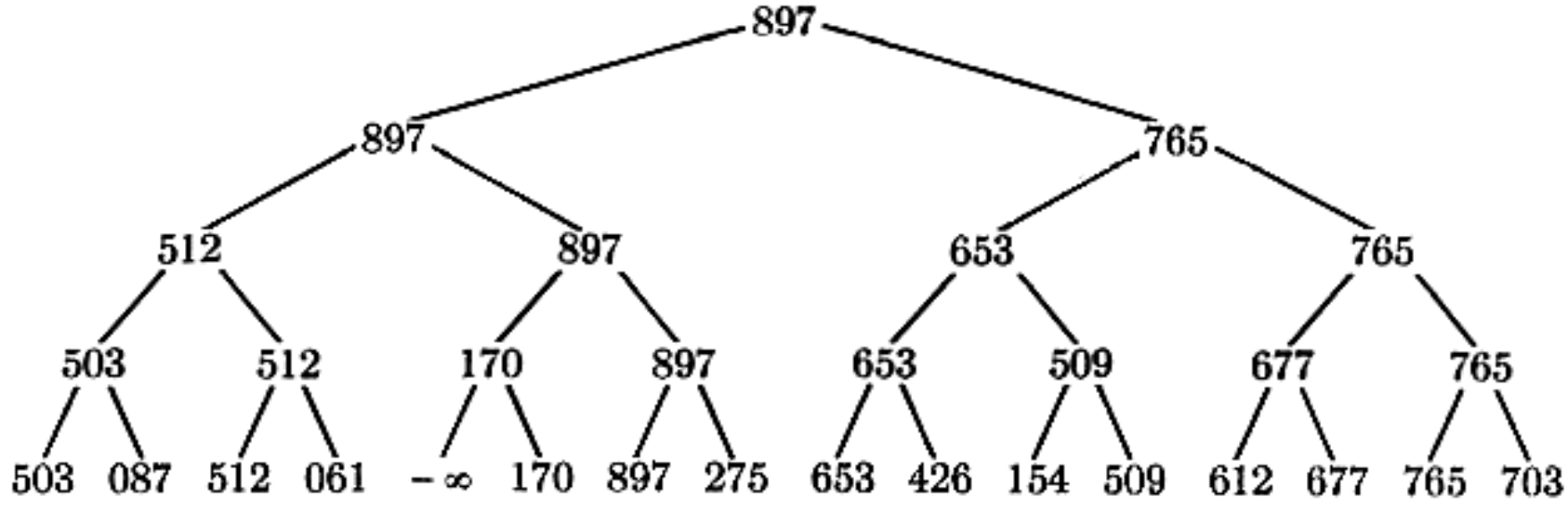
Fig. 22. A ping-pong tournament.

In general, we can "output" the player at the root of the tree, and replace him by an extremely weak player. Substituting this weak player means that the original second-best player will now be best, so he will appear at the root if we recalculate the winners in upper levels of the tree. For this purpose only one path in the tree must be changed, so that less than $\lceil \log_2 N \rceil$ further comparisons are needed to select the next-best player. The total time for such a selection sort is roughly proportional to $N \log N$.

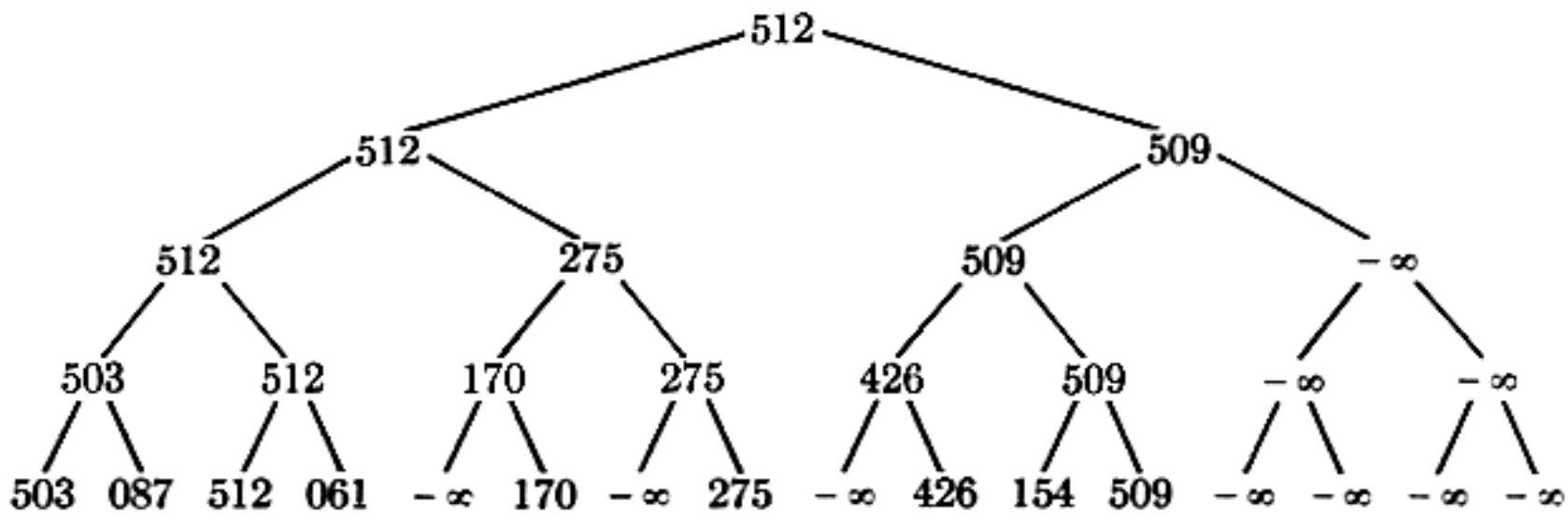
Figure 23 shows tree selection sorting in action, on our 16 example numbers. Note that it is necessary to know where the key at the root came from, in order to know where to insert the next "$-\infty$". Therefore the branch nodes



(a) Initial configuration.



(b) 908 is replaced by $-\infty$, and the second highest element moves up to the root.



(c) Configuration after 908, 897, 765, 703, 677, 653, 612 have been output.

Fig. 23. An example of tree selection sorting.

of the tree actually contain a pointer or index specifying the position of the relevant key, instead of the key itself. It follows that we need memory space for N input records, $N - 1$ pointer words, and N output records. (If the output goes to tape or disk, of course, we don't need to retain the output records in high-speed memory.)

The reader should pause at this point until he understands tree selection, in order to appreciate the remarkable improvements we are about to discuss. For example, it should now be easy to work exercise 10.

One way to modify tree selection, essentially introduced by K. E. Iverson [*A Programming Language*, (Wiley, 1962), 223–227], does away with the need for pointers by "looking ahead" in the following way: When the winner of a match in the bottom level of the tree is moved up, he can be replaced immediately by "$-\infty$" at the bottom level; and when a winner moves up from one branch to another, we can replace him by the one which eventually should move up into his former place (namely the larger of the two keys below). Repeating this operation as often as possible converts Fig. 23(a) into Fig. 24.

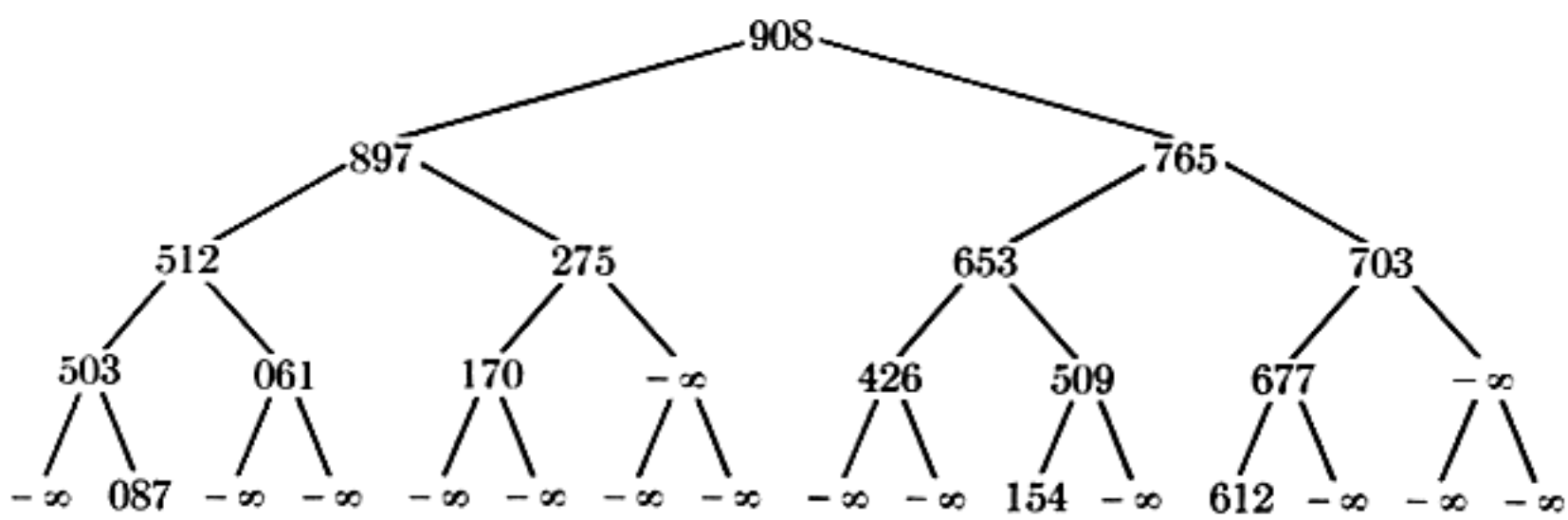


Fig. 24. The Peter Principle applied to sorting. Everyone rises to his level of incompetence in the hierarchy.

Once the tree has been set up in this way we can proceed to sort by a "top-down" method, instead of the "bottom up" method of Fig. 23; we output the root, move up his largest descendant, move up the latter's largest descendant, etc. The process begins to look less like a ping-pong tournament and more like a corporate system of promotions.

The reader should be able to see that this top-down method has the advantage that redundant comparisons of $-\infty$ with $-\infty$ can be avoided. (The bottom-up approach finds $-\infty$ omnipresent in the latter stages of sorting, but the top-down approach can stop modifying the tree during each stage as soon as a $-\infty$ has been stored.)

Figures 23 and 24 are *complete binary trees* with 16 terminal nodes (cf. Section 2.3.4.5), and it is convenient to represent such a tree in consecutive locations as shown in Fig. 25. Note that the father of node number k is node $\lfloor k/2 \rfloor$, and its sons are nodes $2k$ and $2k + 1$. This leads to another advantage of the top-down approach, since it is often considerably simpler to go top-down from node k

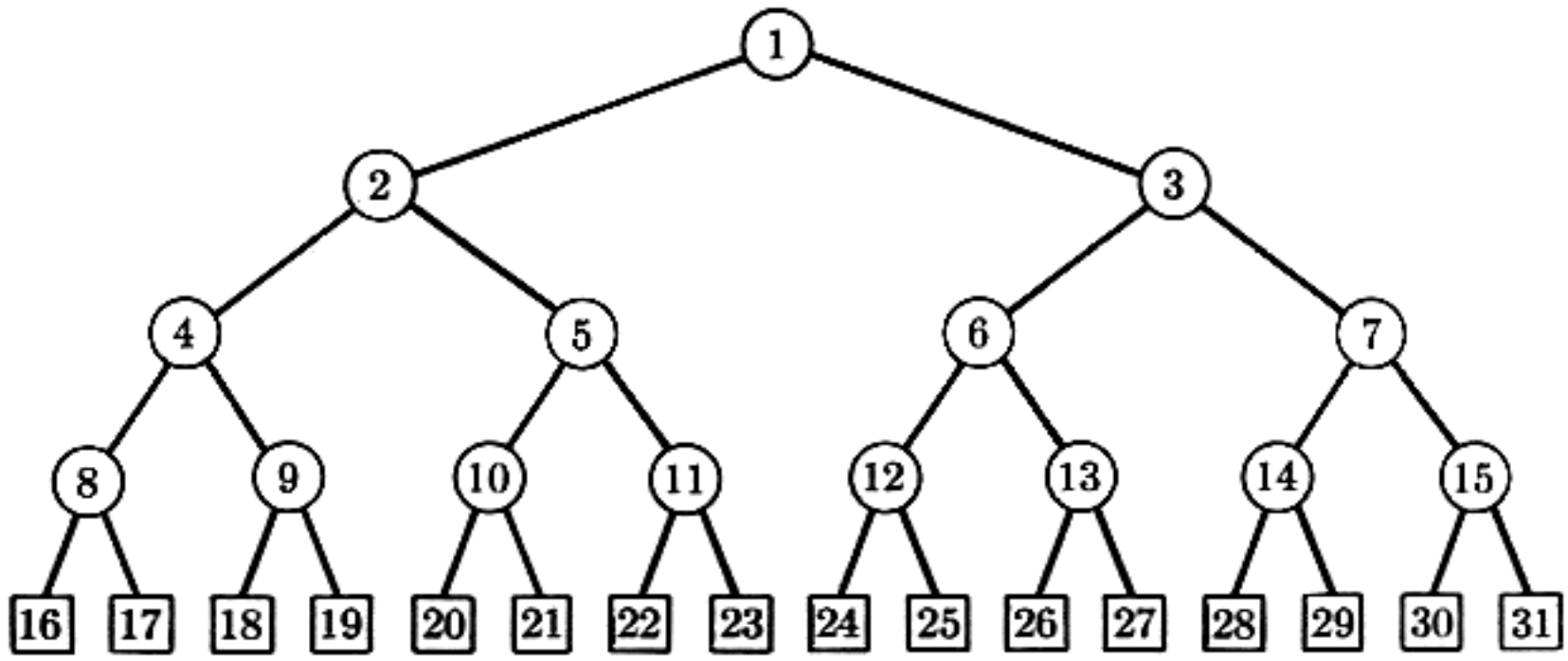


Fig. 25. Sequential storage allocation for a complete binary tree.

to nodes $2k$ and $2k+1$ than bottom-up from node k to nodes $k \oplus 1$ and $\lfloor k/2 \rfloor$. (Here $k \oplus 1$ stands for $k+1$ or $k-1$, according as k is even or odd.)

Our examples of tree selection so far have more or less assumed that N is a power of 2; but actually we can work with arbitrary N, since the complete binary tree with N terminal nodes is readily constructed for any N.

Now we come to the crucial question: Can't we do the top-down method without using "$-\infty$" at all? Wouldn't it be nice if the important information of Fig. 24 were all in locations 1 through 16 of the complete binary tree, without the useless "holes" containing $-\infty$? Some reflection shows that it is indeed possible to achieve this goal, not only eliminating $-\infty$ but also making it possible to sort N records in place without an auxiliary output area. This brings us to an important sorting algorithm which was christened "heapsort" by its discoverer J. W. J. Williams [*CACM* **7** (1964), 347–348].

Heapsort. Let us say that a file of keys $K_1, K_2, \ldots, K_N$ is a "heap" if

$$K_{\lfloor j/2 \rfloor} \geq K_j \qquad \text{for} \qquad 1 \leq \lfloor j/2 \rfloor < j \leq N. \tag{3}$$

Thus, $K_1 \geq K_2$, $K_1 \geq K_3$, $K_2 \geq K_4$, etc.; this is exactly the condition which holds in Fig. 24, and it implies in particular that the largest key appears "on top of the heap,"

$$K_1 = \max(K_1, K_2, \ldots, K_N). \tag{4}$$

If we can somehow transform an arbitrary input file into a heap, we can use a "top-down" selection procedure like that described above to obtain an efficient sorting algorithm.

An efficient approach to heap creation has been suggested by R. W. Floyd [*CACM* **7** (1964), 701]. Let us assume that we have been able to arrange the file so that

$$K_{\lfloor j/2 \rfloor} \geq K_j \qquad \text{for} \qquad l < \lfloor j/2 \rfloor < j \leq N, \tag{5}$$

where l is some number ≥ 1. (In the original file this condition holds "vacu-

ously" for $l = \lfloor N/2 \rfloor$, since no subscript j satisfies the condition $\lfloor N/2 \rfloor < \lfloor j/2 \rfloor < j \leq N$.) It is not difficult to see how to transform the file so that the inequalities in (5) are extended to the case $\lfloor j/2 \rfloor = l$, working entirely in the subtree whose root is node l. Therefore we can decrease l by 1, until condition (3) is finally achieved. These ideas of Williams and Floyd lead to the following elegant algorithm which merits careful study:

Algorithm H (*Heapsort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete, their keys will be in order, $K_1 \leq \cdots \leq K_N$. First we rearrange the file so that it forms a heap, then we repeatedly remove the top of the heap and transfer it to its proper final position. Assume that N ≥ 2.

H1. [Initialize.] Set $l \leftarrow \lfloor N/2 \rfloor + 1$, $r \leftarrow N$.

H2. [Decrease l or r.] If $l > 1$, set $l \leftarrow l - 1$, $R \leftarrow R_l$, $K \leftarrow K_l$. (If $l > 1$, we are in the process of transforming the input file into a heap; on the other hand if $l = 1$, the keys $K_1 K_2 \ldots K_r$ presently constitute a heap.) Otherwise set $R \leftarrow R_r$, $K \leftarrow K_r$, $R_r \leftarrow R_1$, and $r \leftarrow r - 1$; if this makes $r = 1$, set $R_1 \leftarrow R$ and terminate the algorithm.

H3. [Prepare for "sift-up."] Set $j \leftarrow l$. (At this point we have

$$K_{\lfloor j/2 \rfloor} \geq K_j \qquad \text{for} \qquad l < \lfloor j/2 \rfloor < j \leq r; \tag{6}$$

and record R_k is in its final position for $r < k \leq N$. Steps H3–H8 are called the "sift-up" algorithm; their effect is equivalent to setting $R_l \leftarrow R$ and then rearranging $R_l, \ldots, R_r$ so that condition (6) holds also for $\lfloor j/2 \rfloor = l$.)

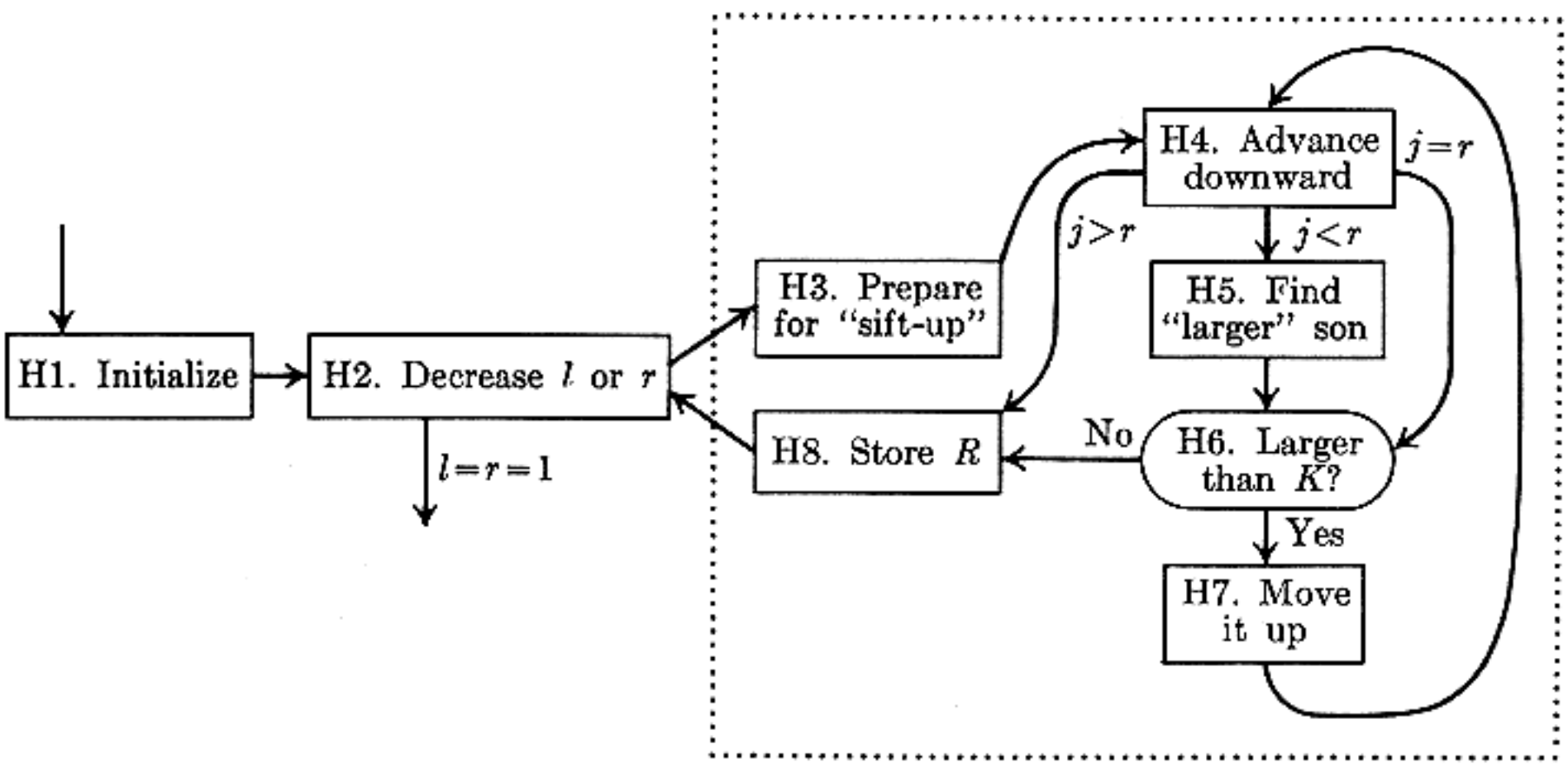


Fig. 26. Heapsort; dotted lines enclose the "sift-up" algorithm.

H4. [Advance downward.] Set $i \leftarrow j$ and $j \leftarrow 2j$. (In the following steps we have $i = \lfloor j/2 \rfloor$.) If $j < r$, go right on to step H5; if $j = r$, go to step H6; and if $j > r$, go to H8.

H5. [Find "larger" son.] If $K_j < K_{j+1}$, then set $j \leftarrow j + 1$.

H6. [Larger than K?] If $K \geq K_j$, then go to step H8.

H7. [Move it up.] Set $R_i \leftarrow R_j$, and go back to step H4.

H8. [Store R.] Set $R_i \leftarrow R$. (This terminates the "sift-up" algorithm initiated in step H3.) Return to step H2. ∎

Heapsort has sometimes been described as the "⧄" algorithm, because of the motion of l and r. The upper triangle represents the heap creation phase, when $r = N$ and l decreases to 1; and the lower triangle represents the selection phase, when $l = 1$ and r decreases to 1. Table 2 shows the process of heapsorting our sixteen example numbers. (Each line in that table shows the state of affairs after step H2, and brackets indicate the position l and r.)

Table 2

EXAMPLE OF HEAPSORT

K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9	K_{10}	K_{11}	K_{12}	K_{13}	K_{14}	K_{15}	K_{16}	l	r	K
503	087	512	061	908	170	897	[275	653	426	154	509	612	677	765	703]	8	16	275
503	087	512	061	908	170	[897	703	653	426	154	509	612	677	765	275]	7	16	897
503	087	512	061	908	[170	897	703	653	426	154	509	612	677	765	275]	6	16	170
503	087	512	061	[908	612	897	703	653	426	154	509	170	677	765	275]	5	16	908
503	087	512	[061	908	612	897	703	653	426	154	509	170	677	765	275]	4	16	061
503	087	[512	703	908	612	897	275	653	426	154	509	170	677	765	061]	3	16	512
503	[087	897	703	908	612	765	275	653	426	154	509	170	677	512	061]	2	16	087
[503	908	897	703	426	612	765	275	653	087	154	509	170	677	512	061]	1	16	503
[908	703	897	653	426	612	765	275	503	087	154	509	170	677	512]	908	1	15	061
[897	703	765	653	426	612	677	275	503	087	154	509	170	061]	897	908	1	14	512
[765	703	677	653	426	612	512	275	503	087	154	509	170]	765	897	908	1	13	061
[703	653	677	503	426	612	512	275	061	087	154	509]	703	765	897	908	1	12	170
[677	653	612	503	426	509	512	275	061	087	154]	677	703	765	897	908	1	11	170
[653	503	612	275	426	509	512	170	061	087]	653	677	703	765	897	908	1	10	154
[612	503	512	275	426	509	154	170	061]	612	653	677	703	765	897	908	1	9	087
[512	503	509	275	426	087	154	170]	512	612	653	677	703	765	897	908	1	8	061
[509	503	154	275	426	087	061]	509	512	612	653	677	703	765	897	908	1	7	170
[503	426	154	275	170	087]	503	509	512	612	653	677	703	765	897	908	1	6	061
[426	275	154	061	170]	426	503	509	512	612	653	677	703	765	897	908	1	5	087
[275	170	154	061]	275	426	503	509	512	612	653	677	703	765	897	908	1	4	087
[170	087	154]	170	275	426	503	509	512	612	653	677	703	765	897	908	1	3	061
[154	087]	154	170	275	426	503	509	512	612	653	677	703	765	897	908	1	2	061
[061]	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908	1	1	061

Program H (*Heapsort*). The records in locations INPUT+1 through INPUT+N are sorted by Algorithm H, with the following register assignments: rI1 $\equiv l - 1$, rI2 $\equiv r - 1$, rI3 $\equiv i$, rI4 $\equiv j$, rI5 $\equiv r - l$, rA $\equiv K \equiv R$, rX $\equiv R_j$.

01	START	ENT1	N/2	1	*H1. Initialize.* $l \leftarrow \lfloor N/2 \rfloor + 1$.
02		ENT2	N-1	1	$r \leftarrow N$.
03	1H	DEC1	1	$\lfloor N/2 \rfloor$	$l \leftarrow l - 1$.
04		LDA	INPUT+1,1	$\lfloor N/2 \rfloor$	$R \leftarrow R_l$, $K \leftarrow K_l$.
05	3H	ENT4	1,1	P	*H3. Prepare for "sift-up."* $j \leftarrow l$.
06		ENT5	0,2	P	
07		DEC5	0,1	P	rI5 $\leftarrow r - j$.
08		JMP	4F	P	To H4.
09	5H	LDX	INPUT,4	$B + A - D$	*H5. Find "larger" son.*
10		CMPX	INPUT+1,4	$B + A - D$	
11		JGE	6F	$B + A - D$	Jump if $K_j \geq K_{j+1}$.
12		INC4	1	C	Otherwise set $j \leftarrow j + 1$.
13		DEC5	1	C	
14	9H	LDX	INPUT,4	$C + D$	rX $\leftarrow R_j$.
15	6H	CMPA	INPUT,4	$B + A$	*H6. Larger than K?*
16		JGE	8F	$B + A$	To H8 if $K \geq K_j$.
17	7H	STX	INPUT,3	B	*H7. Move it up.* $R_i \leftarrow R_j$.
18	4H	ENT3	0,4	$B + P$	*H4. Advance downward.* $i \leftarrow j$.
19		DEC5	0,4	$B + P$	rI5 $\leftarrow$ rI5 $- j$.
20		INC4	0,4	$B + P$	$j \leftarrow j + j$.
21		J5P	5B	$B + P$	To H5 if $j < r$.
22		J5Z	9B	$P - A + D$	To H6 if $j = r$.
23	8H	STA	INPUT,3	P	*H8. Store R.* $R_i \leftarrow R$.
24	2H	J1P	1B	P	*H2. Decrease l or r.*
25		LDA	INPUT+1,2	$N - 1$	If $l = 1$, set $R \leftarrow R_r$, $K \leftarrow K_r$.
26		LDX	INPUT+1	$N - 1$	
27		STX	INPUT+1,2	$N - 1$	$R_r \leftarrow R_1$.
28		DEC2	1	$N - 1$	$r \leftarrow r - 1$.
29		J2P	3B	$N - 1$	To H3 if $r > 1$.
30		STA	INPUT+1	1	$R_1 \leftarrow R$. ∎

This program is only about twice as long as Program S, but it is much more efficient when N is large. Its running time depends on

$P = N + \lfloor N/2 \rfloor - 2$, the number of sift-up passes;

A, the number of sift-up passes in which the key K finally lands in an interior node of the heap;

B, the total number of keys scanned during sift-ups;

C, the number of times $j \leftarrow j + 1$ in step H5; and

D, the number of times $j = r$ in step H4.

These quantities are analyzed below; in practice they show comparatively little fluctuation about their average values,

$$A \approx 0.349N, \qquad B \approx N\log_2 N - 1.87N,$$
$$C \approx \tfrac{1}{2}N\log_2 N - 0.9N, \qquad D \approx \ln N. \tag{7}$$

For example, when $N = 1000$, four experiments on random input gave $(A, B, C, D) = (371, 8055, 4056, 12)$, $(351, 8072, 4087, 14)$, $(341, 8094, 4017, 8)$, $(340, 8108, 4083, 13)$, respectively. The total running time, $7A + 14B + 4C + 20N - 2D + 15\lfloor N/2 \rfloor - 28$, is therefore approximately $16N\log_2 N + 0.2N$ units on the average.

A glance at Table 2 makes it had to believe that heapsort is very efficient; large keys shift to the left before we stash them at the right! It is indeed a strange way to sort, when N is small; the sorting time for the 16 keys in Table 2 is $1068u$, while the simple method of straight insertion (Program 5.2.1S) takes only $514u$. Straight selection (Program S) takes $853u$.

For larger N, Program H is more efficient. It invites comparison with Shell's diminishing increment sort (Program 5.2.1D) and Hoare's quicksort (Program 5.2.2Q), since all three programs sort by comparisons of keys and use little or no auxiliary storage. When $N = 1000$, the approximate average running times are

$160000u$ for heapsort,

$130000u$ for Shell's sort,

$80000u$ for quicksort.

(MIX is typical of most present-day computers, but particular machines will of course yield somewhat different relative values.) As N gets larger, heapsort will be superior to Shell's sort, but the asymptotic formula $16N\log_2 N \approx 23.08N\ln N$ will never beat quicksort's $12.67N\ln N$. A modification of heapsort discussed in exercise 18 will speed up the process on many computers, but even this improvement falls short of quicksort.

On the other hand, quicksort is efficient only on the average, and its worst case is of order N^2. Heapsort has the interesting property that its worst case isn't much worse than the average: We always have

$$A \le 1.5N, \qquad B \le N\lfloor\log_2 N\rfloor, \qquad C \le N\lfloor\log_2 N\rfloor, \tag{8}$$

so Program H will take no more than $18N\lfloor\log_2 N\rfloor + 38N$ units of time, regardless of the distribution of the input data. Heapsort is the first sorting method we have seen which is *guaranteed* to be of order $N\log N$. Merge sorting, discussed in Section 5.2.4 below, also has this property, but it requires more memory space.

Largest in, first out. We have seen in Chapter 2 that linear lists can often be classified in a meaningful way by the nature of the insertion and deletion operations performed on them. A *stack* has "last in, first out" behavior, in the sense that every deletion removes the youngest item in the list (the item which was inserted most recently of all items currently present). A simple *queue* has

"first in, first out" behavior, in the sense that every deletion removes the oldest remaining item. In more complex situations, such as the elevator simulation of Section 2.2.5, we want a "smallest in, first out" list, where every deletion removes the item having the smallest key. Such a list may be called a *priority queue*, since the key of each item reflects its relative ability to get out of the list quickly. Selection sorting is a special case of a priority queue in which we do N insertions followed by N deletions.

Priority queues arise in a wide variety of applications. For example, some numerical iterative schemes are based on repeated selection of an item having the largest (or smallest) value of some test criterion; parameters of the selected item are changed, and it is reinserted into the list with a new test value, based on the new values of its parameters. Operating systems often make use of priority queues for the scheduling of jobs. Exercises 15, 29, and 36 mention other typical applications of priority queues, and many other examples will appear in later chapters.

How shall we implement priority queues? One of the obvious methods is to maintain a sorted list, containing the items in order of their keys. Inserting a new item is then essentially the same problem we have treated in our study of insertion sorting, Section 5.2.1. Another even more obvious way to deal with priority queues is to keep the list of elements in arbitrary order, selecting the appropriate element each time a deletion is required by finding the largest (or smallest) key. The trouble with both of these obvious approaches is that they require on the order of N steps either for insertion or deletion, when there are N entries in the list, so they are very time-consuming when N is large.

In his original paper on heapsorting, Williams pointed out that heaps are ideally suited to large priority queue applications, since we can insert or delete elements from a heap in $O(\log N)$ steps; furthermore, all elements of the heap are compactly located in consecutive memory locations. The selection phase of Algorithm H is a sequence of deletion steps of a *largest-in-first-out* process: to delete the largest element K_1 we remove it and "sift-up" K_N into a new heap of $N - 1$ elements. (If we want a smallest-in-first-out algorithm, as in the elevator simulation, we can obviously change the definition of heap so that "$\geq$" becomes "$\leq$" in (3); for convenience, we shall consider only the largest-in-first-out case here.) In general, if we want to delete the largest item and then insert a new element x, we can do the sift-up procedure with $l = 1$, $r = N$, and $K = x$. If we wish to insert an element x without a prior deletion, we can use the "bottom-up" procedure of exercise 16.

A linked representation for priority queues. An efficient way to represent priority queues as linked binary trees was discovered in 1971 by Clark A. Crane. His method requires two link fields and a small count field in every record, but it has the following advantages over a heap:

1) When the priority queue is being treated as a stack, the insertion and deletion operations are more efficient (they take a fixed time independent of the queue size).

2) The records never move, only the pointers change.

3) It is possible to merge two disjoint priority queues, having a total of N elements, into a single priority queue, in only $O(\log N)$ steps.

Crane's original method, slightly modified, is illustrated in Fig. 27, which shows a special kind of binary tree structure. Each node contains a KEY field, a DIST field, and two link fields LEFT and RIGHT. The DIST field is always set equal to the length of the shortest path from that node to a leaf (i.e., an empty Λ node) of the tree. If we regard DIST(Λ) = 0 and KEY(Λ) = $-\infty$, the KEY and DIST fields in the tree satisfy the following properties:

$$\texttt{KEY(P)} \geq \texttt{KEY(LEFT(P))}, \qquad \texttt{KEY(P)} \geq \texttt{KEY(RIGHT(P))}; \tag{9}$$

$$\texttt{DIST(P)} = 1 + \min(\texttt{DIST(LEFT(P))}, \texttt{DIST(RIGHT(P))}); \tag{10}$$

$$\texttt{DIST(LEFT(P))} \geq \texttt{DIST(RIGHT(P))}. \tag{11}$$

Relation (9) is analogous to the heap condition (3), it guarantees that the root of the tree has the largest key; and relation (10) is just the definition of the DIST fields as stated above. Relation (11) is the interesting innovation: It implies that a shortest path to a leaf may always be obtained by moving to the right. We shall say that a binary tree with this property is a *leftist tree*, because it tends to "lean" so heavily to the left.

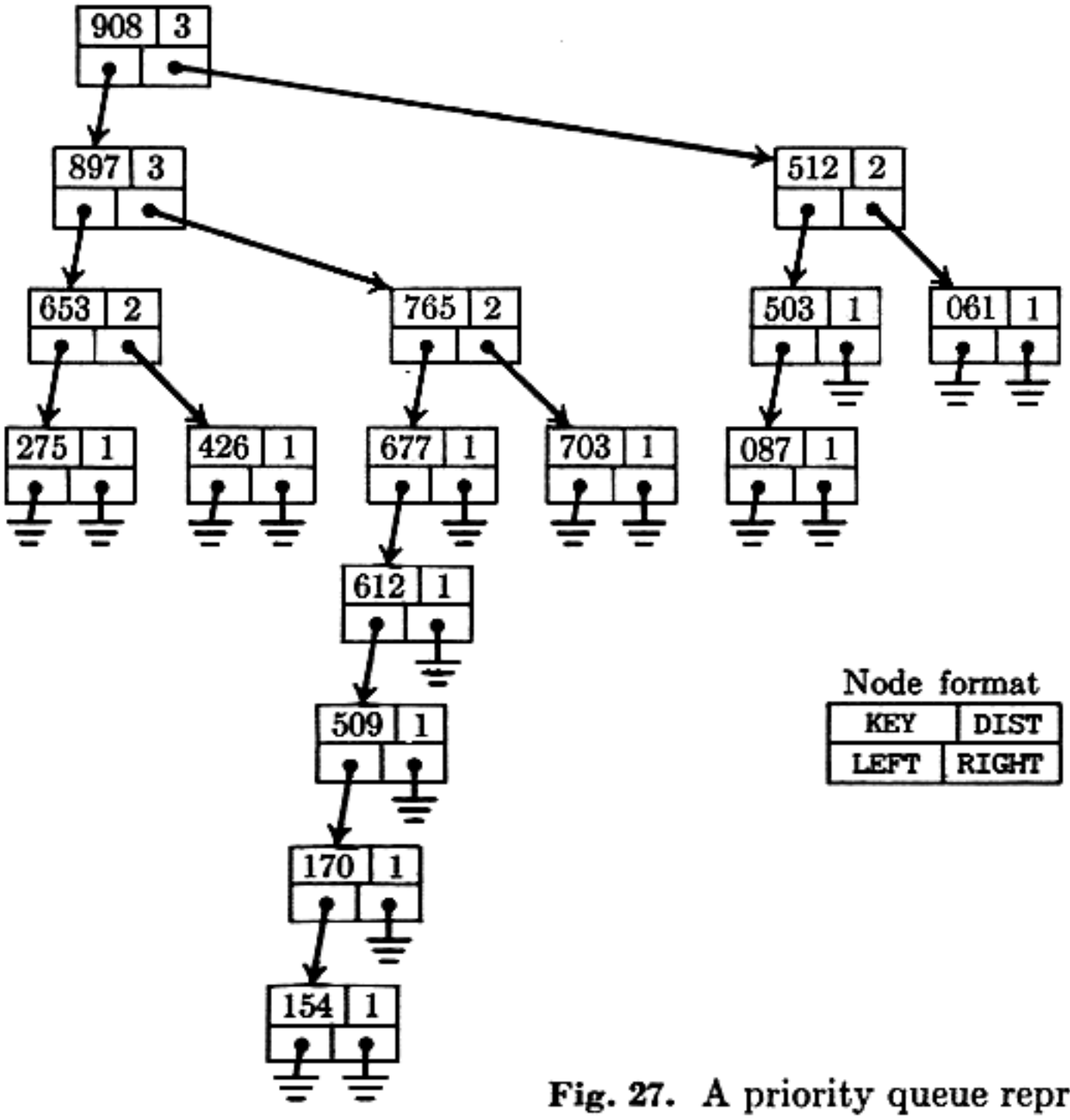


Fig. 27. A priority queue represented as a leftist tree.

It is clear from these definitions that DIST(P) $= n$ implies the existence of at least 2^n leaf nodes below P; otherwise there would be a shorter path from P to a leaf. Thus, if there are N nodes in a leftist tree, the path leading downward from the root towards the right contains at most $\lfloor \log_2 (N + 1) \rfloor$ nodes. It is possible to insert a new node into the priority queue by traversing this path (see exercise 32); hence only $O(\log N)$ steps are needed in the worst case. The best case occurs when the tree is linear (all RIGHT links are Λ), and the worst case occurs when the tree is perfectly balanced.

To remove the node at the root, we simply need to merge its two subtrees. The operation of merging two disjoint leftist trees, pointed to respectively by P and Q, is conceptually simple: If KEY(P) $\geq$ KEY(Q) we take P as the root and merge Q with P's right subtree; then DIST(P) is updated, and LEFT(P) is interchanged with RIGHT(P) if necessary. A detailed description of this process is not difficult to devise (see exercise 32).

Comparison of priority queue techniques. When the number of nodes, N, is small, it is best to use one of the straightforward linear list methods to maintain a priority queue; but when N is large, a $\log N$ method is obviously much faster. Therefore large priority queues are generally represented as heaps or as leftist trees. We shall discuss another way to represent linear lists in Section 6.2.3, via *balanced trees*, and this leads to a third $\log N$ method suitable for representing priority queues. It is therefore appropriate to compare these three techniques.

We have seen that leftist tree operations tend to be slightly faster than heap operations, although heaps take less memory space. Balanced trees take about the same space as leftist trees, perhaps slightly less; the operations are slower than heaps, and the programming is more complicated, but the balanced tree structure is considerably more flexible in several ways. When using a heap or a leftist tree we cannot predict very easily what will happen to two items with equal keys; it is impossible to guarantee that items with equal keys will be treated in a last-in-first-out or first-in-first-out manner, unless the key is extended to include an additional "serial number of insertion" field so that no equal keys are really present. With balanced trees, on the other hand, we can easily stipulate consistent conventions about equal keys, and we can also do things such as "insert x immediately before (or after) y." Balanced trees are symmetrical so that we can delete either the largest or the smallest element at any time, while heaps and leftist trees must be oriented one way or the other. (See exercise 31, however, which shows how to construct *symmetrical* heaps.) Balanced trees can be used for searching as well as for sorting; and we can rather quickly remove consecutive blocks of elements from a balanced tree. But two balanced trees cannot be merged in less than $O(N)$ steps, while leftist trees can be merged in only $O(\log N)$ steps.

In summary, heaps use minimum memory; leftist trees are great for merging disjoint priority queues; and the flexibility of balanced trees is available, if necessary, at reasonable cost.

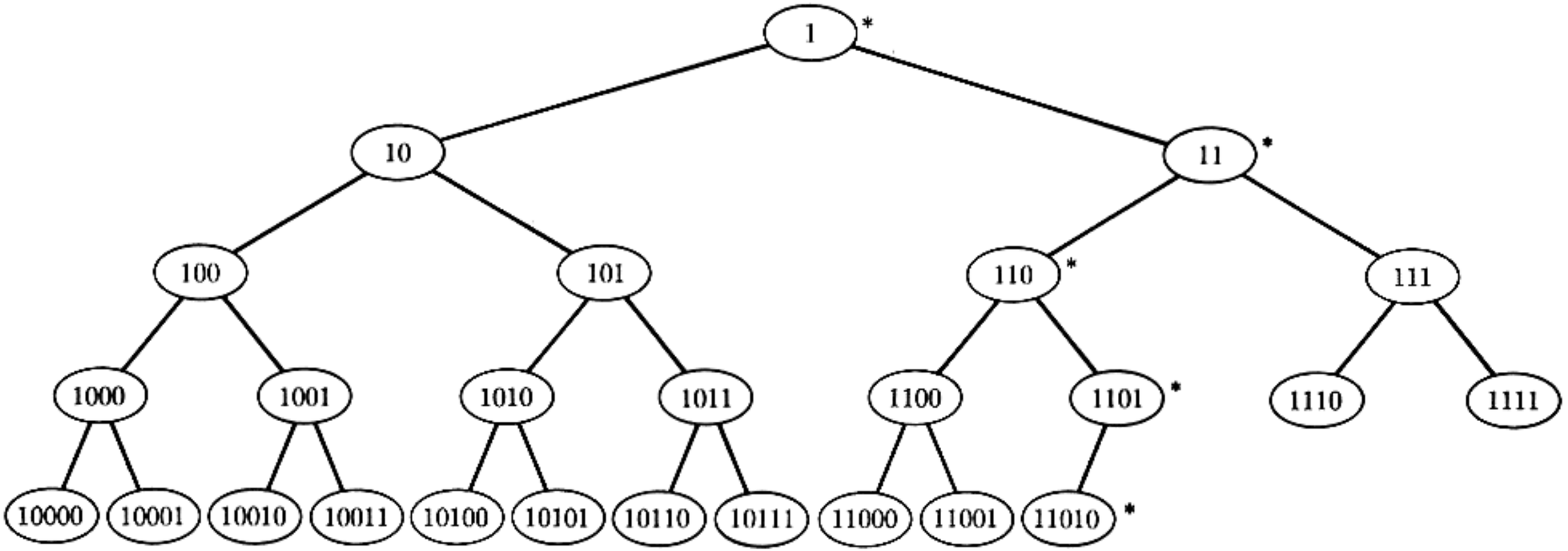


Fig 28. A heap of $26 = (11010)_2$ elements looks like this.

***Analysis of heapsort.** Algorithm H has not yet been analyzed completely, but several of its properties can be deduced without great difficulty. Therefore we shall conclude this section by studying the anatomy of a heap in some detail.

Figure 28 shows the shape of a heap with 26 elements; each node has been labeled in binary notation corresponding to its subscript in the heap. Asterisks in this diagram denote the so-called *special nodes*, which lie on the path from 1 to N.

One of the most important attributes of a heap is the collection of its subtree sizes. For example, in Fig. 28 the sizes of the subtrees rooted at 1, 2, . . . , 26 are, respectively,

$$26^*,\ 15,\ 10^*,\ 7,\ 7,\ 6^*,\ 3,\ 3,\ 3,\ 3,\ 3,\ 3,\ 2^*,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1^*. \tag{12}$$

Asterisks denote *special subtrees*, rooted at the special nodes; exercise 20 shows that if the binary representation of N is

$$N = (b_n b_{n-1} \ldots b_1 b_0)_2, \qquad n = \lfloor \log_2 N \rfloor, \tag{13}$$

the special subtree sizes are always

$$(1b_{n-1} \ldots b_1 b_0)_2,\ (1b_{n-2} \ldots b_1 b_0)_2,\ \ldots,\ (1b_1 b_0)_2,\ (1b_0)_2,\ (1)_2. \tag{14}$$

Nonspecial subtrees are always perfectly balanced, so their size is always of the form $2^k - 1$. Exercise 21 shows that the nonspecial sizes consist of exactly

$$\begin{gathered} \lfloor (N-1)/2 \rfloor \text{ 1's}, \qquad \lfloor (N-2)/4 \rfloor \text{ 3's}, \\ \lfloor (N-4)/8 \rfloor \text{ 7's}, \qquad \ldots, \qquad \lfloor (N - 2^{n-1})/2^n \rfloor \ (2^n - 1)\text{'s}. \end{gathered} \tag{15}$$

For example, Fig. 28 contains twelve nonspecial subtrees of size 1, six of size 3, two of size 7, and one of size 15.

Let s_l be the size of the subtree whose root is l, and let M_N be the multiset $\{s_1, s_2, \ldots, s_N\}$ of all these sizes. We can calculate M_N easily for any given N by using (14) and (15). Exercise 5.1.4–20 tells us that the total number of ways to arrange the integers $\{1, 2, \ldots, N\}$ into a heap is

$$N! \,/\, s_1 s_2 \ldots s_N \;=\; N! \,/ \prod_{s \in M_N} s. \tag{16}$$

For example, the number of ways to place the 26 letters $\{A, B, C, \ldots, Z\}$ into Fig. 28 so that vertical lines preserve alphabetic order is

$$26!/(26 \cdot 10 \cdot 6 \cdot 2 \cdot 1 \cdot 1^{12} \cdot 3^6 \cdot 7^2 \cdot 15^1).$$

We are now in a position to analyze the heap-creation phase of Algorithm H, namely, the computations which take place before the condition $l = 1$

occurs for the first time in step H2. Fortunately we can reduce the study of heap creation to the study of independent sift-up operations, because of the following theorem.

Theorem H. *If the input to Algorithm H is a random permutation of* $\{1, 2, \ldots, N\}$, *then each of the* $N!/(\Pi_{s\in M_N} s)$ *possible heaps is an equally likely outcome of the heap creation phase. Moreover, each of the* $\lfloor N/2 \rfloor$ *sift-up operations performed during this phase is "uniform," in the sense that each of the* s_l *possible values of* i *is equally likely when step H8 is reached.*

Proof. We can apply what numerical analysts might call a "backwards analysis"; given a possible result $K_1 \ldots K_N$ of the sift-up operation rooted at l, we see that there are exactly s_l prior configurations $K'_1 \ldots K'_N$ of the file which will siftup to that result. Each of these prior configurations has a different value of K'_l; hence, working backwards, there are exactly $s_l s_{l+1} \ldots s_N$ input permutations of $\{1, 2, \ldots, N\}$ which yield the configuration $K_1 \ldots K_N$ after the siftup at position l has been completed.

The case $l = 1$ is typical: Let $K_1 \ldots K_N$ be a heap, and let $K'_1 \ldots K'_N$ be a file which is transformed by siftup into $K_1 \ldots K_N$ when $l = 1$, $K = K'_1$. If $K = K_i$, we must have $K'_i = K_{\lfloor i/2 \rfloor}$, $K'_{\lfloor i/2 \rfloor} = K_{\lfloor i/4 \rfloor}$, etc., while $K'_j = K_j$ for all j not on the path from 1 to i. Conversely, for each i this construction yields a file $K'_1 \ldots K'_N$ such that (a) sift-up transforms $K'_1 \ldots K'_N$ into $K_1 \ldots K_N$, and (b) $K_{\lfloor j/2 \rfloor} \geq K_j$ for $2 \leq \lfloor j/2 \rfloor < j \leq N$. Therefore exactly N such files $K'_1 \ldots K'_N$ are possible, and the sift-up operation is uniform. (An example of the proof of this theorem appears in exercise 22.) ∎

Referring to the quantities A, B, C, D in the analysis of Program H, we can see that a uniform sift-up operation on a subtree of size s contributes $\lfloor s/2 \rfloor/s$ to the average value of A; it contributes

$$\frac{1}{s}(0 + 1 + 1 + 2 + \cdots + \lfloor \log_2 s \rfloor) = \frac{1}{s}\Bigg(\sum_{1\leq k\leq s} \lfloor \log_2 k \rfloor\Bigg)$$
$$= \frac{1}{s}\left((s+1)\lfloor \log_2 s \rfloor - 2^{\lfloor \log_2 s \rfloor + 1} + 2\right)$$

to the average value of B (see exercise 1.2.4–42); and it contributes either $2/s$ or 0 to the average value of D, according as s is even or odd. The corresponding contribution to C is somewhat more difficult to determine, so it has been left to the reader (see exercise 26). Summing over all sift-ups, we find that the average value of A during heap creation is

$$A'_N = \sum_{s\in M_N} \lfloor s/2 \rfloor/s, \tag{17}$$

and similar formulas hold for B, C, and D. It is therefore possible to compute these average values exactly without great difficulty, and the following table shows typical results:

N	A'_N	B'_N	C'_N	D'_N
99	19.18	68.35	42.95	0.00
100	19.93	69.39	42.71	1.84
999	196.16	734.66	464.53	0.00
1000	196.94	735.80	464.16	1.92
9999	1966.02	7428.18	4695.54	0.00
10000	1966.82	7429.39	4695.06	1.97
10001	1966.45	7430.07	4695.84	0.00
10002	1967.15	7430.97	4695.95	1.73

Asymptotically speaking, we may ignore the special subtree sizes in M_N, and we find for example that

$$A'_N = \frac{N}{2}\cdot\frac{0}{1} + \frac{N}{4}\cdot\frac{1}{3} + \frac{N}{8}\cdot\frac{3}{7} + \cdots + O(\log N) = (1 - \tfrac{1}{2}\alpha)N + O(\log N), \tag{18}$$

where

$$\alpha = \sum_{k\ge 1}\frac{1}{2^k - 1} = 1.60669\ 51524\ 15291\ 76378\ 33015\ 23190\ 92458\ 04805\text{—}. \tag{19}$$

(This value has been computed by J. W. Wrench, Jr., using the series transformation of exercise 27.) For large N, we may use the approximate formulas

$$\begin{aligned} A'_N &\approx 0.1967N + 0.3\ (N \text{ even}), \qquad 0.1967N - 0.3\ (N \text{ odd});\\ B'_N &\approx 0.74403N - 1.3\ln N;\\ C'_N &\approx 0.47034N - 0.8\ln N;\\ D'_N &\approx 1.8 \pm 0.2\ (N \text{ even}), \qquad 0\ (N \text{ odd}). \end{aligned} \tag{20}$$

The minimum and maximum values are also readily determined. Only $O(N)$ steps are needed to create the heap (cf. exercise 23).

This essentially disposes of the heap creation phase of Algorithm H. But the selection phase is another story, which remains to be written! Let A''_N, B''_N, C''_N, and D''_N denote the average values of A, B, C, and D during the selection phase when N elements are being heapsorted. The behavior of Algorithm H on random input is subject to comparatively little fluctuation about the empirically-determined average values

$$\begin{aligned} A''_N &\approx 0.152N;\\ B''_N &\approx N\log_2 N - 2.61N;\\ C''_N &\approx \tfrac{1}{2}N\log_2 N - 1.4N;\\ D''_N &\approx \ln N \pm 2; \end{aligned} \tag{21}$$

but no adequate theoretical explanation for these constants has yet been found. It is not hard to deduce the upper bounds stated in (8), by considering individual sift-up operations, but the upper bound for C should probably be lowered to about $\frac{1}{2}N \log_2 N$ when the algorithm as a whole is considered.

EXERCISES

1. [*10*] Is straight selection (Algorithm S) a "stable" sorting method?

2. [*15*] Why does it prove to be more convenient to select the largest key, then the second-largest, etc., in Algorithm S, instead of first finding the smallest, then the second-smallest, etc.?

3. [*M21*] (a) Prove that if Algorithm S starts with a random permutation of $\{1, 2, \ldots, N\}$, the first iteration of steps S2 and S3 yields a random permutation of $\{1, 2, \ldots, N-1\}$ followed by N. (In other words, the presence of each permutation of $\{1, 2, \ldots, N-1\}$ in $K_1 \ldots K_{N-1}$ is equally likely.) (b) Therefore if B_N denotes the average value of the quantity B in Program S, given randomly ordered input, we have $B_N = H_N - 1 + B_{N-1}$. [*Hint:* Cf. Eq. 1.2.10–16.]

▸ **4.** [*M25*] Step S3 of Algorithm S accomplishes nothing when $i = j$; is it a good idea to test whether or not $i = j$ before doing step S3? What is the average number of times the condition $i = j$ will occur in step S3 for random input?

5. [*20*] What is the value of the quantity B in the analysis of Program S, when the input is $N \ldots 3\ 2\ 1$?

6. [*M29*] (a) Let $a_1\, a_2 \ldots a_N$ be a permutation of $\{1, 2, \ldots, N\}$ having C cycles, I inversions, and B changes to the right-to-left maxima when sorted by Program S. Prove that $2B \leq I + N - C$. [*Hint:* See exercise 5.2.2–1.] (b) Show that $I + N - C \leq \lfloor n^2/2 \rfloor$; hence B can never exceed $\lfloor n^2/4 \rfloor$.

7. [*M46*] Find the variance of the quantity B in Program S, as a function of N, assuming random input.

▸ **8.** [*24*] Show that if the search for $\max(K_1, \ldots, K_j)$ in step S2 is carried out by examining keys in left-to-right order $K_1, K_2, \ldots, K_j$, instead of $K_j, \ldots, K_2, K_1$ as in Program S, it is often possible to reduce the number of comparisons needed on the next iteration of step S2. Write a MIX program based on this observation.

9. [*M25*] What is the average number of comparisons performed by the algorithm of exercise 8, for random input?

10. [*12*] What will be the configuration of the tree in Fig. 23 after 14 of the original 16 items have been output?

11. [*10*] What will be the configuration of the tree in Fig. 24 after the element 908 has been output?

12. [*M20*] How many times will $-\infty$ be compared with $-\infty$ when the "bottom up" method of Fig. 23 is used to sort a file of 2^n elements into order?

13. [*20*] (J. W. J. Williams.) Step H4 of Algorithm H distinguishes between the three cases $j < r$, $j = r$, and $j > r$. Show that if $K \geq K_{r+1}$ it would be possible to simplify step H4 so that only a two-way branch is made. How could the condition $K \geq K_{r+1}$ be ensured throughout the heapsort process, by modfying step H2?

14. [*10*] Show that simple queues are special cases of priority queues. (Explain how keys can be assigned to the elements so that a largest-in-first-out procedure is equivalent to first-in-first-out.) Is a stack also a special case of a priority queue?

▶**15.** [*M22*] (B. A. Chartres.) Design a high-speed algorithm which builds a table of the prime numbers $\leq N$, making use of a *priority queue* to avoid division operations. [*Hint:* Let the smallest key in the priority queue be the least odd nonprime number greater than the last odd number considered as a prime candidate. Try to minimize the number of elements in the queue.]

16. [*20*] Design an efficient algorithm which inserts a new key into a given heap of n elements, producing a heap of $n + 1$ elements.

17. [*20*] The algorithm of exercise 16 can be used for heap creation, instead of the "decrease l to 1" method used in Algorithm H. Do both methods create the same heap when they begin with the same input file?

▶**18.** [*21*] (R. W. Floyd.) During the selection phase of heapsort, the key K tends to be quite small, so that nearly all of the comparisons in step H6 find $K < K_j$. Show how to modify the algorithm so that K is not compared with K_j in the main loop of the computation.

19. [*21*] Design an algorithm which deletes a given element of a heap of length N, producing a heap of length $N - 1$.

20. [*M20*] Prove that (14) gives the special subtree sizes in a heap.

21. [*M24*] Prove that (15) gives the nonspecial subtree sizes in a heap.

▶**22.** [*20*] What permutations of {1, 2, 3, 4, 5} are transformed into 5 3 4 1 2 by the heap creation phase of Algorithm H?

23. [*M28*] (a) Prove that the length of scan, B, in a sift-up algorithm never exceeds $\lfloor \log_2 (r/l) \rfloor$. (b) According to (8), B can never exceed $N\lfloor \log_2 N \rfloor$ in any particular application of Algorithm H. Find the maximum value of B as a function of N, taken over all possible input files. (You must prove that an input file exists such that B takes on this maximum value.)

24. [*M24*] Derive an exact formula for the standard deviation of B'_N (the total length of scan during the heap-creation phase of Algorithm H).

25. [*M20*] What is the average value of the contribution to C made during the sift-up pass when $l = 1$ and $r = N$, if $N = 2^{n+1} - 1$?

26. [*M30*] Solve exercise 25, (a) for $N = 26$, (b) for general N.

27. [*M25*] (J. W. Wrench, Jr.) Prove that $\sum_{n\geq 1} x^n/(1 - x^n) = \sum_{n\geq 1} x^{n^2}(1 + x^n)/(1 - x^n)$. (Setting $x = \frac{1}{2}$ gives a very rapidly converging series for evaluating (19).)

28. [*35*] Explore the idea of *ternary heaps*, based on complete ternary trees instead of binary trees. Do ternary heaps sort faster than binary heaps?

29. [*26*] (W. S. Brown.) Design an algorithm for multiplication of polynomials or power series $(a_1x^{i_1} + a_2x^{i_2} + \cdots)(b_1x^{j_1} + b_2x^{j_2} + \cdots)$, in which the coefficients of the answer $c_1x^{i_1+j_1} + \cdots$ are generated in order as the input coefficients are being multiplied. [*Hint:* Use an appropriate priority queue.]

30. [*M48*] Can C ever exceed $\frac{1}{2}N \log_2 N$ when a file is heapsorted? What is the maximum value of C? What is the minimum value?

31. [*37*] (J. W. J. Williams.) Show that if two heaps are placed "back to back" in a suitable way, it is possible to maintain a structure in which either the smallest or the largest element can be deleted at any time in $O(\log n)$ steps. (Such a structure may be called a *priority deque*.)

32. [*21*] Design an algorithm which merges two disjoint priority queues, represented as leftist trees, into one. (In particular, if one of the given queues contains a single element, your algorithm will insert it into the other queue.)

▸**33.** [*15*] In a priority queue represented as a leftist tree, why is the operation of removing the root done by merging the two subtrees, instead of by "promoting" nodes toward the top as in a heap?

34. [*M47*] How many leftist trees with N nodes are possible, ignoring the KEY values? [The sequence begins 1, 1, 2, 4, 8, 17, 38, . . . ; is there a simple asymptotic formula?]

35. [*26*] If UP links are added to a leftist tree (cf. the discussion of triply-linked trees in Section 6.2.3), it is possible to delete an arbitrary node P from within the priority queue as follows: Replace P by the merger of LEFT(P) and RIGHT(P); then adjust the DIST fields of P's ancestors until either reaching the root or reaching a node whose DIST is unchanged.

Prove that this process never requires changing more than $O(\log N)$ of the DIST fields, if there are N nodes in the tree, even though the tree may contain very long upward paths.

36. [*18*] (*Least-recently-used page replacement.*) Many operating systems make use of the following type of algorithm: A collection of nodes is subjected to two operations, "using" a node, and replacing the least-recently-"used" node by a new node. What data structure makes it easy to ascertain the least-recently-"used" node?

5.2.4. Sorting by Merging

Merging (or *collating*) means the combination of two or more ordered files into a single ordered file. For example, we can merge the two files 503 703 765 and 087 512 677 to obtain 087 503 512 677 703 765. A simple way to accomplish this is to compare the two smallest items, output the smallest, and then repeat the same process. Starting with

$$\begin{cases} 503 & 703 & 765 \\ 087 & 512 & 677 \end{cases}$$

we obtain

$$087 \begin{cases} 503 & 703 & 765 \\ 512 & 677 & \end{cases},$$

then

$$087 \quad 503 \begin{cases} 703 & 765 \\ 512 & 677 \end{cases},$$

and so on. Some care is necessary when one of the two files becomes exhausted; a detailed description of the process appears in the following algorithm:

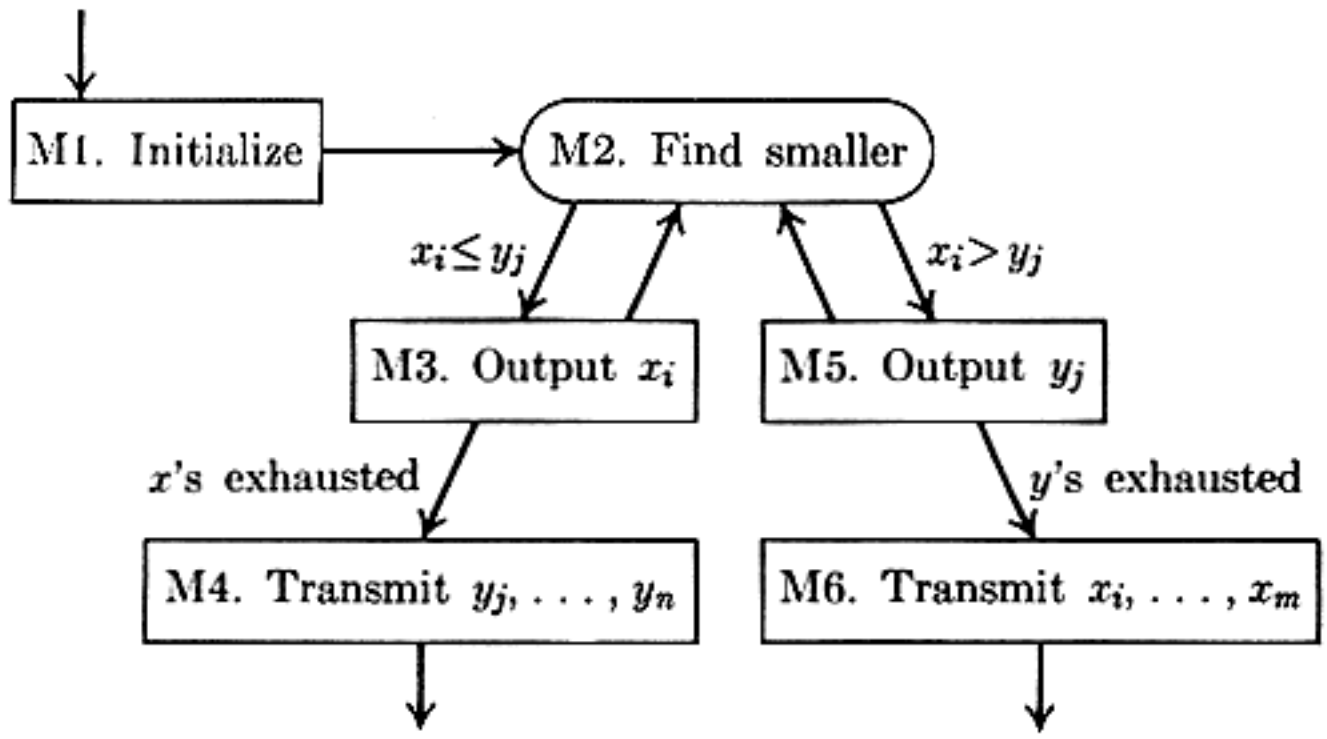


Fig. 29. Merging $x_1 \le \cdots \le x_m$ with $y_1 \le \cdots \le y_n$.

Algorithm M (*Two-way merge*). This algorithm merges the ordered files $x_1 \le x_2 \le \cdots \le x_m$ and $y_1 \le y_2 \le \cdots \le y_n$ into a single file $z_1 \le z_2 \le \cdots \le z_{m+n}$.

M1. [Initialize.] Set $i \leftarrow 1$, $j \leftarrow 1$, $k \leftarrow 1$.

M2. [Find smaller.] If $x_i \le y_j$, go to step M3, otherwise go to M5.

M3. [Output x_i.] Set $z_k \leftarrow x_i$, $k \leftarrow k + 1$, $i \leftarrow i + 1$. If $i \le m$, return to M2.

M4. [Transmit $y_j, \ldots, y_n$.] Set $(z_k, \ldots, z_{m+n}) \leftarrow (y_j, \ldots, y_n)$ and terminate the algorithm.

M5. [Output y_j.] Set $z_k \leftarrow y_j$, $k \leftarrow k + 1$, $j \leftarrow j + 1$. If $j \le n$, return to M2.

M6. [Transmit $x_i, \ldots, x_m$.] Set $(z_k, \ldots, z_{m+n}) \leftarrow (x_i, \ldots, x_m)$ and terminate the algorithm. ∎

We shall see in Section 5.3.2 that this straightforward procedure is essentially the "best possible" way to merge on a conventional computer, when $m \approx n$. (On the other hand, when m is much smaller than n, it is possible to devise more efficient merging algorithms, although they are rather complicated in general.) Algorithm M could be made slightly simpler without much loss of efficiency by placing artificial "sentinel" elements $x_{m+1} = y_{n+1} = \infty$ at the end of the input files, stopping just before ∞ is output. For an analysis of Algorithm M, see exercise 2.

The total amount of work involved in Algorithm M is essentially proportional to $m + n$, so it is clear that merging is a simpler problem than sorting. Furthermore, we can reduce the problem of sorting to merging, for it is possible to continue merging longer and longer subfiles until everything is in sort. We may consider this to be an extension of the idea of insertion sorting: inserting a new element into a sorted file is the special case $n = 1$ of merging! If we want to speed up the insertion process we can consider inserting several elements at a time, "batching" them, and this leads naturally to the general idea of merge sorting. From a historical point of view, merge sorting is one of the very first

methods proposed for computer sorting; it was suggested by John von Neumann as early as 1945 (see Section 5.5).

We shall study merging in considerable detail in Section 5.4, with regard to external sorting algorithms; in the present section, we shall focus only on the question of merge sorting within a high-speed random-access memory.

Table 1 shows a merge sort which "burns the candle at both ends" in a manner similar to the scanning procedure we have used in quicksort, radix exchange, etc.: we examine the input from the left and from the right, working towards the middle. Ignoring the top line of the table for a moment, let us consider the transformation from line 2 to line 3. At the left we have the ascending run 503 703 765; at the right, reading leftwards, we have the run 087 512 677. Merging these two sequences leads to 087 503 512 677 703 765, which is placed at the left of line 3. Then the keys 061 612 908 in line 2 are merged with 170 509 897, and the result (061 170 509 612 897 908) is recorded at the *right* end of line 3. Finally, 154 275 426 653 is merged with 653 (the overlap is discovered before it causes any harm), and the result is placed at the left. Line 2 of the table was formed in the same way from the original input in line 1.

Table 1

NATURAL TWO-WAY MERGE SORTING

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	765	061	612	908	154	275	426	653	897	509	170	677	512	087
087	503	512	677	703	765	154	275	426	653	908	897	612	509	170	061
061	087	170	503	509	512	612	677	703	765	897	908	653	426	275	154
061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

Vertical lines in Table 1 represent the boundaries between runs. They are the so-called "step-downs," where a smaller element follows a larger one. In the middle of the file we generally have an ambiguous situation where we read the same key from both directions; this causes no problem if we are a little bit careful as in the following algorithm. The method is traditionally called a "natural" merge because it makes use of the runs which occur "naturally" in its input.

Algorithm N (*Natural two-way merge sort*). Records $R_1, \ldots, R_N$ are sorted using two areas of memory, each of which is capable of holding N records. For convenience, we shall say that the records of the second area are R_{N+1}, $\ldots$, R_{2N}, although it is not really necessary that R_{N+1} be adjacent to R_N. The initial contents of $R_{N+1}, \ldots, R_{2N}$ is immaterial. After sorting is complete, the keys will be in order, $K_1 \leq \cdots \leq K_N$.

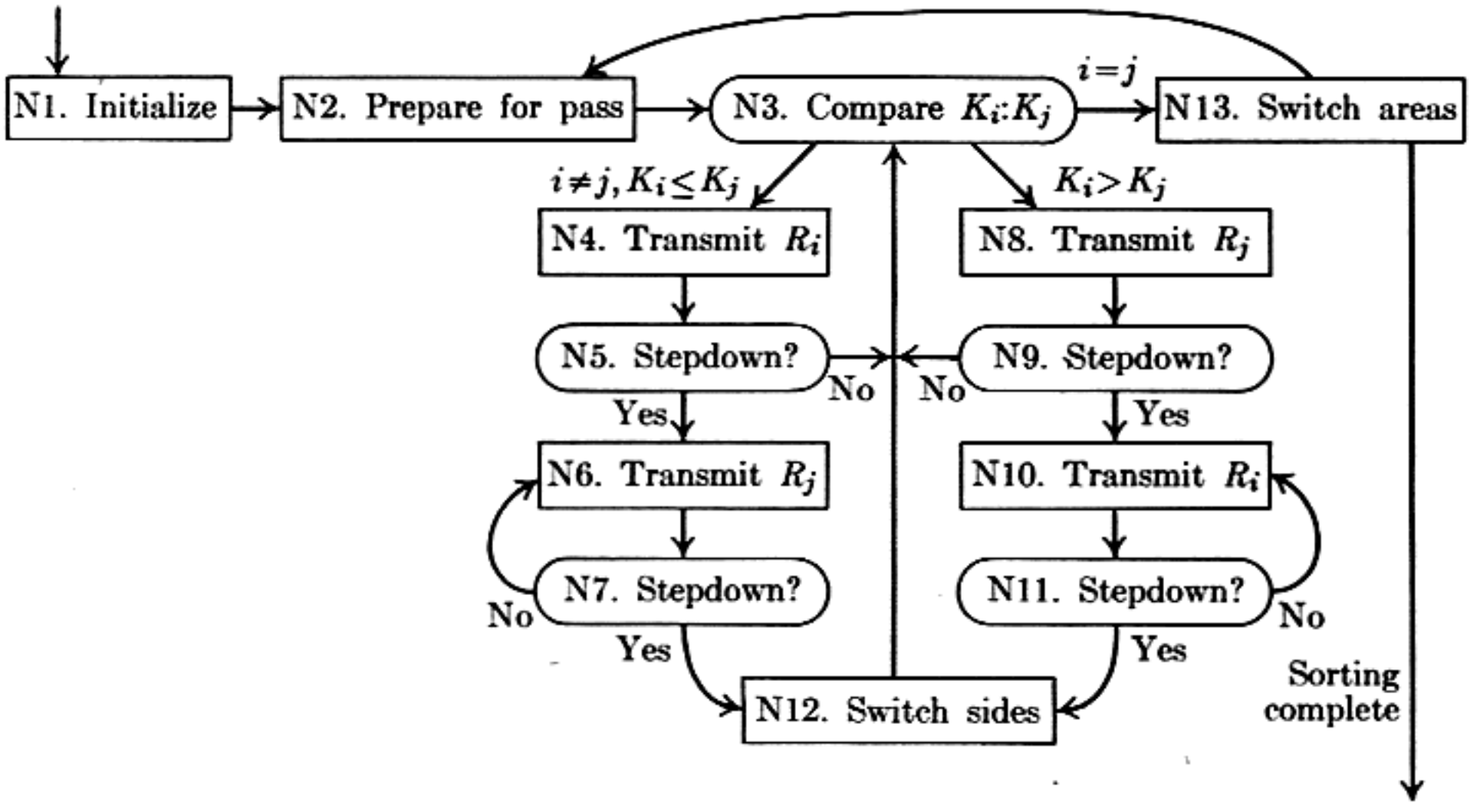


Fig. 30. Merge sorting.

N1. [Initialize.] Set $s \leftarrow 0$. (When $s = 0$, we will be transferring records from the $(R_1, \ldots, R_N)$ area to the $(R_{N+1}, \ldots, R_{2N})$ area; when $s = 1$, we will be going the other way.)

N2. [Prepare for pass.] If $s = 0$, set $i \leftarrow 1$, $j \leftarrow N$, $k \leftarrow N + 1$, $l \leftarrow 2N$; if $s = 1$, set $i \leftarrow N + 1$, $j \leftarrow 2N$, $k \leftarrow 1$, $l \leftarrow N$. (Variables i, j, k, l point to the current positions in the "source files" being read and the "destination files" being written.) Set $d \leftarrow 1$, $f \leftarrow 1$. (Variable d gives the current direction of output; f is set to zero if future passes are necessary.)

N3. [Compare K_i:K_j.] If $K_i > K_j$, go to step N8. If $i = j$, set $R_k \leftarrow R_i$ and go to N13.

N4. [Transmit R_i.] (Steps N4–N7 are analogous to steps M3–M4 of Algorithm M.) Set $R_k \leftarrow R_i$, $k \leftarrow k + d$.

N5. [Stepdown?] Increase i by 1. Then if $K_{i-1} \leq K_i$, go back to step N3.

N6. [Transmit R_j.] Set $R_k \leftarrow R_j$, $k \leftarrow k + d$.

N7. [Stepdown?] Decrease j by 1. Then if $K_{j+1} \leq K_j$, go back to step N6; otherwise go to step N12.

N8. [Transmit R_j.] (Steps N8–N11 are dual to steps N4–N7.) Set $R_k \leftarrow R_j$, $k \leftarrow k + d$.

N9. [Stepdown?] Decrease j by 1. Then if $K_{j+1} \leq K_j$, go back to step N3.

N10. [Transmit R_i.] Set $R_k \leftarrow R_i$, $k \leftarrow k + d$.

N11. [Stepdown?] Increase i by 1. Then if $K_{i-1} \leq K_i$, go back to step N10.

N12. [Switch sides.] Set $f \leftarrow 0$, $d \leftarrow -d$, and interchange $k \leftrightarrow l$. Return to step N3.

N13. [Switch areas.] If $f = 0$, set $s \leftarrow 1 - s$ and return to N2. Otherwise sorting is complete; if $s = 0$, set $(R_1, \ldots, R_N) \leftarrow (R_{N+1}, \ldots, R_{2N})$. (This last copying operation is unnecessary if it is acceptable to have the output in $(R_{N+1}, \ldots, R_{2N})$ at times.) ∎

This algorithm contains one tricky feature which is explained in exercise 5.

It would not be difficult to program Algorithm N for MIX, but we can deduce the essential facts of its behavior without constructing the entire program. The number of ascending runs in the input will be about $\frac{1}{2}N$, under random conditions, since we have $K_i > K_{i+1}$ with probability $\frac{1}{2}$; detailed information about the number of runs, under slightly different hypotheses, has been derived in Section 5.1.3. Each pass cuts the number of runs in half (except in unusual cases such as the situation in exercise 6). So the number of passes will usually be about $\log_2 N$. Each pass requires us to transmit each of the N records, and by exercise 2 most of the time is spent in steps N3, N4, N5, N8, N9. We can sketch the time in the inner loop as follows, if we assume that there is low probability of equal keys:

	Step	Operations	Time
	N3	CMPA, JG, JE	$3.5u$
Either	N4	STA, INC	$3u$
	N5	INC, LDA, CMPA, JGE	$6u$
Or	N8	STX, INC	$3u$
	N9	DEC, LDX, CMPX, JGE	$6u$

Thus about $12.5u$ is spent on each record in each pass, and the total running time will be asymptotically $12.5N \log_2 N$, for both the average case and the worst case. This is slower than quicksort's average time, and it may not be enough better than heapsort to justify taking twice as much memory space, since the asymptotic running time of Program 5.2.3H is $16N \log_2 N$.

The boundary lines between runs are determined in Algorithm N entirely by "step-downs." This has the possible advantage that input files with a preponderance of increasing *or decreasing* order can be handled very quickly; but it slows down the main loop of the calculation. Instead of testing step-downs, we can determine the length of runs artifically, by saying that all runs in the input have length 1, all runs after the first pass (except possibly the last run) have length 2, . . . , all runs after k passes (except possibly the last run) have length 2^k. This is called a *straight* two-way merge, as opposed to the "natural" merge in Algorithm N.

Straight two-way merging is very similar to Algorithm N, and it has essentially the same flowchart; but things are just different enough to make it worth while to write down the whole algorithm again:

Algorithm S (*Straight two-way merge sort*). Records $R_1, \ldots, R_N$ are sorted using two memory areas as in Algorithm N.

S1. [Initialize.] Set $s \leftarrow 0$, $p \leftarrow 1$. (For the significance of variables s, i, j, k, l, d, see Algorithm N. Here p represents the size of ascending runs to be merged on the current pass; q and r keep track of the number of unmerged items in a run.)

S2. [Prepare for pass.] If $s = 0$, set $i \leftarrow 1$, $j \leftarrow N$, $k \leftarrow N$, $l \leftarrow 2N + 1$; if $s = 1$, set $i \leftarrow N + 1$, $j \leftarrow 2N$, $k \leftarrow 0$, $l \leftarrow N + 1$. Then set $d \leftarrow 1$, $q \leftarrow p$, $r \leftarrow p$.

S3. [Compare K_i:K_j.] If $K_i > K_j$, go to step S8.

S4. [Transmit R_i.] Set $k \leftarrow k + d$, $R_k \leftarrow R_i$.

S5. [End of run?] Set $i \leftarrow i + 1$, $q \leftarrow q - 1$. If $q > 0$, go back to step S3.

S6. [Transmit R_j.] Set $k \leftarrow k + d$. Then if $k = l$, go to step S13; otherwise set $R_k \leftarrow R_j$.

S7. [End of run?] Set $j \leftarrow j - 1$, $r \leftarrow r - 1$. If $r > 0$, go back to step S6; otherwise go to S12.

S8. [Transmit R_j.] Set $k \leftarrow k + d$, $R_k \leftarrow R_j$.

S9. [End of run?] Set $j \leftarrow j - 1$, $r \leftarrow r - 1$. If $r > 0$, go back to step S3.

S10. [Transmit R_i.] Set $k \leftarrow k + d$. Then if $k = l$, go to step S13; otherwise set $R_k \leftarrow R_i$.

S11. [End of run?] Set $i \leftarrow i + 1$, $q \leftarrow q - 1$. If $q > 0$, go back to step S10.

S12. [Switch sides.] Set $q \leftarrow p$, $r \leftarrow p$, $d \leftarrow -d$, and interchange $k \leftrightarrow l$. Return to step S3.

S13. [Switch areas.] Set $p \leftarrow p + p$. If $p < N$, set $s \leftarrow 1 - s$ and return to S2. Otherwise sorting is complete; if $s = 0$, set

$$(R_1, \ldots, R_N) \leftarrow (R_{N+1}, \ldots, R_{2N}).$$

(The latter copying operation will be done if and only if $\lceil \log_2 N \rceil$ is odd, regardless of the distribution of the input. Therefore it is possible to predict the location of the sorted output in advance, and copying will usually be unnecessary.) ∎

For an example of this algorithm, see Table 2. It is somewhat amazing that the method works properly when N is not a power of 2; the runs being merged are not all of length 2^k, yet no provision has apparently been made for the exceptions! (See exercise 8.) The former tests for step-downs have been replaced by decrementing q or r and testing the result for zero; this reduces the asymptotic MIX running time to $11N \log_2 N$ units, slightly faster than we were able to achieve with Algorithm N.

Table 2

STRAIGHT TWO-WAY MERGE SORTING

503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
503	703	512	677	509	908	426	897	653	275	170	154	612	061	765	087
087	503	703	765	154	170	509	908	897	653	426	275	677	612	512	061
061	087	503	512	612	677	703	765	908	897	653	509	426	275	170	154
061	087	154	179	275	426	503	509	512	612	653	677	703	765	897	908

In practice it would be worth while to combine Algorithm S with straight insertion; we can sort groups of, say, 16 items using straight insertion, in place of the first four passes of Algorithm S, thereby avoiding the comparatively wasteful bookkeeping operations involved in short merges. As we saw with quicksort, such a combination of methods does not affect the asymptotic running time, but it gives us a reasonable improvement nevertheless.

Let us now study Algorithms N and S from the standpoint of data structures. Why did we need $2N$ record locations instead of N? The reason is comparatively simple: We are dealing with four lists of varying size (two "source" lists and two "destination" lists on each pass); and we are using the standard "growing together" idea discussed in Section 2.2.2, for each pair of sequentially allocated lists. But half of the memory space is always unused, and a little reflection shows that we should really make use of a *linked* allocation for the four lists. If we add one link field to each of the N records, we can do everything required by the merging algorithms using simple link manipulations without moving the records at all! Adding N link fields is generally better than adding the space needed for N more records, and the reduced record movement may also save us time. Therefore we ought to consider a merging algorithm like the following one:

Algorithm L (*List merge sort*). Records $R_1, \ldots, R_N$ are assumed to contain keys $K_1, \ldots, K_N$, and "link fields" $L_1, \ldots, L_N$ capable of holding the numbers $-(N+1)$ through $(N+1)$. There are two auxiliary link fields L_0 and L_{N+1} in artificial records R_0 and R_{N+1} at the beginning and end of the file. This algorithm is a "list sort" which sets the link fields so that the records are linked together in ascending order. After sorting is complete, L_0 is the index of the record with the smallest key; and for $1 \le k \le N$, L_k is the index of the record which follows R_k, or $L_k = 0$ if R_k is the record with the largest key. (See Eq. 5.2.1–9.)

During the course of this algorithm, R_0 and R_{N+1} serve as "list heads" for two linear lists whose sublists are being merged. A negative link denotes the end of a sublist known to be ordered; a zero link denotes the end of the entire list. We assume that $N \ge 2$.

The notation "$|L_s| \leftarrow p$" means "Set L_s to p or $-p$, retaining the previous sign of L_s." This operation is well-suited to MIX, but unfortunately not to most computers; it is possible to modify the algorithm in straightforward ways to obtain an equally efficient method for most other machines.

L1. [Prepare two lists.] Set $L_0 \leftarrow 1$, $L_{N+1} \leftarrow 2$, $L_i \leftarrow -(i+2)$ for $1 \leq i \leq N-2$, and $L_{N-1} \leftarrow L_N \leftarrow 0$. (We have created two lists containing R_1, R_3, $R_5, \ldots$ and R_2, R_4, $R_6, \ldots$, respectively; the negative links indicate that each ordered "sublist" consists of one element only. For another way to do this step, taking advantage of ordering which may be present in the initial data, see exercise 12.)

L2. [Begin new pass.] Set $s \leftarrow 0$, $t \leftarrow N+1$, $p \leftarrow L_s$, $q \leftarrow L_t$. If $q = 0$, the algorithm terminates. (During each pass, p and q traverse the lists being merged; s usually points to the most recently processed record of the current sublist, while t points to the end of the previously output sublist.)

L3. [Compare K_p:K_q.] If $K_p > K_q$, go to L6.

L4. [Advance p.] Set $|L_s| \leftarrow p$, $s \leftarrow p$, $p \leftarrow L_p$. If $p > 0$, return to L3.

L5. [Complete the sublist.] Set $L_s \leftarrow q$, $s \leftarrow t$. Then set $t \leftarrow q$ and $q \leftarrow L_q$, one or more times, until $q \leq 0$. Finally go to L8.

L6. [Advance q.] (Steps L6 and L7 are dual to L4 and L5.) Set $|L_s| \leftarrow q$, $s \leftarrow q$, $q \leftarrow L_q$. If $q > 0$, return to L3.

L7. [Complete the sublist.] Set $L_s \leftarrow p$, $s \leftarrow t$. Then set $t \leftarrow p$ and $p \leftarrow L_p$, one or more times, until $p \leq 0$.

L8. [End of pass?] (At this point, $p \leq 0$ and $q \leq 0$, since both pointers have moved to the end of their respective sublists.) Set $p \leftarrow -p$, $q \leftarrow -q$. If $q = 0$, set $|L_s| \leftarrow p$, $|L_t| \leftarrow 0$ and return to L2. Otherwise return to L3. ∎

An example of this algorithm in action appears in Table 3, where we can see the link settings each time step L2 is encountered. It is possible to rearrange the records $R_1, \ldots, R_N$ at the end of this algorithm, so that their keys are in order, using the method of exercise 5.2–12. There is an interesting similarity between list merging and the addition of sparse polynomials (see Algorithm 2.2.4A).

Table 3

LIST MERGE SORTING

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
K_j	—	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703	—
L_j	1	−3	−4	−5	−6	−7	−8	−9	−10	−11	−12	−13	−14	−15	−16	0	0	2
L_j	2	−6	1	−8	3	−10	5	−11	7	−13	9	12	−16	14	0	0	15	4
L_j	4	3	1	−11	2	−13	8	5	7	0	12	10	9	14	16	0	15	6
L_j	4	3	6	7	2	0	8	5	1	14	12	10	13	9	16	0	15	11
L_j	4	12	11	13	2	0	8	5	10	14	1	6	3	9	16	7	15	0

Let us now construct a MIX program for Algorithm L, to see whether the list manipulation is advantageous from the standpoint of speed as well as space:

Program L (*List merge sort*). For convenience, we assume that records are one word long, with L_j in the (0:2) field and K_j in the (3:5) field of location INPUT+j; rI1 $\equiv p$, rI2 $\equiv q$, rI3 $\equiv s$, rI4 $\equiv t$, rA $\equiv R_q$. $N \geq 2$.

```
01    L       EQU   0:2                                  Definition of field names
02    ABSL    EQU   1:2
03    KEY     EQU   3:5
04    START   ENT1  N-2                        1         L1. Prepare two lists.
05            ENNA  2,1                        N - 2
06            STA   INPUT,1(L)                 N - 2     L_i ← -(i+2).
07            DEC1  1                          N - 2
08            J1P   *-3                        N - 2     N - 2 ≥ i > 0.
09            ENTA  1                          1
10            STA   INPUT(L)                   1         L_0 ← 1.
11            ENTA  2                          1
12            STA   INPUT+N+1(L)               1         L_{N+1} ← 2.
13            STZ   INPUT+N-1(L)               1         L_{N-1} ← 0.
14            STZ   INPUT+N(L)                 1         L_N ← 0.
15            JMP   L2                         1         To L2. (See next page.)
16    L3Q     LDA   INPUT,2                    C'' + B'  L3. Compare K_p:K_q.
17    L3P     CMPA  INPUT,1(KEY)               C
18            JL    L6                         C         To L6 if K_q < K_p.
19    L4      ST1   INPUT,3(ABSL)              C'        L4. Advance p. |L_s| ← p.
20            ENT3  0,1                        C'        s ← p.
21            LD1   INPUT,1(L)                 C'        p ← L_q.
22            J1P   L3P                        C'        To L3 if p > 0.
23    L5      ST2   INPUT,3(L)                 B'        L5. Complete the sublist. L_s ← q.
24            ENT3  0,4                        B'        s ← t.
25            ENT4  0,2                        D'        t ← q.
26            LD2   INPUT,2(L)                 D'        q ← L_q.
27            J2P   *-2                        D'        Repeat if q > 0.
28            JMP   L8                         B'        To L8.
29    L6      ST2   INPUT,3(ABSL)              C''       L6. Advance q. |L_s| ← q.
30            ENT3  0,2                        C''       s ← q.
31            LD2   INPUT,2(L)                 C''       q ← L_q.
32            J2P   L3Q                        C''       To L3 if q > 0.
33    L7      ST1   INPUT,3(L)                 B''       L7. Complete the sublist. L_s ← p.
34            ENT3  0,4                        B''       s ← t.
35            ENT4  0,1                        D''       t ← p.
36            LD1   INPUT,1(L)                 D''       p ← L_p.
37            J1P   *-2                        D''       Repeat if p > 0.
38    L8      ENN1  0,1                        B         L8. End of pass? p ← -p.
39            ENN2  0,2                        B         q ← -q.
40            J2NZ  L3Q                        B         To L3 if q ≠ 0.
41            ST1   INPUT,3(ABSL)              A         |L_s| ← p.
42            STZ   INPUT,4(ABSL)              A         |L_t| ← 0.
```

```
43    L2    ENT3  0                 A+1    L2. Begin new pass.  s ← 0.
44          ENT4  N+1               A+1    t ← N+1.
45          LD1   INPUT(L)          A+1    p ← L_s.
46          LD2   INPUT+N+1(L)      A+1    q ← L_t.
47          J2NZ  L3Q               A+1    To L3 if q ≠ 0. ∎
```

The running time of this program can be deduced using techniques we have seen many times before (see exercises 13, 14); it comes to approximately $(10N \log_2 N + 4.92N)u$ on the average, with a small standard deviation of order $\sqrt{N}$. Exercise 15 shows that the running time can be reduced to about $9N \log_2 N$ at the expense of a somewhat longer program.

Thus we have a clear victory for linked-memory techniques over sequential allocation, when internal merging is being done: Less memory space is required, and the program runs about 10 to 20 percent faster. Similar algorithms have been published by L. J. Woodrum [*IBM Systems J.* **8** (1969), 189–203] and A. D. Woodall [*Comp. J.* **13** (1970), 110–111].

EXERCISES

1. [*20*] Generalize Algorithm M to a *k-way merge* of the input files $x_{i1} \le \cdots \le x_{im_i}$ for $i = 1, 2, \ldots, k$.

2. [*M24*] Assuming that each of the $\binom{m+n}{m}$ possible arrangement of m x's among n y's is equally likely, find the mean and standard deviation of the number of times step M2 is performed during Algorithm M. What are the maximum and minimum values of this quantity?

▸ **3.** [*20*] (*Updating.*) Given records $R_1, \ldots, R_M$ and $R'_1, \ldots, R'_N$ whose keys are distinct and in order, so that $K_1 < \cdots < K_M$ and $K'_1 < \cdots < K'_N$, show how to modify Algorithm M to obtain a merged file in which records R_i of the first file have been *discarded* if their key appears also in the second file.

4. [*21*] The text observes that merge sorting may be regarded as a generalization of insertion sorting. Show that merge sorting is also strongly related to tree selection sorting as depicted in Fig. 23.

▸ **5.** [*21*] Prove that i can never be equal to j in steps N6 or N10. (Therefore it is unnecessary to test for a possible jump to N13 in those steps.)

6. [*22*] Find a permutation $K_1 K_2 \ldots K_{16}$ of $\{1, 2, \ldots, 16\}$ such that

$$K_2 > K_3, \quad K_4 > K_5, \quad K_6 > K_7, \quad K_8 > K_9,$$
$$K_{10} < K_{11}, \quad K_{12} < K_{13}, \quad K_{14} < K_{15},$$

yet Algorithm N will sort the file in only two passes. (Since there are eight or more runs, we would expect to have at least four runs after the first pass, two runs after the second pass, and sorting would ordinarily not be complete until after at least three passes. How can we get by with only two passes?)

7. [*16*] Give a formula for the exact number of passes required by Algorithm S, as a function of N.

8. [*22*] During Algorithm S, the variables q and r are supposed to represent the lengths of the unmerged elements in the runs currently being processed; q and r both start out equal to p, while the runs are not always this long. How can this possibly work?

9. [*24*] Write a MIX program for Algorithm S. Specify the instruction frequencies in terms of quantities analogous to A, B', B'', C', ... in Program L.

10. [*25*] (D. A. Bell.) Show that sequentially-allocated straight two-way merging can be done with at most $\frac{3}{2}N$ memory locations, instead of $2N$ as in Algorithm S.

11. [*21*] Is Algorithm L a "stable" sorting method?

▸**12.** [*22*] Revise step L1 of Algorithm L so that the two-way merge is "natural," taking advantage of ascending runs which are initially present. (In particular, if the input is already sorted, step L2 should terminate the algorithm immediately after your step L1 has acted.)

▸**13.** [*M34*] Give an analysis of the average running time of Program L, in the style of other analyses in this chapter: Interpret the quantities A, B, B', ..., and explain how to compute their exact average values. How long does Program L take to sort the 16 numbers in Table 3?

14. [*M24*] Let the binary representation of N be $2^{e_1} + 2^{e_2} + \cdots + 2^{e_t}$, where $e_1 > e_2 > \cdots > e_t \geq 0$, $t \geq 1$. Prove that the maximum number of key comparisons performed by Algorithm L is $1 - 2^{e_t} + \sum_{1 \leq k \leq t} (e_k + k - 1)2^{e_k}$.

15. [*20*] Hand simulation of Algorithm L reveals that it occasionally does redundant operations; the assignments $|L_s| \leftarrow p$, $|L_s| \leftarrow q$ in steps L4 and L6 are unnecessary about half of the time, since we have $L_s = p$ (or q) each time step L4 (or L6) returns to L3. How can Program L be improved so that this redundancy disappears?

16. [*28*] Design a list merging algorithm like Algorithm L but based on three-way merging.

17. [*20*] (J. McCarthy.) Let the binary representation of N be as in exercise 14, and assume that we are given N records arranged in t ordered subfiles of respective sizes $2^{e_1}, 2^{e_2}, \ldots, 2^{e_t}$. Show how to maintain this state of affairs when a new $(N+1)$st record is added and $N \leftarrow N + 1$. (The resulting algorithm may be called an "on-line" merge sort.)

18. [*40*] (M. A. Kronrod.) Given a file on N records containing only two runs,

$$K_1 \leq \cdots \leq K_M \quad \text{and} \quad K_{M+1} \leq \cdots \leq K_N,$$

is it possible to sort the file in $O(N)$ operations in a random-access memory, *using only a small fixed amount of additional memory space* regardless of the size of M and N? (All of the merging algorithms described in this section make use of extra memory space proportional to N.)

19. [*26*] Consider a railway switching network with n "stacks", as shown in Fig. 31 when $n = 5$; such a network has some aspects relevant to n-pass sorting algorithms. We considered one-stack networks in exercises 2.2.1–2 through 2.2.1–5. If N railroad cars enter at the right, we observed that only comparatively few of the $N!$ permutations of those cars could appear at the left, in the one-stack case.

In the n-stack network, assume that 2^n cars enter at the right. Prove that each of the $2^n!$ possible permutations of these cars *is* achievable at the left, by a suitable

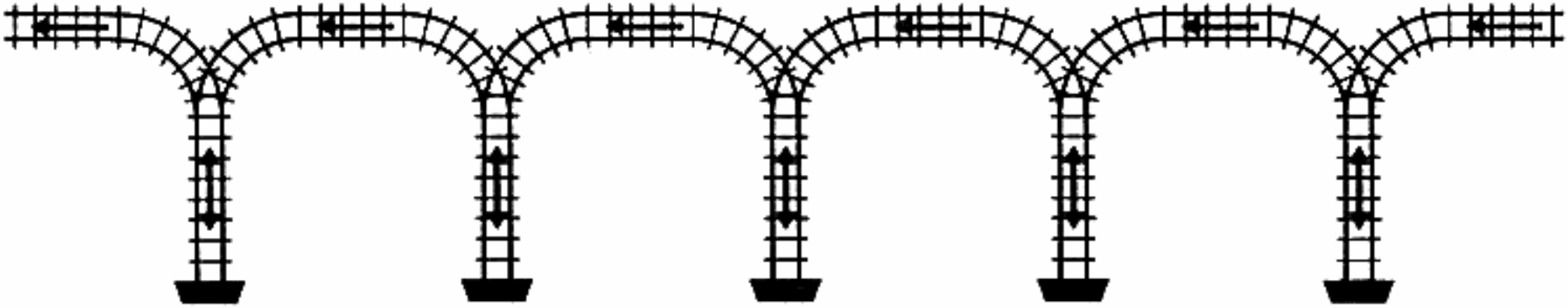


Fig. 31. A railway network with five "stacks."

sequence of operations. (Each stack is big enough to accommodate all the cars, if necessary.)

20. [*47*] In the notation of exercise 2.2.1–4, at most a_N^n permutations of N elements can be produced with an n-stack railway network; hence the number of stacks needed to obtain all $N!$ permutations is at least $\log N!/\log a_N \approx \log_4 N$. Exercise 19 shows that at most $\lceil \log_2 N \rceil$ stacks are needed. What is the true rate of growth of the necessary number of stacks, as $N \to \infty$?

21. [*23*] (A. J. Smith.) Explain how to extend Algorithm L so that, in addition to sorting, it computes the number of *inversions* present in the input permutation.

5.2.5. Sorting by Distribution

We come now to an interesting class of sorting methods, which is shown in Section 5.4.7 to be essentially the exact *opposite* of merging. Readers who are familiar with punched card equipment are well aware of the efficient procedure used on card sorters, based on the digits of the keys; the same idea can be adapted to computer programming, and it is generally known as "radix sorting," "digital sorting," or "pocket sorting."

Suppose we want to sort a 52-card deck of playing cards. We may define

$$\texttt{A} < \texttt{2} < \texttt{3} < \texttt{4} < \texttt{5} < \texttt{6} < \texttt{7} < \texttt{8} < \texttt{9} < \texttt{10} < \texttt{J} < \texttt{Q} < \texttt{K},$$

as an ordering of the face values, and for the suits we may define

$$\clubsuit < \diamondsuit < \heartsuit < \spadesuit$$

One card is to precede another if either (i) its suit is less than the other suit, or (ii) the suits are equal but its face value is less. (This is a particular case of a *lexicographic ordering* between ordered pairs of objects; cf. exercise 5–2). Thus

$$\texttt{A}\clubsuit < \texttt{2}\clubsuit < \cdots < \texttt{K}\clubsuit < \texttt{A}\diamondsuit < \cdots < \texttt{Q}\spadesuit < \texttt{K}\spadesuit.$$

We could sort the cards by any of the methods already discussed; people generally use a technique somewhat analogous to the idea behind radix exchange. It is natural to sort the cards first by their suits into four piles, then to fiddle with each of the individual piles until they are in order.

But there is a faster way to do the trick! First deal the cards face up into 13 piles, one for each face value. Then collect these piles by putting the aces

on the bottom, the 2's face up on top of them, then the 3's, etc., finally putting the kings (face up) on top. Turn the deck face down and deal again, this time into four piles for the four suits. By putting the resulting piles together, with clubs on the bottom, then diamonds, hearts, and spades, the deck will be in perfect order.

The same idea applies to the sorting of numbers and alphabetic data. Why does it work? Because (in our playing card example) if two cards go into different piles in the final deal, they have different suits, so the one with the lower suit is lowest. But if two cards have the same suit (and consequently go into the same pile), they are already in proper order because of the previous sorting. In other words, the face values will be in increasing order on each of the four piles as we deal the cards on the second pass. The same proof can be abstracted to show that any lexicographic ordering can be sorted in this manner; for details, see the answer to exercise 5–2, at the beginning of this chapter.

The sorting method just described is not immediately obvious, and it isn't clear who first discovered the fact that it works so conveniently. A 19-page pamphlet entitled "The Inventory Simplified," published by the Tabulating Machines Company division of IBM in 1923, presents an interesting Digit Plan method for forming sums of products on their Electric Sorting Machine: Suppose, for example, that we want to multiply the number punched in columns 1–10 by the number punched in columns 23–25, and to sum all of these products for a large number of cards. We can sort first on column 25, then use the Tabulating Machine to find the quantities $a_1, a_2, \ldots, a_9$, where a_k is the total of columns 1–10 summed over all cards having k in column 25. Then we can sort on column 24, finding the analogous totals $b_1, b_2, \ldots, b_9$; also on column 23, obtaining $c_1, c_2, \ldots, c_9$. The desired sum of products is easily seen to be

$$a_1 + 2a_2 + \cdots + 9a_9 + 10b_1 + 20b_2 + \cdots + 90b_9 + 100c_1 + 200c_2 + \cdots + 900c_9.$$

This punched-card tabulating method leads naturally to the discovery of least-significant-digit-first radix sorting, so it probably became known to the operators of the machines. The first published reference to this principle for sorting appears in L. J. Comrie's early discussion of punched-card equipment [*Transactions of the Office Machinery Users' Assoc., Ltd.* (1929), 25–37, esp. p. 28].

In order to handle radix sorting inside a computer, we must decide what to do with the piles. Suppose that there are M piles; we could set aside M areas of memory, moving each record from an input area into its appropriate pile area. But this is unsatisfactory, since each area must be large enough to hold N items, and $(M + 1)N$ record spaces would be required. Such an excessive memory requirement originally caused most people to reject the idea of radix sorting within a computer, until H. H. Seward [Master's thesis, M.I.T. Digital Computer Laboratory Report R-232 (Cambridge, Mass.: 1954), 25–28] pointed out that we can achieve the same effect with only $2N$ record areas and M count

fields. We simply count how many elements will lie in each of the M piles, by making a preliminary pass over the data; this tells us precisely how to allocate memory for the piles. We have already made use of the same idea in the "distribution sort," Algorithm 5.2D.

Thus radix sorting can be carried out as follows: Start with a distribution sort based on the *least significant digit* of the keys (in radix M notation), moving records from the input area to an auxiliary area. Then do another distribution sort, on the next least significant digit, moving the records back into the original input area; and so on, until the final pass (on the most significant digit) puts all records into the desired order.

If we have a decimal computer with 12-digit keys, and if N is rather large, we can choose $M = 1000$ (considering three decimal digits as one radix-1000 digit); sorting will be complete in four passes, regardless of the size of N. Similarly, if we have a binary computer and a 40-bit key, we can set $M = 1024$ and complete the sorting in four passes. Actually each "pass" consists of three parts (counting, allocating, moving); Friend [*JACM* **3** (1956), 151] has suggested combining two of these parts at the expense of M more memory locations, by accumulating the counts for pass $k + 1$ while moving the records in pass k.

Table 1 shows how such a radix sort can be applied to our 16 example numbers, with $M = 10$. Radix sorting is generally not useful for such small N, so a small example like this is intended to illustrate the sufficiency rather than the efficiency of the method.

Table 1

RADIX SORTING

Input area contents:	503	087	512	061	908	170	897	275	653	426	154	509	612	677	765	703
Counts for units digit distribution:					1	1	2	3	1	2	1	3	1	1		
Storage allocations based on these counts:					1	2	4	7	8	10	11	14	15	16		
Auxiliary area contents:	170	061	512	612	503	653	703	154	275	765	426	087	897	677	908	509
Counts for tens digit distribution:					4	2	1	0	0	2	2	3	1	1		
Storage allocations based on these counts:					4	6	7	7	7	9	11	14	15	16		
Input area contents:	503	703	908	509	512	612	426	653	154	061	765	170	275	677	087	897
Counts for hundreds digit distribution:					2	2	1	0	1	3	3	2	1	1		
Storage allocations based on these counts:					2	4	5	5	6	9	12	14	15	16		
Auxiliary area contents:	061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908

An alert, "modern" reader will note, however, that the whole idea of making digit counts for the storage allocation is tied to "old-fashioned" ideas about sequential data representation; we know that *linked* allocation is specifically

designed to handle multiple tables of variable size, so it is natural to choose a linked data structure for radix sorting. Since we traverse each pile serially, all that is necessary is a single link from each item to its successor. Furthermore, we never need to move the records, we merely adjust the links and proceed merrily down the lists. The amount of memory required is $(1 + \epsilon)N + 2\epsilon M$ records, where ϵ is the amount of space taken up by a link field. Formal details of this procedure are rather interesting since they furnish an excellent example of typical data structure manipulations, combining sequential and linked allocation:

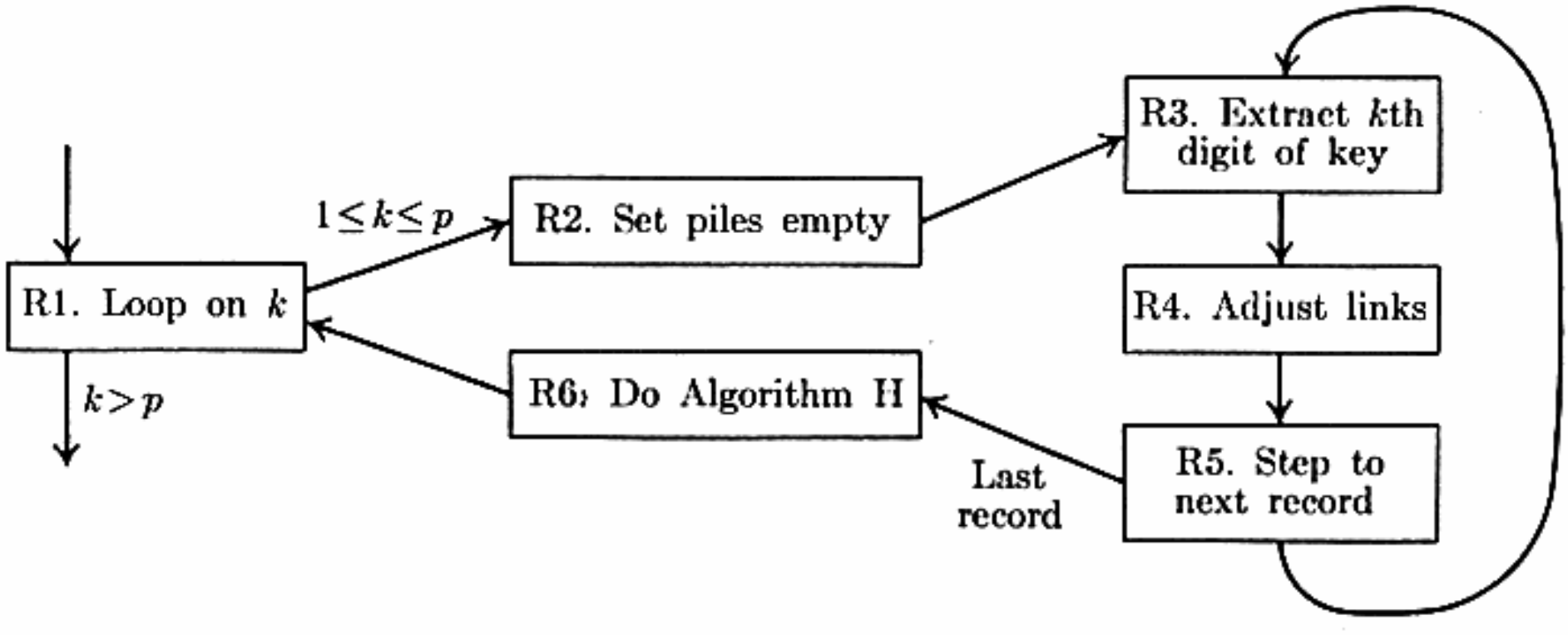


Fig. 32. Radix list sort.

Algorithm R (*Radix list sort*). Records $R_1, \ldots, R_N$ are each assumed to contain a LINK field. Their keys are assumed to be p-tuples

$$(a_p, \ldots, a_2, a_1), \qquad 0 \le a_i < M,$$

where the order is defined lexicographically so that

$$(a_p, \ldots, a_2, a_1) < (b_p, \ldots, b_2, b_1)$$

if and only if for some j, $1 \le j \le p$, we have

$$a_i = b_i \quad \text{for all} \quad i > j, \quad \text{but} \quad a_j < b_j.$$

The keys may, in particular, be thought of as numbers written in radix M notation,

$$a_p M^{p-1} + \cdots + a_2 M + a_1,$$

and in this case lexicographic order corresponds to the normal ordering of nonnegative numbers. The keys may also be strings of alphabetic letters, etc.

Sorting is done by keeping M "piles" of records, in a manner that exactly parallels the action of a card sorting machine. The piles are really queues in the sense of Chapter 2, since we link them together so that they are traversed

in a first-in-first-out manner. There are two pointer variables TOP[i] and BOTM[i] for each pile, $0 \le i < M$, and we assume as in Chapter 2 that

$$\texttt{LINK(LOC(BOTM([}i\texttt{]))))} \equiv \texttt{BOTM[}i\texttt{]}.$$

R1. [Loop on k.] In the beginning, set P $\leftarrow$ LOC(R_N), a pointer to the last record. Then perform steps R2 through R6 for $k = 1, 2, \ldots, p$. (Steps R2 through R6 constitute one "pass.") Then the algorithm terminates, with P pointing to the record with smallest key, LINK(P) to the record with next smallest, then LINK(LINK(P)), etc.; the LINK in the final record will be Λ.

R2. [Set piles empty.] Set TOP[i] $\leftarrow$ LOC(BOTM[i]) and BOTM[i] $\leftarrow \Lambda$, for $0 \le i < M$.

R3. [Extract kth digit of key.] Let KEY(P), the key in the record referenced by P, be $(a_p, \ldots, a_2, a_1)$; set $i \leftarrow a_k$, the kth least significant digit of this key.

R4. [Adjust links.] Set LINK(TOP[i]) $\leftarrow$ P, then set TOP[i] $\leftarrow$ P.

R5. [Step to next record.] If $k = 1$ (the first pass) and if P = LOC(R_j), for some $j \neq 1$, set P $\leftarrow$ LOC(R_{j-1}) and return to R3. If $k > 1$ (subsequent passes), set P $\leftarrow$ LINK(P), and return to R3 if P $\neq \Lambda$.

R6. [Do Algorithm H.] (We are now done distributing all elements onto the piles.) Perform Algorithm H below, which "hooks together" the individual piles into one list, in preparation for the next pass. Then set P $\leftarrow$ BOTM[0], a pointer to the first element of the hooked-up list. (See exercise 3.) ∎

Algorithm H (*Hooking-up of queues*). Given M queues, linked according to the conventions of Algorithm R, this algorithm adjusts at most M links so that a single queue is created, with BOTM[0] pointing to the first element, and with pile 0 preceding pile 1 $\cdots$ preceding pile $(M - 1)$.

H1. [Initialize.] Set $i \leftarrow 0$.

H2. [Point to top of pile.] Set P $\leftarrow$ TOP[i].

H3. [Next pile.] Increase i by 1. If $i = M$, set LINK(P) $\leftarrow \Lambda$ and terminate the algorithm.

H4. [Is pile empty?] If BOTM[i] $= \Lambda$, go back to H3.

H5. [Tie piles together.] Set LINK(P) $\leftarrow$ BOTM[i]. Return to H2. ∎

Figure 33 shows the contents of the piles after each of the three passes, when our 16 example numbers are sorted with $M = 10$. Algorithm R is very easy to program for MIX, once a suitable way to treat the pass-by-pass variation of steps R3 and R5 has been found. The following program does this without sacrificing any speed in the inner loop, by overlaying two of the instructions. Note that TOP[i] and BOTM[i] can be packed into the same word.

Program R (*Radix list sort*). The input in locations INPUT+1 through INPUT+N is assumed to have $p = 3$ components (a_3, a_2, a_1), stored respectively in the (1:1), (2:2), and (3:3) fields. (Thus M is assumed to be less than or equal to

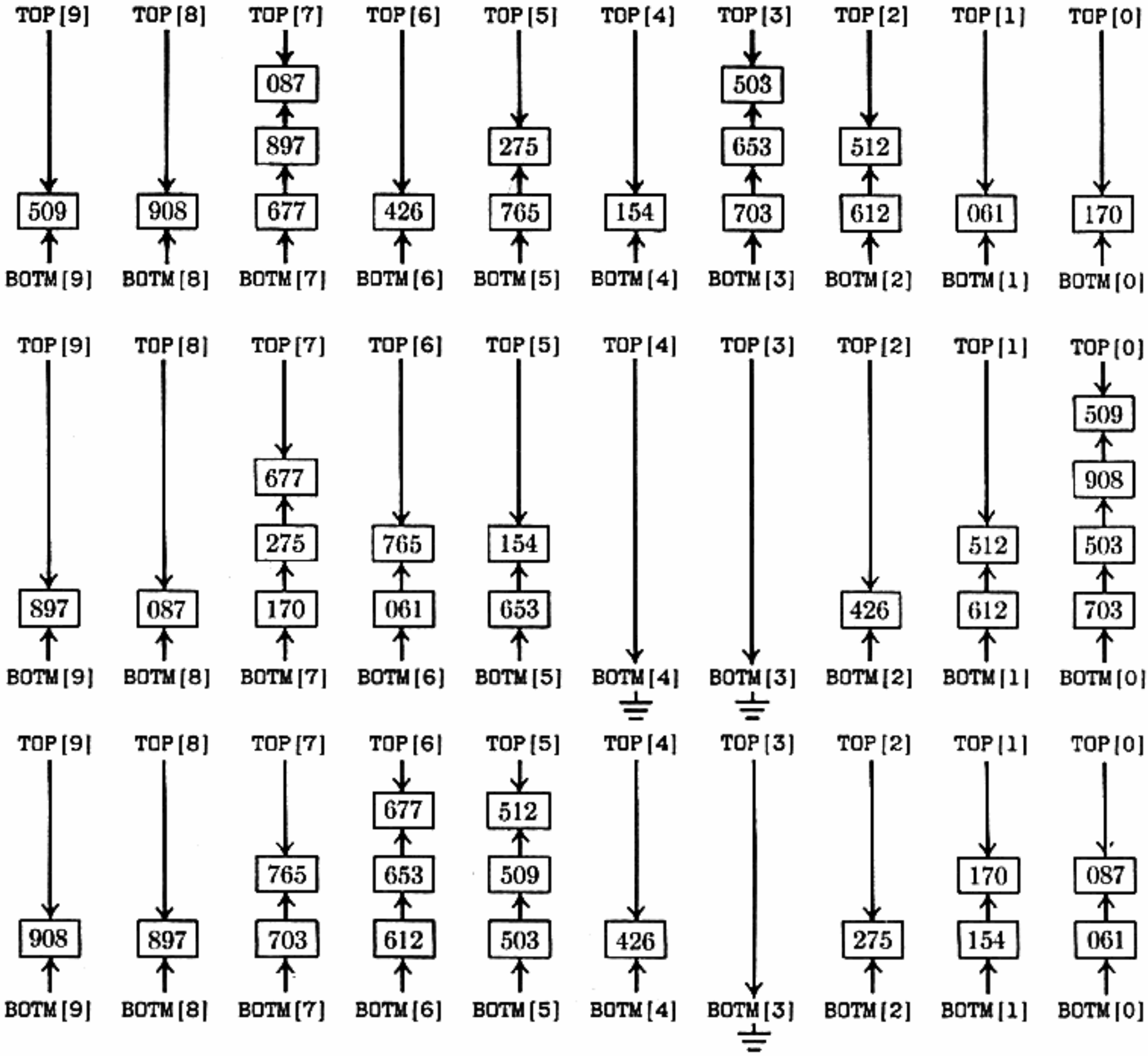


Fig. 33. Radix sort using linked allocation: contents of the ten piles after each pass.

the byte size of MIX.) The (4:5) field of each record is its LINK. We let TOP[i] ≡ PILES+i(1:2) and BOTM[i] ≡ PILES+i(4:5), for $0 \le i < M$. It is convenient to make links relative to location INPUT, so that LOC(BOTM[i]) = PILES + i − INPUT; to avoid negative links we therefore want the PILES table to be in higher locations than the INPUT table. Index registers are assigned as follows: rI1 ≡ P, rI2 ≡ i, rI3 ≡ 3 − k, rI4 ≡ TOP[i]; during Algorithm H, rI2 ≡ $i - M$.

```
01    LINK   EQU   4:5
02    TOP    EQU   1:2
03    START  ENT1  N                1     R1. Loop on k.  P ← LOC(R_N).
04           ENT3  2                1     k ← 1.
05    2H     ENT2  M-1              3     R2. Set piles empty.
06           ENTA  PILES-INPUT,2    3M    LOC(BOTM[i])
07           STA   PILES,2(TOP)     3M        → TOP[i]
08           STZ   PILES,2(LINK)    3M    BOTM[i] ← Λ.
09           DEC2  1                3M
10           J2NN  *-4              3M    M > i ≥ 0.
```

```
11          LDA   R3SW,3              3
12          STA   3F                  3      Modify instructions for pass k.
13          LDA   R5SW,3              3
14          STA   5F                  3
15    3H    [LD2  INPUT,1(3:3)]              R3. Extract kth digit of key.
16    4H    LD4   PILES,2(TOP)        3N     R4. Adjust links.
17          ST1   INPUT,4(LINK)       3N     LINK(TOP[i]) ← P.
18          ST1   PILES,2(TOP)        3N     TOP[i] ← P.
19    5H    [DEC1 1]                         R5. Step to next record.
20          J1NZ  3B                  3N     To R3 if end of pass.
21    6H    ENN2  M                   3      R6. Do Algorithm H.
22          JMP   7F                  3      To H2 with i ← 0.
23    R3SW  LD2   INPUT,1(1:1)        N      Instruction for R3 when k = 3.
24          LD2   INPUT,1(2:2)        N      Instruction for R3 when k = 2.
25          LD2   INPUT,1(3:3)        N      Instruction for R3 when k = 1.
26    R5SW  LD1   INPUT,1(LINK)       N      Instruction for R5 when k = 3.
27          LD1   INPUT,1(LINK)       N      Instruction for R5 when k = 2.
28          DEC1  1                   N      Instruction for R5 when k = 1.
29    9H    LDA   PILES+M,2(LINK)     3M − 3 H4. Is pile empty?
30          JAZ   8F                  3M − 3 To H3 if BOTM[i] = Λ.
31          STA   INPUT,1(LINK)       3M − 3 − E  H5. Tie piles together. LINK(P) ← BOTM[i].
32    7H    LD1   PILES+M,2(TOP)      3M − E H2. Point to top of pile.
33    8H    INC2  1                   3M     H3. Next pile. i ← i + 1.
34          J2NZ  9B                  3M     To H4 if i ≠ M.
35          STZ   INPUT,1(LINK)       3      LINK(P) ← Λ.
36          LD1   PILES(LINK)         3      P ← BOTM[0].
37          DEC3  1                   3
38          J3NN  2B                  3      1 ≤ k ≤ 3. ∎
```

The running time of Program R is $32N + 48M + 38 - 4E$, where N is the number of input records, M is the radix (the number of piles), and E is the number of occurrences of empty piles. This compares very favorably with other programs we have constructed based on similar assumptions (Programs 5.2.1M, 5.2.4L). A p-pass version of the program would take $(11p - 1)N + O(pM)$ units of time; the critical factor in the timing is the inner loop, which involves five references to memory and one branch. On a typical computer we will have $M = b$ and $p = \lceil t/r \rceil$, where t is the number of radix-b digits in the keys; increasing r will decrease p, so the formulas can be used to determine a "best" value of r.

The only variable in the timing is E, the number of empty piles observed in step H4. If we consider each of the M^N sequences of radix-M digits to be equally probable, we know from our study of the "poker test" in Section 3.3.2D that there are $M - r$ empty piles with probability

$$\frac{M(M-1)\cdots(M-r+1)}{M^N}\left\{{N \atop r}\right\}$$

on each pass, where $\{{N \atop r}\}$ is a Stirling number of the second kind. By exercise 5,

$$E = \left(\min\ \max(M-N,0)p, \qquad \text{ave}\ M\left(1-\frac{1}{M}\right)^N p, \qquad \max\,(M-1)p\right).$$

An ever-increasing number of "pipeline" or "number crunching" computers have appeared in recent years. These machines have multiple arithmetic units

and "look-ahead" circuitry so that memory references and computation can be highly overlapped; but their efficiency deteriorates noticeably in the presence of conditional branch instructions unless the branch almost always goes the same way. The inner loop of a radix sort is well adapted to such machines, because it is a straight iterative calculation of typical "number crunching" form. Therefore *radix sorting is usually more efficient than any other known method for internal sorting on such machines*, provided that N is not too small and the keys are not too long.

Of course, radix sorting is not very efficient when the keys are extremely long. For example, imagine sorting cards on an 80-column key; very few pairs of cards will tend to have identical keys in their first five columns, so the first 75 passes accomplish very little. In our analysis of radix-exchange sorting, we found that it was unnecessary to inspect many bits of the key, when we approach the keys from the left instead of the right. Let us therefore reconsider the idea of a radix sort which starts at the most significant digit (MSD) instead of the least significant digit (LSD).

We have already remarked that an MSD-first radix method suggests itself naturally; in fact, it is not hard to see why the post office uses such a method to sort mail. A large collection of letters can be sorted into separate bags for different geographical areas; each of these bags then contains a smaller number of letters which can be sorted independently of the other bags, into finer and finer geographical divisions. (Indeed, they can be transported nearer to their destinations before they are sorted further.) This principle of "divide and conquer" is quite appealing, and the only reason it doesn't work especially well for sorting punched cards is that the multiplicity of piles tends to become confusing. The same phenomenon accounts for the relative efficiency of Algorithm R, even though it considers LSD first, since we never have more than M piles, and they need to be hooked together only p times. On the other hand, it would not be difficult to design an MSD-first radix method using linked memory, with negative links as in Algorithm 5.2.4L to denote the boundaries between piles. (See exercise 10.)

Perhaps the best compromise has been suggested by M. D. MacLaren [*JACM* **13** (1966), 404–411], who recommends an LSD-first sort as in Algorithm R, but *applied only to the most significant digits*. This does not completely sort the file, but it usually brings the file very nearly into order so that very few inversions remain; therefore straight insertion can be used to finish up. Our analysis of Algorithm 5.2.1M applies also to this situation, so that if the keys are uniformly distributed we will have an average of

$$\tfrac{1}{4}N(N-1)M^{-p}$$

inversions remaining in the file after sorting on the leading p digits. (See Eq. 5.2.1–14 and exercise 5.2.1–38.) MacLaren has computed the average number of memory references per item sorted, and the optimum choice of M and p (assuming that M is a power of 2, that the keys are uniformly distributed,

and that $N/M^p \leq 0.1$ so that deviations from uniformity are tolerable) turns out to be given by the following table:

$N =$	100	1000	5000	10000	50000	100000
best $M =$	32	128	256	512	1024	1024
best $p =$	2	2	2	2	2	2
$\tilde{\beta}(N) =$	19.3	18.5	18.2	18.2	18.1	18.0

Here $\tilde{\beta}(N)$ denotes the average number of memory references per item sorted; it is bounded as $N \to \infty$, if we take $p = 2$ and $M > \sqrt{N}$, so the average sorting time is actually $O(N)$ instead of order $N \log N$. This method is an improvement over multiple list insertion (Algorithm 5.2.1M) which is essentially the case $p = 1$. Exercise 12 gives MacLaren's interesting procedure for final rearrangement of a partially list-sorted file.

It is also possible to avoid the link fields, using the methods of Algorithm 5.2D and exercise 5.2–13, so that only $O(\sqrt{N})$ memory locations are needed in addition to the space required for the records themselves. The average sorting time is proportional to N if the input records are uniformly distributed.

EXERCISES

▶ **1.** [*20*] The algorithm of exercise 5.2–13 shows how to do a distribution sort with only N record areas (and M count fields), instead of $2N$ record areas. Does this lead to an improvement over the radix sorting algorithm illustrated in Table 1?

2. [*13*] Is Algorithm R a "stable" sorting method?

3. [*15*] Explain why Algorithm H makes `BOTM[0]` point to the first record in the "hooked-up" queue, *even though pile* 0 *might be empty*.

▶ **4.** [*23*] Algorithm R keeps the M piles linked together as queues (first-in-first-out). Explore the idea of linking the piles as *stacks* instead. (The arrows in Fig. 33 would go downward instead of upward, and the `BOTM` table would be unnecessary.) Show that if the piles are "hooked together" in an appropriate order, it is possible to achieve a valid sorting method. Does this lead to a simpler or a faster algorithm?

5. [*M24*] Let $g_{MN}(z) = \sum p_{MNk} z^k$, where p_{MNk} is the probability that exactly k empty piles are present after a random radix-sort pass puts N elements into M piles. (a) Show that $g_{M,N+1}(z) = g_{MN}(z) + ((1 - z)/M) g'_{MN}(z)$. (b) Use this relation to find simple expressions for the mean and variance of this probability distribution, as a function of M and N.

6. [*20*] What changes are necessary to Program R so that it sorts eight-byte keys instead of three-byte keys? Assume that the most significant bytes of K_i are stored in location `KEY+`i`(1:5)`, while the three least significant bytes are in location `INPUT+`i`(1:3)` as presently. What is the running time of the program, after these changes have been made?

7. [*20*] Discuss the similarities and differences between Algorithm R and radix-exchange sorting (Algorithm 5.2.2R).

▸ **8.** [*20*] The radix-sorting algorithms discussed in the text assume that all keys being sorted are nonnegative. What changes should be made to the algorithms when the keys are numbers expressed in *two's complement* or *ones' complement* notation?

9. [*20*] Continuing exercise 8, what changes should be made to the algorithms when the keys are numbers expressed in *signed-magnitude* notation?

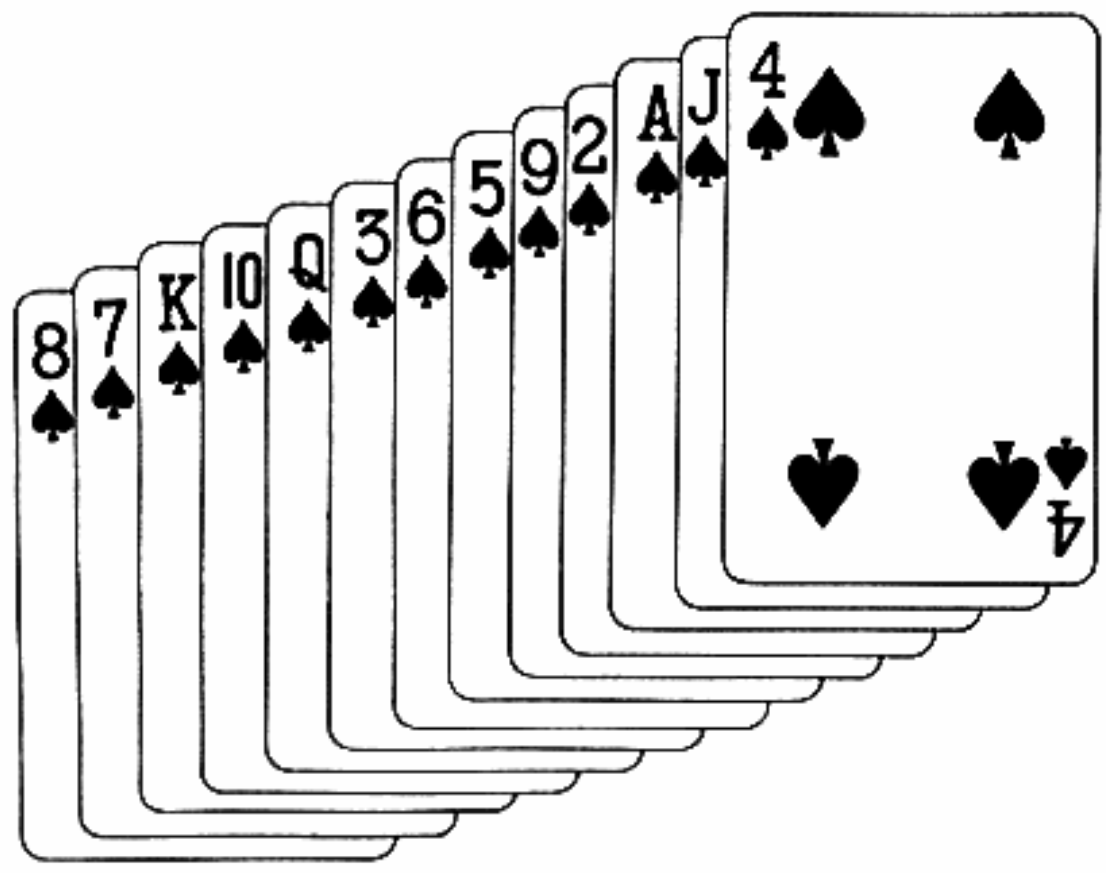
10. [*30*] Design an efficient most-significant-digit-first radix-sorting algorithm which uses linked memory. (As the size of the subfiles decreases, it is wise to decrease M, and to use a nonradix method on the really short subfiles.)

11. [*16*] The sixteen input numbers shown in Table 1 start with 41 inversions; after sorting is complete, of course, there are no inversions remaining. How many inversions would be present in the file if we omitted pass 1, doing a radix sort only on the tens and hundreds digits? How many inversions would be present if we omitted both pass 1 and pass 2?

12. [*24*] (M. D. MacLaren.) Suppose that Algorithm R has been applied only to the p leading digits of the actual keys; thus the file is nearly sorted when we read it in the order of the links, but keys which agree in their first p digits may be out of order. Design an algorithm which rearranges the records in place so that their keys are in order, $K_1 \leq K_2 \leq \ldots \leq K_N$. [*Hint:* The special case that the file is perfectly sorted appears in the answer to exercise 5.2–12; it is possible to combine this with straight insertion without loss of efficiency, since few inversions remain in the file.]

13. [*40*] Implement the internal sorting method suggested in the text at the close of this section, producing a subroutine which sorts random data in $O(N)$ units of time with only $O(\sqrt{N})$ additional memory locations.

14. [*22*] The sequence of playing cards



can be sorted into increasing order A 2 . . . J Q K from top to bottom in two passes, using just two piles for intermediate storage: Deal the cards face down into two piles containing respectively A 2 9 3 10 and 4 J 5 6 Q K 7 8 (from bottom to top); then put the second pile on the first, turn the deck face up, and deal into two piles A 2 3 4 5 6 7 8, 9 10 J Q K. Combine these piles, turn them face up, and you're done.

Prove that the above sequence of cards cannot be sorted into *decreasing* order K Q J . . . 2 A from top to bottom in two passes, even if you are allowed to use up to three piles for intermediate storage. (Dealing must always be from the top of the deck, turning the cards face down as they are dealt. Top to bottom is right to left in the illustration.)

15. [*M25*] Consider the problem of exercise 14 when all cards must be dealt face up instead of face down. Thus, one pass can be used to convert increasing order into decreasing order. How many passes are required?

As soon as an Analytical Engine exists,
it will necessarily guide the future course of the science.
Whenever any result is sought by its aid,
the question will then arise—
By what course of calculation can these
results be arrived at by the machine
in the shortest time?

—CHARLES BABBAGE (1864)

5.3. OPTIMUM SORTING

Now that we have analyzed a great many methods for internal sorting, it is time to turn to a broader question: *What is the best possible way to sort?* Can we place limits on the maximum sorting speeds that will ever be achievable, no matter how clever a programmer might be?

Of course there *is* no best possible way to sort; we must define precisely what is meant by "best," and there is no best possible way to define "best." We have discussed similar questions about the theoretical optimality of algorithms in Sections 4.3.3, 4.6.3, and 4.6.4, where high-precision multiplication and polynomial evaluation were considered. In each case it was necessary to formulate a rather simple definition of a "best possible" algorithm, in order to give sufficient structure to the problem to make it workable. And in each case we ran into interesting problems that are so difficult they still haven't been completely resolved. The same situation holds for sorting; some very interesting discoveries have been made, but there are many fascinating questions still unanswered.

Studies of the inherent complexity of sorting have usually been directed towards minimizing the number of times we make comparisons between keys while sorting n items, or merging m items with n, or selecting the tth largest of an unordered set of n items. Sections 5.3.1, 5.3.2, and 5.3.3 discuss these questions in general, and Section 5.3.4 deals with similar issues under the interesting restriction that the pattern of comparisons must essentially be fixed in advance. Some other types of interesting theoretical questions related to optimum sorting appear in the exercises for Section 5.3.4, and in the discussion of external sorting in Section 5.4.4.

5.3.1. Minimum-Comparison Sorting

The minimum number of key comparisons needed to sort n elements is obviously *zero*, because we have seen radix methods that do no comparisons at all. In fact, it is possible to write MIX programs that are able to sort, although they contain no conditional jump instructions at all! (See exercise 5–6 at the beginning of this chapter.) We have also seen several sorting methods which are based essentially on comparisons of keys, yet their running time in practice is dominated by other considerations such as data movement, housekeeping operations, etc.

Therefore it is clear that comparison counting is not the only way to measure the effectiveness of a sorting method. But it is fun to scrutinize the number of comparisons anyway, since a theoretical study of this subject gives us a good deal of useful insight into the nature of sorting processes, and it also helps us to sharpen our wits for the more mundane problems that may confront us at other times.

In order to rule out radix-sorting methods, which do no comparisons at all, we shall restrict our discussion to sorting techniques which are based solely on

an abstract linear ordering relation "$<$" between keys, as discussed at the beginning of this chapter. For simplicity, we shall also confine our discussion to the case of *distinct* keys, so that there are only two possible outcomes of any comparison of K_i vs. K_j: either $K_i < K_j$ or $K_i > K_j$. (For an extension of the theory to the general case where equal keys are allowed, see exercises 3 through 12.)

The problem of sorting by comparisons can also be expressed in other equivalent ways. Given a set of n distinct weights and a balance scale, we can ask for the least number of weighings necessary to completely rank the weights in order of magnitude, when the pans of the balance scale can each accommodate only one weight. Alternatively, given a set of n players in a tournament, we can ask for the smallest number of games which suffice to rank all contestants, assuming that the strengths of the players can be linearly ordered (with no ties).

All n-element sorting methods which satisfy the above constraints can be represented in terms of an extended binary tree structure such as that shown in Fig. 34. Each *internal node* (drawn as a circle) contains two indices "$i:j$" denoting a comparison of K_i versus K_j. The left subtree of this node represents the subsequent comparisons to be made if $K_i < K_j$, and the right subtree represents the actions to be taken when $K_i > K_j$. Each *external node* of the tree (drawn as a box) contains a permutation $a_1\, a_2 \ldots a_n$ of $\{1, 2, \ldots, n\}$, denoting the fact that the ordering

$$K_{a_1} < K_{a_2} < \cdots < K_{a_n}$$

has been established. (If we look at the path from the root to this external node, each of the $n - 1$ relationships $K_{a_i} < K_{a_{i+1}}$ for $1 \leq i < n$ will be the result of some comparison $a_i:a_{i+1}$ or $a_{i+1}:a_i$ on this path.)

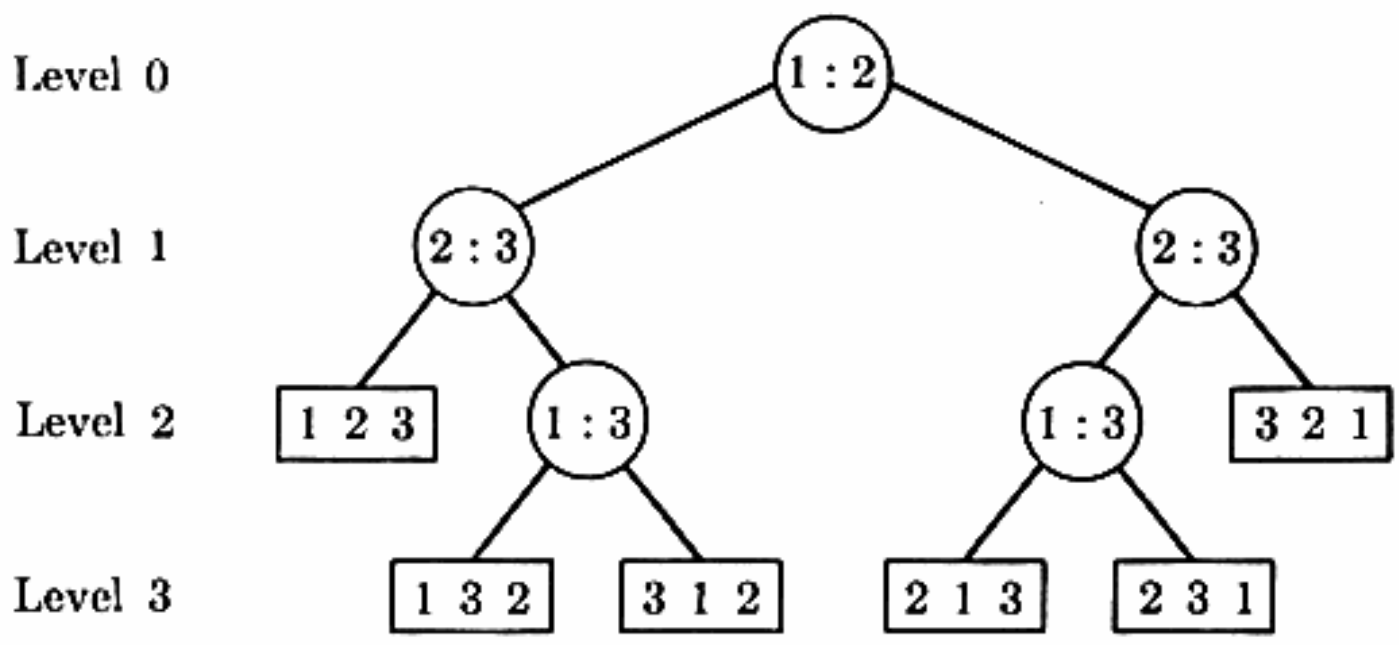


Fig. 34. A comparison tree for sorting three elements.

Thus Fig. 34 represents a sorting method which first compares K_1 with K_2; if $K_1 > K_2$, it goes on (via the right subtree) to compare K_2 with K_3, and then if $K_2 < K_3$ it compares K_1 with K_3; finally if $K_1 > K_3$ it knows that $K_2 < K_3 < K_1$. An actual sorting algorithm will usually also move the keys around in the file, but we are interested here only in the comparisons, so we ignore

all data movement. A comparison of K_i with K_j in this tree always means the *original* keys K_i and K_j, not the keys which might currently occupy the ith and jth positions of the file after the records have been shuffled around.

It is possible to make redundant comparisons; for example, in Fig. 35 there is no reason to compare 3:1, since $K_1 < K_2$ and $K_2 < K_3$ implies that $K_1 < K_3$. No permutation can possibly correspond to the left subtree of node (3:1) in Fig. 35, so that part of the algorithm will never be performed! Since we are interested in minimizing the number of comparisons, we may assume that no redundant comparisons are made; hence we have an extended binary tree structure in which every external node corresponds to a permutation. All permutations of the input keys are possible, and every permutation defines a unique path from the root to an external node; it follows that *there are exactly $n!$ external nodes in a comparison tree which sorts n elements with no redundant comparisons.*

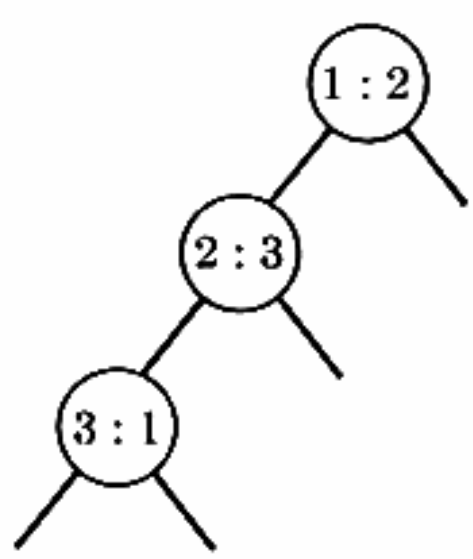


Fig. 35. Example of a redundant comparison.

The best worst case. The first problem that naturally arises is to find comparison trees which minimize the *maximum* number of comparisons made. (Later we shall consider the *average* number of comparisons.)

Let $S(n)$ be the minimum number of comparisons which will suffice to sort n elements. If all the internal nodes of a comparison tree are at levels $< k$, it is obvious that there can be at most 2^k external nodes in the tree. Hence, letting $k = S(n)$, we have

$$n! \le 2^{S(n)}.$$

Since $S(n)$ is an integer, we can rewrite this formula to obtain the lower bound

$$S(n) \ge \lceil \log_2 n! \rceil. \tag{1}$$

Note that by Stirling's approximation,

$$\lceil \log_2 n! \rceil = n \log_2 n - n/(\ln 2) + \tfrac{1}{2} \log_2 n + O(1), \tag{2}$$

so that about $n \log_2 n$ comparisons are needed.

Relation (1) is often called the "information theoretic lower bound," since cognoscenti of information theory would say that $(\log_2 n!)$ "bits of information" are being acquired during a sorting process; each comparison yields at most one "bit of information." Trees such as Fig. 34 have also been called "questionnaires," and their mathematical properties have been explored in Claude Picard's book *Théorie des questionnaires* (Paris: Gauthier-Villars, 1965).

Of all the sorting methods we have seen, the three that require fewest comparisons are binary insertion (cf. Section 5.2.1), tree selection (cf. Section 5.2.3), and straight two-way merging as reflected in Algorithm 5.2.4L. The maximum number of comparisons for binary insertion is readily seen to be

$$B(n) = \sum_{1\le k\le n} \lceil \log_2 k \rceil = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + 1 \tag{3}$$

(cf. exercise 1.2.4–42), and the maximum number of comparisons in Algorithm 5.2.4L is given in exercise 5.2.4–14. It turns out (see Section 5.3.3) that tree selection has the same bound on its comparisons as either binary insertion or two-way merging, depending on how the tree is set up. In all three cases we achieve an asymptotic value of $n \log_2 n$; combining these lower and upper bounds for $S(n)$ proves that

$$\lim_{n\to\infty} \frac{S(n)}{n \log_2 n} = 1. \tag{4}$$

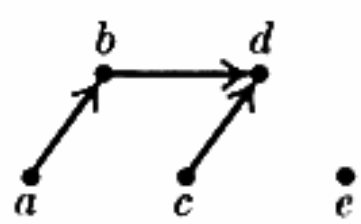
Thus we have an approximate formula for $S(n)$, but it is desirable to obtain more precise information. The following table gives exact values of the above quantities, for small n:

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\lceil \log_2 n! \rceil =$	0	1	3	5	7	10	13	16	19	22	26	29	33	37	41	45	49
$B(n) =$	0	1	3	5	8	11	14	17	21	25	29	33	37	41	45	49	54
$L(n) =$	0	1	3	5	9	11	14	17	25	27	30	33	38	41	45	49	65

Here $B(n)$ and $L(n)$ refer respectively to binary insertion and list merging. It can be shown that $B(n) \le L(n)$ for all n (see exercise 2).

From the above table, we can see that $S(4) = 5$, but $S(5)$ might be either 7 or 8. This brings us back to a problem stated at the beginning of Section 5.2: What is the best way to sort five elements? Can five elements be sorted using only seven comparisons?

The answer is yes, but it is not especially easy to find the method. We begin as if we were sorting four elements by merging, first comparing $K_1{:}K_2$, then $K_3{:}K_4$, then the larger elements of these pairs. This produces a configuration which may be diagramed as



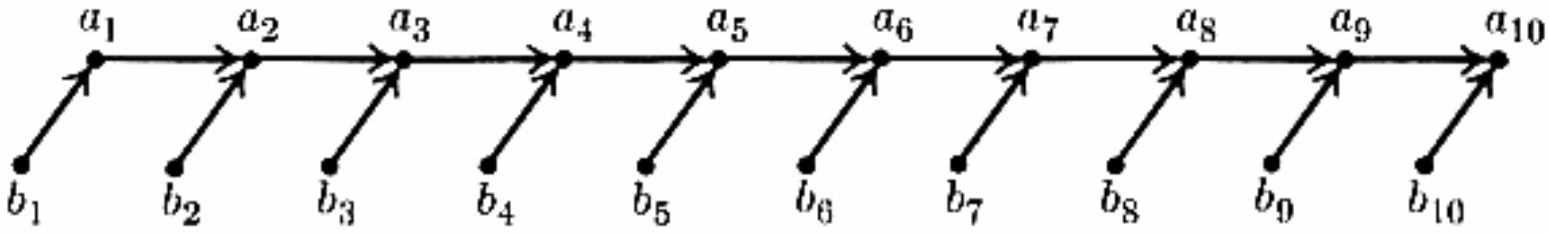
(5)

indicating that $a < b < d$ and $c < d$. (It is convenient to represent known ordering relations between elements by drawing directed graphs such as this, where x is known to be less than y if and only if there is a path from x to y in the graph.) At this point we insert the fifth element $K_5 = e$ into its proper place among $\{a, b, d\}$; only two comparisons are needed, since we may compare it first with b and then with a or d. This leaves one of four possibilities,

(6)

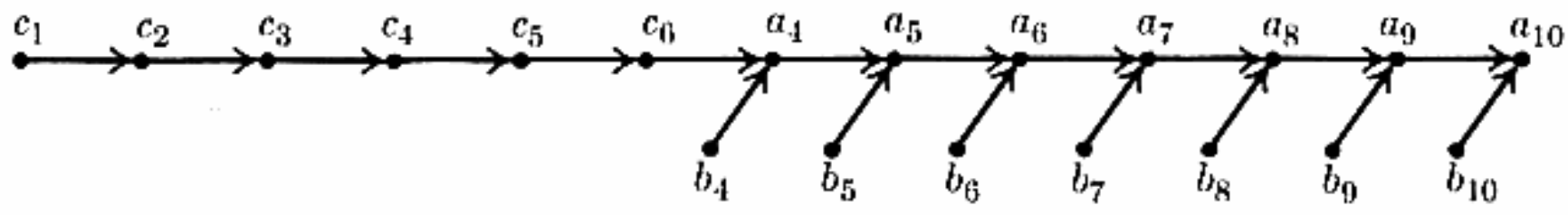
and in each case we can insert c among the remaining elements less than d in two more comparisons. This method for sorting five elements was first discovered by H. B. Demuth [Ph.D. thesis, Stanford University (Oct. 1956), 41–43].

Merge insertion. A pleasant generalization of the above method has been discovered by Lester Ford, Jr. and Selmer Johnson. Since it involves some aspects of merging and some aspects of insertion, we shall call it *merge insertion*. For example, consider the problem of sorting 21 elements. We start by comparing the ten pairs $K_1{:}K_2$, $K_3{:}K_4$, . . . , $K_{19}{:}K_{20}$; then we sort the ten larger elements of the pairs, using merge insertion. As a result we obtain the configuration



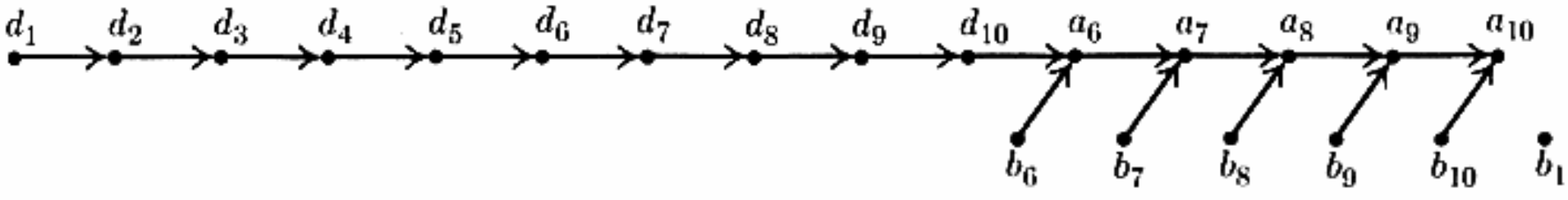
(7)

analogous to (5). The next step is to insert b_3 among $\{b_1, a_1, a_2\}$, then b_2 among the other elements less than a_2; we arrive at the configuration



(8)

Let us call the upper-line-elements the *main chain*. We can insert b_5 into its proper place in the main chain, using three comparisons (first comparing it to c_4, then c_2 or c_6, etc.); then b_4 can be moved into the main chain in three more steps, leading to



(9)

The next step is crucial; is it clear what to do? We insert b_{11} (*not* b_7) into the main chain, using only four comparisons. Then each of b_{10}, b_9, b_8, b_7, b_6 (in this order) can also be inserted into their proper places in the main chain, using at most four comparisons each.

A careful count of the comparisons involved here shows that the 21 elements have been sorted in at most $10 + 22 + 2 + 2 + 3 + 3 + 4 + 4 + 4 + 4 + 4 + 4 = 66$ steps. Since

$$2^{65} < 21! < 2^{66},$$

we also know that no fewer than 66 would be possible in any event; hence

$$S(21) = 66. \tag{10}$$

(Binary insertion would have required 74 comparisons.)

In general, merge insertion proceeds as follows for n elements:

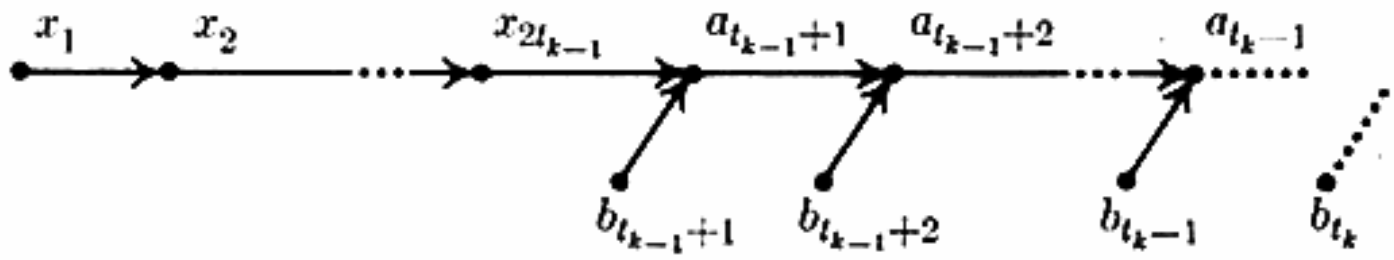
i) Make pairwise comparisons of $\lfloor n/2 \rfloor$ disjoint pairs of elements. (If n is odd, leave one element out.)

ii) Sort the $\lfloor n/2 \rfloor$ larger numbers, found in step (i), by merge insertion.

iii) Name the elements $a_1, a_2, \ldots, a_{\lfloor n/2 \rfloor}, b_1, b_2, \ldots, b_{\lceil n/2 \rceil}$ as in (7), where $a_1 \le a_2 \le \cdots \le a_{\lfloor n/2 \rfloor}$ and $b_i \le a_i$ for $1 \le i \le \lfloor n/2 \rfloor$; call b_1 and the a's the "main chain." Insert the remaining b's into the main chain, using binary insertion, in the following order, leaving out all b_j for $j > \lceil n/2 \rceil$:

$$b_3, b_2; b_5, b_4; b_{11}, b_{10}, \ldots, b_6; \ldots; b_{t_k}, b_{t_k-1}, \ldots, b_{t_{k-1}+1}; \ldots \tag{11}$$

We wish to define the sequence $(t_1, t_2, t_3, t_4, \ldots) = (1, 3, 5, 11, \ldots)$, which appears in (11), in such a way that each of $b_{t_k}, b_{t_k-1}, \ldots, b_{t_{k-1}+1}$ can be inserted into the main chain with at most k comparisons. Generalizing (7), (8), and (9), we obtain the diagram



where the main chain up to and including a_{t_k-1} contains $2t_{k-1} + (t_k - t_{k-1} - 1)$ elements. This number must be less than 2^k; our best bet is to set it equal to $2^k - 1$, so that

$$t_{k-1} + t_k = 2^k. \tag{12}$$

Since $t_1 = 1$, we may set $t_0 = 1$ for convenience, and we find that

$$\begin{aligned} t_k = 2^k - t_{k-1} = 2^k - 2^{k-1} + t_{k-2} = \cdots &= 2^k - 2^{k-1} + \cdots + (-1)^k 2^0 \\ &= (2^{k+1} + (-1)^k)/3 \end{aligned} \tag{13}$$

by summing a geometric series. (Curiously, this same sequence arose in our study of an algorithm for calculating the greatest common divisor of two integers; cf. exercise 4.5.2–27.)

Let $F(n)$ be the number of comparisons required to sort n elements by merge insertion. Clearly

$$F(n) = \lfloor n/2 \rfloor + F(\lfloor n/2 \rfloor) + G(\lceil n/2 \rceil), \tag{14}$$

where G represents the amount of work involved in step (iii). If $t_{k-1} \le m \le t_k$, we have

$$\begin{aligned} G(m) &= \sum_{1 \le j < k} j(t_j - t_{j-1}) + k(m - t_{k-1}) \\ &= km - (t_0 + t_1 + \cdots + t_{k-1}), \end{aligned} \tag{15}$$

summing by parts. Let us set

$$w_k = t_0 + t_1 + \cdots + t_{k-1} = \lfloor 2^{k+1}/3 \rfloor, \tag{16}$$

so that $(w_0, w_1, w_2, w_3, w_4, \ldots) = (0, 1, 2, 5, 10, 21, \ldots)$. Exercise 13 shows that

$$F(n) - F(n-1) = k \quad \text{iff} \quad w_k < n \le w_{k+1}, \tag{17}$$

and the latter condition is equivalent to

$$\frac{2^{k+1}}{3} < n \le \frac{2^{k+2}}{3},$$

$$k + 1 < \log_2 (3n) \le k + 2;$$

hence

$$F(n) - F(n-1) = \lceil \log_2 (\tfrac{3}{4}n) \rceil. \tag{18}$$

(This formula is due to A. Hadian [Ph. D. thesis, Univ. of Minnesota (1969), 38–42].) It follows that $F(n)$ has a remarkably simple expression,

$$F(n) = \sum_{1 \le k \le n} \lceil \log_2 (\tfrac{3}{4}k) \rceil, \tag{19}$$

quite similar to the corresponding formula (3) for binary insertion. A "closed form" for this sum appears in exercise 14.

Equation (19) makes it easy to construct a table of $F(n)$; we have

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\lceil \log_2 n! \rceil =$	0	1	3	5	7	10	13	16	19	22	26	29	33	37	41	45	49
$F(n) =$	0	1	3	5	7	10	13	16	19	22	26	30	34	38	42	46	50

$n =$	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
$\lceil \log_2 n! \rceil =$	53	57	62	66	70	75	80	84	89	94	98	103	108	113	118	123
$F(n) =$	54	58	62	66	71	76	81	86	91	96	101	106	111	116	121	126

Note that $F(n) = \lceil \log_2 n! \rceil$ for $1 \le n \le 11$ and for $20 \le n \le 21$, so we know that merge insertion is optimum for these n:

$$S(n) = \lceil \log_2 n! \rceil = F(n) \quad \text{for} \quad 1 \le n \le 11 \quad \text{and} \quad 20 \le n \le 21. \tag{20}$$

Hugo Steinhaus posed the problem of finding $S(n)$ in the second edition of his classic book *Mathematical Snapshots* (Oxford University Press, 1950), pp. 38–39. He described the method of binary insertion, which is the best possible way to sort n objects if we start by sorting $n - 1$ of them first before the nth is considered; and he conjectured that binary insertion is optimum in general. Several years later [*Calcutta Math. Soc. Golden Jubilee Commemoration* **2** (1959), 323–327], he reported that two of his colleagues, S. Trybuła and C. Ping, had "recently" disproved his conjecture, and that they had determined $S(n)$ for $n \le 11$. Trybuła and Ping may have independently discovered the method of merge insertion, which was published soon afterwards by Ford and Johnson [*AMM* **66** (1959), 387–389].

After the discovery of merge insertion, the first unknown value of $S(n)$ was $S(12)$. Table 1 shows that 12! is quite close to 2^{29}, so that the existence of a 29-step sorting procedure for 12 elements is somewhat unlikely. An exhaustive search (about 60 hours on a Maniac II computer) was therefore carried out by Mark Wells, who discovered that $S(12) = 30$ [*Proc. IFIP Congress* 65 **2** (1965), 497–498]. Thus the merge insertion procedure turns out to be optimum for $n = 12$ as well.

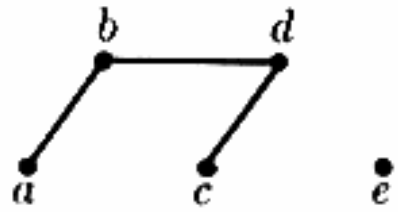
Table 1

VALUES OF FACTORIALS IN BINARY NOTATION

1 = 1!
10 = 2!
110 = 3!
11000 = 4!
1111000 = 5!
1011010000 = 6!
1001110110000 = 7!
1001110110000000 = 8!
101100010011000000 = 9!
11011101011111000000000 = 10!
1001100001000101010000000000 = 11!
111001000110011111100000000000 = 12!
101110011001010001100110000000000 = 13!
101000100110111111011001010000000000 = 14!
100110000011101110111011101011000000000000 = 15!
1001100000111011101110111010110000000000000000 = 16!

***A slightly deeper analysis.** In order to study $S(n)$ more carefully, let us look more closely at partial ordering diagrams such as (5). After several comparisons

have been made, we can represent the knowledge we have gained in terms of a directed graph. This directed graph contains no cycles, in view of the transitivity of the $<$ relation, so we can draw it in such a way that all arcs go from left to right; it is therefore convenient to leave arrows off the diagram. In this way (5) becomes



(21)

If G is such a directed graph, let $T(G)$ be the number of permutations consistent with G, i.e., the number of ways to assign the integers $\{1, 2, \ldots, n\}$ to the vertices of G so that the number on vertex x is less than the number on vertex y whenever $x \longrightarrow y$ in G. For example, one of the permutations consistent with (21) has $a = 1$, $b = 4$, $c = 2$, $d = 5$, $e = 3$. We have studied $T(G)$ for various G in Section 5.1.4, where we observed that $T(G)$ is the number of ways in which G can be "topologically sorted."

If G is a graph on n elements which can be obtained after k comparisons, we define the *efficiency* of G to be

$$E(G) = \frac{n!}{2^k T(G)}. \tag{22}$$

(This idea is due to Frank Hwang and Shen Lin.) Strictly speaking, the efficiency is not a function of the graph G alone, it depends on the way we arrived at G during a sorting process, but it is convenient to be a little careless in our language. After making one more comparison, between elements i and j, we obtain two graphs G_1 and G_2, one for the case $K_i < K_j$ and one for the case $K_i > K_j$. Clearly

$$T(G) = T(G_1) + T(G_2).$$

If $T(G_1) \geq T(G_2)$, we have

$$T(G) \leq 2T(G_1),$$

$$E(G_1) = \frac{n!}{2^{k+1} T(G_1)} = \frac{E(G)T(G)}{2T(G_1)} \leq E(G). \tag{23}$$

Therefore each comparison leads to a graph of less or equal efficiency; we can't improve the efficiency by making further comparisons.

Note that when G has no arcs at all, we have $k = 0$ and $T(G) = n!$, so the initial efficiency is 1. And when G is a graph representing the final result of sorting, G looks like a straight line, and $T(G) = 1$. Thus, for example, if we want to find a sorting procedure that sorts five elements in seven or less steps, we must obtain the linear graph  whose efficiency is $5!/(2^7 \times 1) = 120/128 = 15/16$. It follows that all of the graphs arising in the sorting procedure must have efficiency $\geq \frac{15}{16}$; if any less efficient graph were to appear, at least one of its descendants would also be less efficient, and

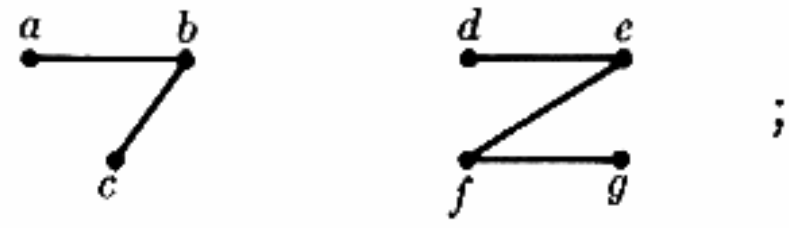
we would ultimately reach a linear graph whose efficiency is $< \frac{15}{16}$. In general, this argument proves that all graphs corresponding to the tree nodes of a sorting procedure for n elements must have efficiency $\geq n!/2^l$, where $l+1$ is the number of levels of the tree. This is another way to prove that $S(n) \geq \lceil \log_2 n! \rceil$, although the argument is not really much different than what we said before.

The graph (21) has efficiency 1, since $T(G) = 15$ and since G has been obtained in three comparisons. In order to see what vertices should be compared next, we can form the *comparison matrix*

$$C(G) = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{pmatrix} 0 & 15 & 10 & 15 & 11 \\ 0 & 0 & 5 & 15 & 7 \\ 5 & 10 & 0 & 15 & 9 \\ 0 & 0 & 0 & 0 & 3 \\ 4 & 8 & 6 & 12 & 0 \end{pmatrix} \end{array}, \tag{24}$$

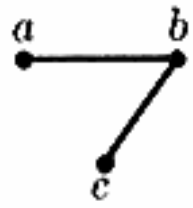
where C_{ij} is $T(G_1)$ for the graph G_1 obtained by adding the arc $i \longrightarrow j$ to G. For example, if we compare K_c with K_e, the 15 permutations consistent with G split up into $C_{ce} = 6$ having $K_c < K_e$ and $C_{ec} = 9$ having $K_c > K_e$. The latter graph would have efficiency $15/(2 \times 9) = \frac{5}{6} < \frac{15}{16}$, so it could not lead to a seven-step sorting procedure. The next comparison *must* be $K_b : K_c$ in order to keep the efficiency $\geq \frac{15}{16}$.

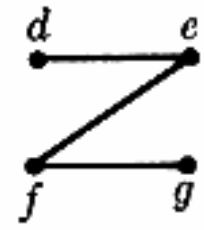
The concept of "efficiency" is especially useful when we consider the "connected components" of graphs. Consider for example the graph

$G =$ 

;

it has two components,

$G' =$ 

and $G'' =$ 

with no arcs connecting G' to G'', so it has been formed by making some comparisons entirely within G' and others entirely within G''. In general, assume that $G = G' + G''$ has no arcs between G' and G'', where G' and G'' have respectively n' and n'' vertices; it is easy to see that

$$T(G) = \binom{n'+n''}{n'} T(G')T(G''), \tag{25}$$

since each consistent permutation of G is obtained by choosing n' elements to

assign to G' and then making consistent permutations within G' and G'' independently. If k' comparisons have been made within G' and k'' within G'', we have the basic result

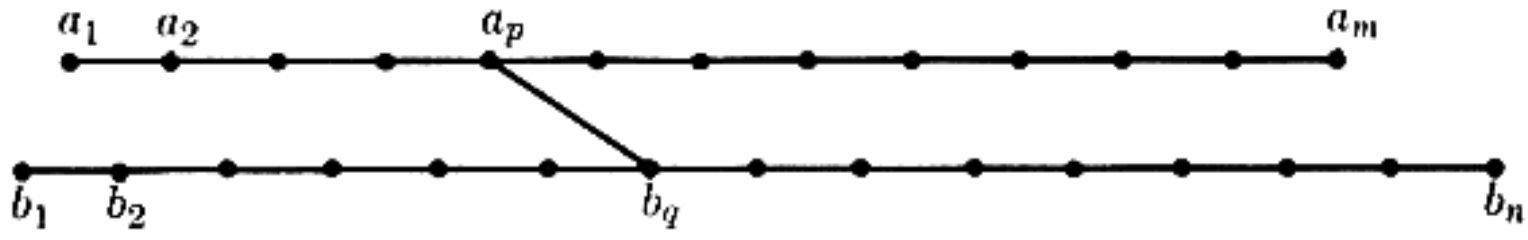
$$E(G) = \frac{(n'+n'')!}{2^{k'+k''}T(G)} = \frac{n'!}{2^{k'}T(G')} \cdot \frac{n''!}{2^{k''}T(G'')} = E(G')E(G''), \tag{26}$$

showing that the efficiency of a graph is related in a simple way to the efficiency of its components. Therefore we may restrict consideration to graphs having only one component.

Now suppose that G' and G'' are one-component graphs, and suppose that we want to hook them together by comparing a vertex x of G' with a vertex y of G''. We want to know how efficient this will be. For this purpose we need a function which can be denoted by

$$\binom{p}{m} < \binom{q}{n}, \tag{27}$$

defined to be the number of permutations consistent with the graph



(28)

Thus $\binom{p}{m}<\binom{q}{n}$ is $\binom{m+n}{m}$ times the probability that the pth smallest of a set of m numbers is less than the qth smallest of an independently chosen set of n numbers. Exercise 17 shows that we can express $\binom{p}{m}<\binom{q}{n}$ in two ways in terms of binomial coefficients,

$$\binom{p}{m} > \binom{q}{n} = \sum_{0\le k<q} \binom{m-p+n-k}{m-p}\binom{p-1+k}{p-1}$$

$$= \sum_{p\le j\le m} \binom{n-q+m-j}{n-q}\binom{q-1+j}{q-1}. \tag{29}$$

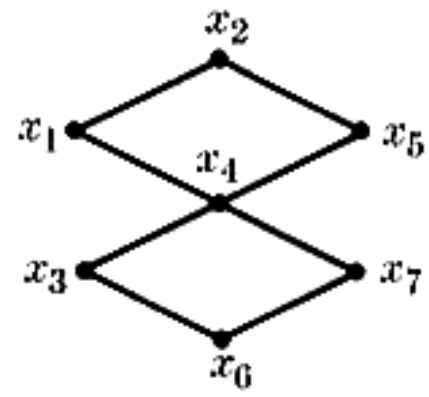
(Incidentally, it is by no means obvious on algebraic grounds that these two sums of products of binomial coefficients should come out to be equal.) We also have the formulas

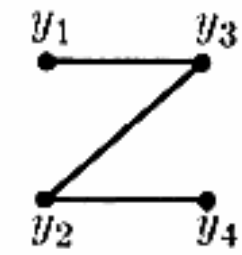
$$\binom{p}{m} < \binom{q}{n} + \binom{q}{n} < \binom{p}{m} = \binom{m+n}{m}, \tag{30}$$

$$\binom{q}{n} < \binom{p}{m} = \binom{m+1-p}{m} < \binom{n+1-q}{n}. \tag{31}$$

For definiteness, let us now consider the two graphs

$$G' = \quad , \qquad G'' = \qquad (32)$$





It is not hard to show by direct enumeration that $T(G') = 42$ and $T(G'') = 5$; so if G is the 11-vertex graph having G' and G'' as components, we have $T(G) = \binom{11}{4} \cdot 42 \cdot 5 = 69300$ by Eq. (25). This is a formidable number of permutations to list, if we want to know how many of them have $x_i < y_j$ for each i and j. But the calculation can be done by hand, in less than an hour, as follows. We form the matrices $A(G')$ and $A(G'')$, where A_{ik} is the number of consistent permutations of G' (or G'') in which x_i (or y_i) is equal to k. Thus the number of permutations of G in which x_i is less than y_j is the (ip) element of $A(G')$ times $\binom{p}{7}<\binom{q}{4}$ times the (jq) element of $A(G'')$, summed over $1 \le p \le 7$ and $1 \le q \le 4$. In other words, we want to form the matrix product $A(G') \cdot L \cdot A(G'')^T$, where $L_{pq} = \binom{p}{7}<\binom{q}{4}$. This comes to

$$\begin{pmatrix} 21 & 16 & 5 & 0 & 0 & 0 & 0 \\ 0 & 5 & 10 & 12 & 10 & 5 & 0 \\ 21 & 16 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 12 & 18 & 12 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 & 16 & 21 \\ 0 & 5 & 10 & 12 & 10 & 5 & 0 \\ 0 & 0 & 0 & 0 & 5 & 16 & 21 \end{pmatrix} \begin{pmatrix} 210 & 294 & 322 & 329 \\ 126 & 238 & 301 & 325 \\ 70 & 175 & 265 & 315 \\ 35 & 115 & 215 & 295 \\ 15 & 65 & 155 & 260 \\ 5 & 29 & 92 & 204 \\ 1 & 8 & 36 & 120 \end{pmatrix} \begin{pmatrix} 2 & 3 & 0 & 0 \\ 2 & 2 & 0 & 1 \\ 1 & 0 & 2 & 2 \\ 0 & 0 & 3 & 2 \end{pmatrix} = \begin{pmatrix} 48169 & 42042 & 66858 & 64031 \\ 22825 & 16005 & 53295 & 46475 \\ 48169 & 42042 & 66858 & 64031 \\ 22110 & 14850 & 54450 & 47190 \\ 5269 & 2442 & 27258 & 21131 \\ 22825 & 16005 & 53295 & 46475 \\ 5269 & 2442 & 27258 & 21131 \end{pmatrix}.$$

Thus the "best" way to hook up G' and G'' is to compare x_1 with y_2; this gives 42042 cases with $x_1 < y_2$ and $69300 - 42042 = 27258$ cases with $x_1 > y_2$. (By symmetry, we could also compare x_3 with y_2, x_5 with y_3, or x_7 with y_3, leading to essentially the same results.) The efficiency of the resulting graph for $x_1 < y_2$ is

$$\frac{69300}{84084} E(G')E(G''),$$

which is none too good; hence it is probably a bad idea to hook G' up with G'' in any sorting method! The point of this example is that we are able to make such a decision without excessive calculation.

These ideas can be used to provide independent confirmation of Mark Well's proof that $S(12) = 30$. Starting with a graph containing one vertex, we can repeatedly try to add a comparison to one of our graphs G or to a pair of graph components G' and G'' in such a way that the two resulting graphs have 12 or less vertices and efficiency $\ge 12!/2^{29} \approx 0.89221$. Whenever this is possible, we take the resulting graph of least efficiency and add it to our set, unless it is isomorphic to a graph we already have included (or to the dual of such a graph, obtained by reversing the order). If both of the resulting graphs

have the same efficiency, we arbitrarily choose one of them. The first 24 graphs obtained in this way are displayed in Fig. 36 together with their efficiencies.

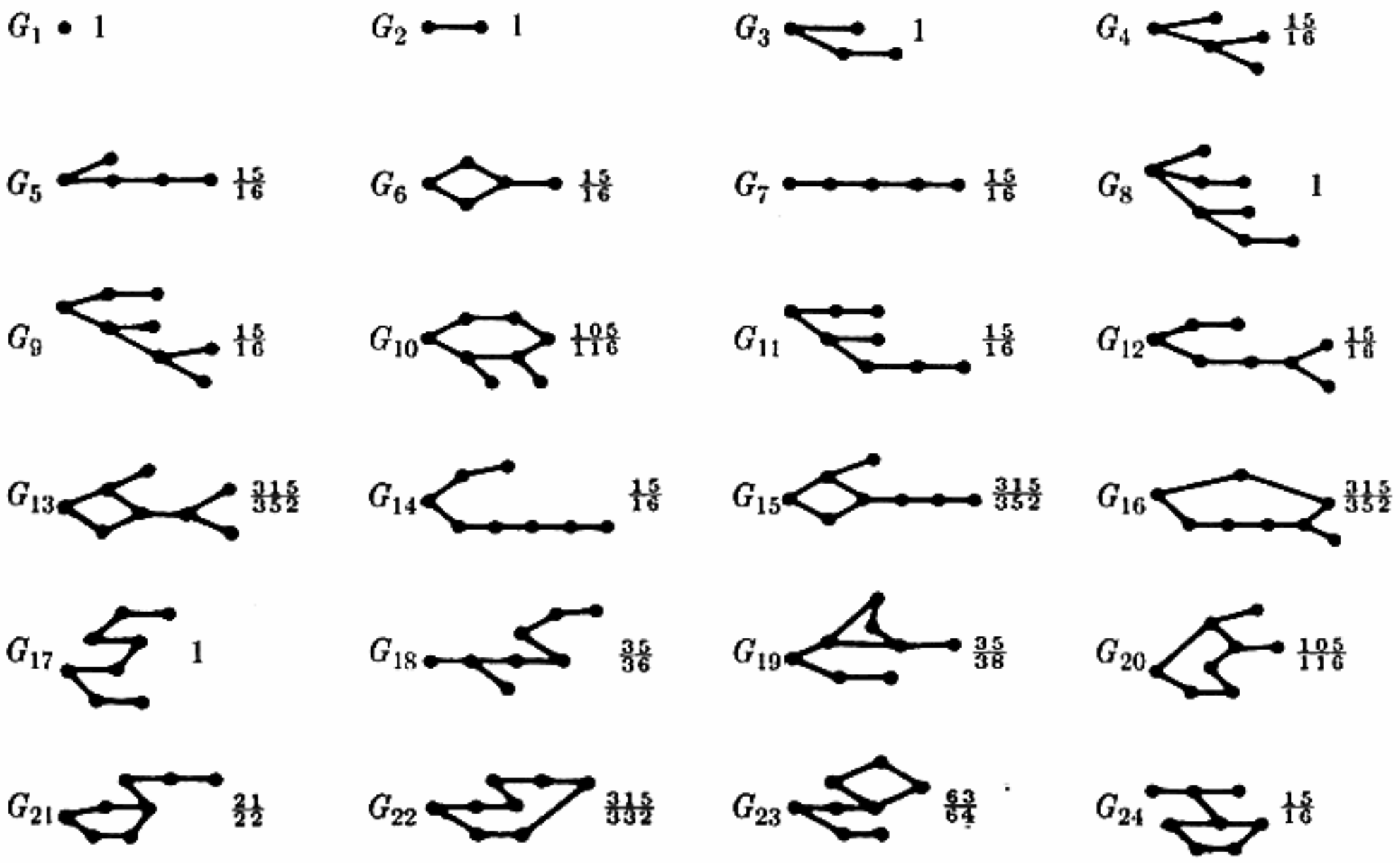


Fig. 36. Some graphs and their efficiencies, obtained at the beginning of a long proof that $S(12) > 29$.

Exactly 1594 graphs were generated, by computer, before this process terminated. Since the graph  was not obtained, we may conclude that $S(12) > 29$. It is plausible that a similar experiment could be performed to deduce that $S(22) > 70$ in a fairly reasonable amount of time, since $22!/2^{70} \approx 0.952$ requires extremely high efficiency to sort in 70 steps. (Only 92 of the 1594 graphs found on 12 or less vertices had such high efficiency.)

The intermediate results suggest strongly that $S(13) = 33$, so that merge insertion would not be optimum when $n = 13$. But as yet nobody has discovered *any* n such that $S(n) < F(n)$.

It should certainly be possible to prove that $S(16) < F(16)$, since $F(16)$ takes no fewer comparisons than sorting ten elements with $S(10)$ comparisons and then inserting six others by binary insertion, one at a time. There must be a way to improve upon such a method!

The average number of comparisons. So far we have been considering procedures which are best possible in the sense that their worst case isn't bad; we have looked for "minimax" procedures which minimize the *maximum* number of comparisons. Now let us look for a "minimean" procedure which minimizes the *average* number of comparisons, assuming that the input is random so that each permutation is equally likely.

Consider once again the tree representation of a sorting procedure, as shown in Fig. 34. The average number of comparisons in that tree is

$$\frac{2+3+3+3+3+2}{6} = 2\tfrac{2}{3},$$

averaging over all permutations. In general, the average number of comparisons in a sorting method is the *external path length of the tree* divided by $n!$. (Recall that the external path length is the sum of the distances from the root to each of the external nodes; see Section 2.3.4.5.) It is easy to see from the considerations in Section 2.3.4.5 that the minimum external path length occurs in a binary tree with N external nodes if there are $2^q - N$ external nodes at level $q-1$ and $2N - 2^q$ at level q, where $q = \lceil \log_2 N \rceil$. (The root is at level zero.) The minimum external path length is therefore

$$(q-1)(2^q - N) + q(2N - 2^q) = (q+1)N - 2^q. \tag{33}$$

The minimum path length can also be characterized in another interesting way: *An extended binary tree has minimum external path length if and only if there is a number l such that all external nodes appear on levels l and $l+1$.* (See exercise 20.)

If we set $q = \log_2 N + \theta$, where $0 \le \theta < 1$, the formula for minimum external path length becomes

$$N(\log_2 N + 1 + \theta - 2^\theta). \tag{34}$$

The function $1 + \theta - 2^\theta$ is shown in Fig. 37; for $0 < \theta < 1$ it is positive but very small, never exceeding

$$1 - (1 + \ln \ln 2)/(\ln 2) = 0.08607\ 13320\ 55934+. \tag{35}$$

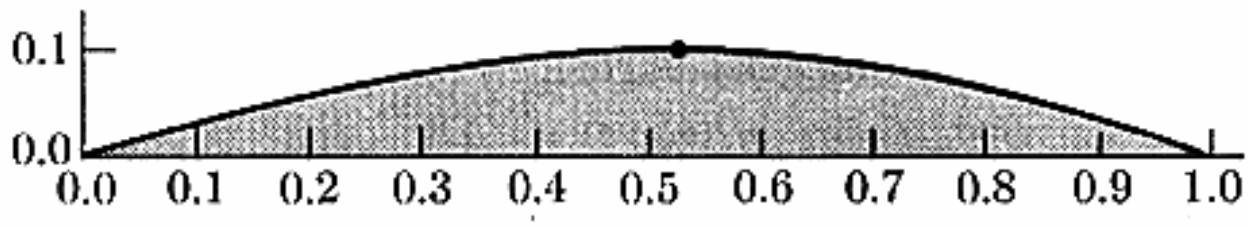


Fig. 37. The function $1 + \theta - 2^\theta$.

Thus the minimum possible average path length, obtained by dividing (34) by N, is never less than $\log_2 N$ and never more than $\log_2 N + 0.0861$. (This result was first obtained by A. Gleason in an unpublished memorandum (1956).)

Now if we set $N = n!$ we get a lower bound for the average number of comparisons in any sorting scheme. Note that this lower bound is $\log_2 n! + O(1) = n \log_2 n - n/(\ln 2) + O(\log n)$.

Let $\overline{F}(n)$ be the average number of comparisons performed by the merge insertion algorithm; we have

$n =$	1	2	3	4	5	6	7	8
lower bound (33) $=$	0	2	16	112	832	6896	62368	619904
$n!\,\overline{F}(n) =$	0	2	16	112	832	6912	62784	623232

Thus merge insertion is optimum in both senses for $n \leq 5$, but for $n = 6$ it averages $6912/720 = 9.6$ comparisons while our lower bound says that an average of $6896/720 = 9.577777\cdots$ comparisons may be possible. A moment's reflection shows why this is true: Some "fortunate" permutations of six elements are sorted after only eight comparisons by merge insertion, so the comparison tree has external nodes appearing on three levels instead of two. This forces the overall path length to be higher. Exercise 24 shows that it is possible to construct a six-element sorting procedure which requires nine or ten comparisons in each case; it follows that this method is superior to merge insertion, on the average, and no worse than merge insertion in its worst case.

When $n = 7$, Y. Césari [thesis (Paris, 1968), p. 37] has shown that no sorting method can attain the lower bound 62368 on external path length. (It is possible to prove this fact by hand, using the results of exercise 22.) On the other hand, he has constructed procedures which do achieve the lower bound (33) when $n = 9$ or 10. In general, the problem of minimizing the average number of comparisons turns out to be substantially more difficult than the problem of determining $S(n)$. It may even be true that, for some n, all methods which minimize the *average* number of comparisons require *more* than $S(n)$ comparisons in their worst case.

EXERCISES

1. [*20*] Draw the comparison trees for sorting four elements by the method of (a) binary insertion; (b) straight two-way merging. What are the external path lengths of these trees?

2. [*M24*] Prove that $B(n) \leq L(n)$, and find all n for which equality holds.

3. [*M22*] When equality between keys is allowed, there are 13 possible outcomes when sorting three elements:

$$K_1 = K_2 = K_3 \qquad K_1 = K_2 < K_3 \qquad K_1 = K_3 < K_2$$
$$K_2 = K_3 < K_1 \qquad K_1 < K_2 = K_3 \qquad K_2 < K_1 = K_3$$
$$K_3 < K_1 = K_2 \qquad K_1 < K_2 < K_3 \qquad K_1 < K_3 < K_2$$
$$K_2 < K_1 < K_3 \qquad K_2 < K_3 < K_1 \qquad K_3 < K_1 < K_2 \qquad K_3 < K_2 < K_1.$$

Let P_n denote the number of possible outcomes when n elements are sorted with ties allowed, so that $(P_0, P_1, P_2, P_3, P_4, P_5, \ldots) = (1, 1, 3, 13, 75, 541, \ldots)$. Prove that the generating function $P(z) = \sum_{n\geq 0} P_n z^n/n!$ is equal to $1/(2 - e^z)$. *Hint:* Show that

$$P_n = \sum_{k>0} \binom{n}{k} P_{n-k} \qquad \text{when} \qquad n > 0.$$

4. [*HM27*] (O. A. Gross.) Determine the asymptotic value of the numbers P_n of exercise 3, as $n \to \infty$. [*Possible hint:* Consider the partial fraction expansion of cot z.]

5. [*16*] When keys can be equal, each comparison may have three results instead of two: $K_i < K_j$, $K_i = K_j$, $K_i > K_j$. Sorting algorithms for this general situation can be represented as extended *ternary* trees, in which each internal node $i{:}j$ has three subtrees; the left, middle, and right subtrees correspond respectively to the three possible outcomes of the comparison.

Draw an extended ternary tree which defines a sorting algorithm for $n = 3$, when equal keys are allowed. There should be 13 external nodes, corresponding to the 13 possible outcomes listed in exercise 3.

▸ **6.** [*M22*] Let $S'(n)$ be the minimum number of comparisons necessary to sort n elements and to determine all equalities between keys, when each comparison has three outcomes as in exercise 5. The "information theory" argument of the text can readily be generalized to show that $S'(n) \geq \lceil \log_3 P_n \rceil$, where P_n is the function studied in exercises 3 and 4; but prove that, in fact, $S'(n) = S(n)$.

7. [*20*] Draw an extended ternary tree in the sense of exercise 5 for sorting four elements, when it is known that all keys are either 0 or 1. (Thus if $K_1 < K_2$ and $K_3 < K_4$, we know that $K_1 = K_3$ and $K_2 = K_4$!) Use the minimum average number of comparisons, assuming that the 2^4 possible inputs are equally likely.

8. [*26*] Draw an extended ternary tree as in exercise 7, for sorting four elements, when it is known that all keys are either -1, 0, or $+1$. Use the minimum average number of comparisons, assuming that the 3^4 possible inputs are equally likely.

9. [*M20*] When sorting n elements as in exercise 7, knowing that all keys are 0 or 1, what is the minimum number of comparisons in the worst case?

▸ **10.** [*M25*] When sorting n elements as in exercise 7, knowing that all keys are 0 or 1, what is the minimum *average* number of comparisons as a function of n?

11. [*HM25*] When sorting n elements as in exercise 5, and knowing that all keys are members of the set $\{1, 2, \ldots, m\}$, let $S_m(n)$ be the minimum number of comparisons needed in the worst case. [Thus by exercise 6, $S_n(n) = S(n)$.] Prove that, for fixed m, $S_m(n)$ is asymptotically $n \log_2 m$ as $n \to \infty$.

▸ **12.** [*M25*] (W. G. Bouricius, circa 1954.) Suppose that equal keys may occur, but we merely want to sort the elements $\{K_1, K_2, \ldots, K_n\}$ so that a permutation $a_1\, a_2 \ldots a_n$ is determined with $K_{a_1} \leq K_{a_2} \leq \cdots \leq K_{a_n}$; we do not need to know whether or not equality occurs between K_{a_i} and $K_{a_{i+1}}$.

Let us say that a comparison tree sorts a sequence of keys *strongly* if it will sort the sequence in the above sense no matter which branch is taken below the nodes $i{:}j$ for which $K_i = K_j$. (The tree is binary, not ternary.)

a) Prove that a comparison tree with no redundant comparisons sorts every sequence of keys strongly if and only if it sorts every sequence of distinct keys.

b) Prove that a comparison tree sorts every sequence of keys strongly if and only if it sorts every sequence of zeros and ones strongly.

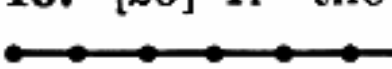
13. [*M28*] Prove (17).

14. [*M24*] Find a "closed form" for the sum (19).

15. [*M21*] Determine the asymptotic behavior of $B(n)$ and $F(n)$ up to $O(\log n)$. [*Hint:* Show that in both cases the coefficient of n involves the function shown in Fig. 37.]

16. [*HM26*] (F. Hwang and S. Lin.) Prove that $F(n) > \lceil \log_2 n! \rceil$ for $n \geq 22$.

17. [*M20*] Prove (29).

18. [*20*] If the procedure which starts in Fig. 36 had produced the graph  with efficiency $12!/2^{29}$, would this have proved that $S(12) = 29$?

19. [*40*] Experiment with the following heuristic rule for deciding which pair of elements to compare next while designing a comparison tree: At each stage of sorting $\{K_1, \ldots, K_n\}$, let u_i be the number of keys known to be $\leq K_i$ as a result of the comparisons made so far, and let v_i be the number of keys known to be $\geq K_i$, for $1 \leq i \leq n$. Renumber the keys in terms of increasing u_i/v_i, so that $u_1/v_1 \leq u_2/v_2 \leq \cdots \leq u_n/v_n$. Now compare $K_i{:}K_{i+1}$ for some i which minimizes $|u_i v_{i+1} - u_{i+1} v_i|$. (Although this method is based on far less information than a full comparison matrix as in (24), it appears to give optimum results in many cases.)

▸**20.** [*M26*] Prove that an extended binary tree has minimum external path length if and only if there is a number l such that all external nodes appear on levels l and $l+1$ (or perhaps all on a single level l).

21. [*M21*] The *height* of an extended binary tree is the maximum level number of its external nodes. If x is an internal node of an extended binary tree, let $t(x)$ be the number of external nodes below x, and let $l(x)$ denote the root of x's left subtree. If x is an external node, let $t(x) = 1$. Prove that an extended binary tree has minimum height among all binary trees with the same number of nodes if and only if

$$|t(x) - 2t(l(x))| \leq 2^{\lceil \log_2 t(x) \rceil} - t(x)$$

for all internal nodes x.

22. [*M24*] Continuing exercise 21, prove that a binary tree has minimum external path length among all binary trees with the same number of nodes if and only if

$$|t(x) - 2t(l(x))| \leq 2^{\lceil \log_2 t(x) \rceil} - t(x) \qquad \text{and} \qquad |t(x) - 2t(l(x))| \leq t(x) - 2^{\lfloor \log_2 t(x) \rfloor}$$

for all internal nodes x. [Thus, for example, if $t(x) = 67$, we must have $t(l(x)) = 32$, 33, 34, or 35. If we merely wanted to minimize the height of the tree we could have $3 \leq t(l(x)) \leq 64$, by the preceding exercise.]

23. [*10*] The text proves [cf. (34)] that the average number of comparisons made by any sorting method for n elements must be at least $\lceil \log_2 n! \rceil \approx n \log_2 n$. But multiple list insertion (Algorithm 5.2.1M) takes only $O(n)$ units of time on the average. How can this be?

24. [*27*] (C. Picard.) Find a sorting tree for six elements such that all external nodes appear on levels 10 and 11.

25. [*11*] If there were a sorting procedure for seven elements which achieves the minimum average number of comparisons predicted by the use of Eq. (34), how many external nodes would there be on level 13?

26. [*M42*] Find a sorting procedure for seven elements which minimizes the average number of comparisons performed.

▸**27.** [*20*] Suppose it is known that the configurations ($K_1 < K_2 < K_3$, $K_1 < K_3 < K_2$, $K_2 < K_1 < K_3$, $K_2 < K_3 < K_1$, $K_3 < K_1 < K_2$, $K_3 < K_2 < K_1$) occur with respective probabilities (.01, .25, .01, .24, .25, .24). Find a comparison tree which sorts these three elements with the smallest average number of comparisons.

28. [*40*] Write a MIX program which sorts five one-word keys in the minimum possible amount of time, and halts. (See the beginning of Section 5.2 for ground rules.)

29. [*M25*] (S. M. Chase.) Let $a_1 a_2 \ldots a_n$ be a permutation of $\{1, 2, \ldots, n\}$. Prove that any algorithm which decides whether this permutation is even or odd (i.e., whether it has an even or odd number of inversions), based solely on comparisons between the a's, must make at least $n \log_2 n$ comparisons, even though the algorithm has only two possible outcomes.

30. [*M23*] (*Optimum exchange sorting.*) Every exchange sorting algorithm as defined in Section 5.2.2 can be represented as a *comparison-exchange tree*, namely a binary tree structure whose internal nodes have the form $\textcircled{i:j}$ for $i < j$, interpreted as the following operation: "If $K_i \leq K_j$, continue by taking the left branch of the tree; if $K_i > K_j$, continue by interchanging records i and j and then taking the right branch of the tree." When an external node is encountered, it must be true that $K_1 \leq K_2 \leq \cdots \leq K_n$. Thus, a comparison-exchange tree differs from a comparison tree in that it specifies data movement as well as comparison operations.

Let $S_e(n)$ denote the minimum number of comparison-exchanges needed, in the worst case, to sort n elements by means of a comparison-exchange tree. Prove that $S_e(n) \leq S(n) + n - 1$.

31. [*M38*] Continuing exercise 30, prove that $S_e(5) = 8$.

32. [*M42*] Continuing exercise 31, investigate $S_e(n)$ for small values of $n > 5$.

33. [*M30*] (T. N. Hibbard.) A *real-valued search tree* of order x and resolution δ is an extended binary tree in which all nodes contain a nonnegative real value, such that (i) the value in each external node is $\leq \delta$; (ii) the value in each internal node is $\leq$ the sum of the values in its two sons; (iii) the value in the root is x. The *weighted path length* of such a tree is defined to be the sum, over all external nodes, of the level of that node times the value it contains.

Prove that a real-valued search tree of order x and resolution 1 has minimum weighted path length, taken over all such trees of the same order and resolution, if and only if equality holds in (ii) and the following further conditions hold for all pairs of values x_0 and x_1 that are contained in brother nodes: (iv) There is no integer $k \geq 0$ such that $x_0 < 2^k < x_1$ or $x_1 < 2^k < x_0$; (v) $\lceil x_0 \rceil - x_0 + \lceil x_1 \rceil - x_1 < 1$. (In particular if x is an integer, condition (v) implies that all values in the tree are integers, and condition (iv) is equivalent to the result of exercise 22.)

Also prove that the corresponding minimum weighted path length is $x\lceil \log_2 x \rceil + \lceil x \rceil - 2^{\lceil \log_2 x \rceil}$.

34. [*M50*] Determine the exact value of $S(n)$ for infinitely many n.

*5.3.2. Minimum-Comparison Merging

Let us now consider a related question: What is the best way to merge an ordered set of m elements with an ordered set of n? Denoting the elements to be merged by

$$A_1 < A_2 < \cdots < A_m,$$
$$B_1 < B_2 < \cdots < B_n, \tag{1}$$

we shall assume as in Section 5.3.1 that the $m + n$ elements are distinct. The

A's may appear among the B's in $\binom{m+n}{m}$ ways, so the arguments we have used for the sorting problem tell us immediately that at least

$$\left\lceil \log_2 \binom{m+n}{m} \right\rceil \tag{2}$$

comparisons are required. If we set $m = \alpha n$ and let $n \to \infty$, while α is fixed, Stirling's approximation tells us that

$$\log_2 \binom{\alpha n + n}{\alpha n} = n((1+\alpha)\log_2 (1+\alpha) - \alpha \log_2 \alpha) - \tfrac{1}{2}\log_2 n + O(1). \tag{3}$$

The normal merging procedure, Algorithm 5.2.4M, takes $m + n - 1$ comparisons in its worst case.

Let $M(m, n)$ denote the function analogous to $S(n)$, namely the minimum number of comparisons which will always suffice to merge m things with n. By the observations we have just made,

$$\left\lceil \log_2 \binom{m+n}{m} \right\rceil \le M(m, n) \le m + n - 1 \qquad \text{for all} \qquad m, n \ge 1. \tag{4}$$

Formula (3) shows how far apart this lower bound and upper bound can be. When $\alpha = 1$ (i.e., $m = n$), the lower bound is $2n - \frac{1}{2}\log_2 n + O(1)$, so both bounds have the right order of magnitude but the difference between them can be arbitrarily large. When $\alpha = 0.5$ (i.e., $m = \frac{1}{2}n$), the lower bound is

$$\tfrac{3}{2}n(\log_2 3 - \tfrac{2}{3}) + O(\log n),$$

which is about $\log_2 3 - \frac{2}{3} \approx 0.918$ times the upper bound. And as α decreases, the bounds get farther and farther apart, since the standard merging algorithm is primarily designed for files with $m \approx n$.

When $m = n$, the merging problem has a fairly simple solution; it turns out that the *lower* bound of (4), not the upper bound, is at fault. The following theorem was discovered independently by R. L. Graham and R. M. Karp about 1968:

Theorem M. $M(m, m) = 2m - 1$, *for* $m \ge 1$.

Proof. Consider any algorithm which merges $A_1 < \cdots < A_m$ with $B_1 < \cdots < B_m$. When it compares $A_i{:}B_j$, take the branch $A_i < B_j$ if $i < j$, the branch $A_i > B_j$ if $i \ge j$. Merging must eventually terminate with the configuration

$$B_1 < A_1 < B_2 < A_2 < \cdots < B_m < A_m, \tag{5}$$

since this is consistent with all of the branches taken. And each of the $2m - 1$ comparisons $B_1{:}A_1$, $A_1{:}B_2$, $B_2{:}A_2, \ldots, B_m{:}A_m$ must have been made explicitly, or else there would be at least two configurations consistent with

the known facts. For example, if A_1 has not been compared to B_2, the configuration

$$B_1 < B_2 < A_1 < A_2 < \cdots < B_m < A_m$$

is indistinguishable from (5). ∎

A simple modification of this proof yields the companion formula

$$M(m, m+1) = 2m, \qquad \text{for} \qquad m \geq 0. \tag{6}$$

Oracular lower bounds. Theorem M shows that the "information theoretic" lower bound can be considerably lower than the true lower bound; thus the technique used to prove Theorem M gives us another way to discover lower bounds. It is convenient to regard this proof technique as the construction of an "oracle": When merging by some algorithm, we decide to compare A_i:B_j and we say, "Pray tell me, O thou wise Oracle, which is the larger, A_i or B_j?" After we have made a suitable offering, the answer comes back:

I am Sir Oracle, and I ope my lips to say that A_i is the greater.

If we can construct a suitable oracle, as in the proof of Theorem M, we can ensure that every valid merging algorithm will have to ask a rather large number of questions.

It turns out to be important to consider the idea of a *constrained oracle*, which has been enjoined to give specific answers to certain questions. (The merging method which consults the oracle does not know, of course, that the oracle is constrained, so it may be necessary to ask the question anyway.) The constraints we speak of apply to the left and right ends of the files. Left constraints are symbolized by

. (meaning no left constraint),
\ (meaning that all pronouncements must be consistent with $A_1 < B_1$),
/ (meaning that all pronouncements must be consistent with $A_1 > B_1$);

right constraints are symbolized by

. (meaning no right constraint),
\ (meaning that all pronouncements must be consistent with $A_m < B_n$),
/ (meaning that all pronouncements must be consistent with $A_m > B_n$).

There are nine kinds of oracles, denoted by $\nabla M \phi$, where ∇ is a left constraint and ϕ is a right constraint. For example, a "$\backslash M \backslash$" oracle must say that $A_1 < B_j$ and $A_i < B_n$; a "$.M.$" oracle is unconstrained. For some small m and n, constrained oracles of certain kinds are impossible; when $m = 1$ we obviously can't have a "$\backslash M /$" oracle.

Let us now construct a rather complicated, but very wise, oracle for merging. It does not always produce optimum results, but it gives lower bounds

which cover a lot of interesting cases. Given m, n, and the left and right constraints ∇ and ϕ, suppose the oracle is asked which is the greater of A_i or B_j. The oracle can use six strategies in general which reduce the problem to cases of smaller $m + n$:

Strategy $A(k, l)$, *for* $i \leq k \leq m$ *and* $1 \leq l \leq j$. Say that $A_i < B_j$, and require that subsequent operations merge $\{A_1, \ldots, A_k\}$ with $\{B_1, \ldots, B_{l-1}\}$ and $\{A_{k+1}, \ldots, A_m\}$ with $\{B_l, \ldots, B_n\}$. Thus future questions $A_p{:}B_q$ will be answered $A_p < B_q$ if $p \leq k$ and $q \geq l$; $A_p > B_q$ if $p > k$ and $q < l$; they will be handled by a $(k, l-1, \nabla, .)$ oracle if $p \leq k$ and $q < l$; by an $(m-k, n+1-l, ., \phi)$ oracle if $p > k$ and $q \geq l$.

Strategy $B(k, l)$, *for* $i \leq k \leq m$ *and* $1 \leq l < j$. Say that $A_i < B_j$, and require that subsequent operations merge $\{A_1, \ldots, A_k\}$ with $\{B_1, \ldots, B_l\}$ and $\{A_{k+1}, \ldots, A_m\}$ with $\{B_l, \ldots, B_n\}$, stipulating that $A_k < B_l < A_{k+1}$. (Note that B_l appears in both lists to be merged. The condition $A_k < B_l < A_{k+1}$ ensures that merging one group gives no information that could help to merge the other.) Thus future questions $A_p{:}B_q$ will be answered $A_p < B_q$ if $p \leq k$ and $q \geq l$; $A_p > B_q$ if $p > k$ and $q \leq l$; they will be handled by a $(k, l, \nabla, \backslash)$ oracle if $p \leq k$ and $q \leq l$; by an $(m-k, n+1-l, /, \phi)$ oracle if $p > k$ and $q \geq l$.

Strategy $C(k, l)$ *for* $i < k \leq m$ *and* $1 \leq l \leq j$. Say that $A_i < B_j$, and require that subsequent operations merge $\{A_1, \ldots, A_k\}$ with $\{B_1, \ldots, B_{l-1}\}$ and $\{A_k, \ldots, A_m\}$ with $\{B_l, \ldots, B_n\}$, stipulating that $B_{l-1} < A_k < B_l$. (Analogous to Strategy B, interchanging the roles of A, B.)

Strategy $A'(k, l)$ *for* $1 \leq k \leq i$ *and* $j \leq l \leq n$. Say that $A_i > B_j$ and require the merging of $\{A_1, \ldots, A_{k-1}\}$ with $\{B_1, \ldots, B_l\}$ and $\{A_k, \ldots, A_m\}$ with $\{B_{l+1}, \ldots, B_n\}$. (Analogous to Strategy A.)

Strategy $B'(k, l)$ *for* $1 \leq k \leq i$ *and* $j < l \leq n$. Say that $A_i > B_j$, and require the merging of $\{A_1, \ldots, A_{k-1}\}$ with $\{B_1, \ldots, B_l\}$ and $\{A_k, \ldots, A_m\}$ with $\{B_l, \ldots, B_n\}$, subject to $A_{k-1} < B_l < A_k$. (Analogous to Strategy B.)

Strategy $C'(k, l)$ *for* $1 \leq k < i$ *and* $j \leq l \leq n$. Say that $A_i > B_j$, and require the merging of $\{A_1, \ldots, A_k\}$ with $\{B_1, \ldots, B_l\}$ and $\{A_k, \ldots, A_m\}$ with $\{B_{l+1}, \ldots, B_n\}$, subject to $B_l < A_k < B_{l+1}$. (Analogous to Strategy C.)

Because of the constraints, the above strategies cannot be used in certain cases summarized here:

Strategy	Must be omitted when
$A(k, 1)$, $B(k, 1)$, $C(k, 1)$	$\nabla = /$
$A'(1, l)$, $B'(1, l)$, $C'(1, l)$	$\nabla = \backslash$
$A(m, l)$, $B(m, l)$, $C(m, l)$	$\phi = /$
$A'(k, n)$, $B'(k, n)$, $C'(k, n)$	$\phi = \backslash$

Let $\nabla M\phi(m, n)$ denote the maximum lower bound for merging which is obtainable by an oracle of the class described above. Each strategy, when applicable, gives us an inequality relating these nine functions, when the first comparison is $A_i{:}B_j$, namely,

$$A(k, l): \quad \nabla M\phi(m, n) \geq 1 + \nabla M.(k, l-1) + .M\phi(m-k, n+1-l);$$
$$B(k, l): \quad \nabla M\phi(m, n) \geq 1 + \nabla M\backslash(k, l) + /M\phi(m-k, n+1-l);$$
$$C(k, l): \quad \nabla M\phi(m, n) \geq 1 + \nabla M/(k, l-1) + \backslash M\phi(m+1-k, n+1-l);$$
$$A'(k, l): \quad \nabla M\phi(m, n) \geq 1 + \nabla M.(k-1, l) + .M\phi(m+1-k, n-l);$$
$$B'(k, l): \quad \nabla M\phi(m, n) \geq 1 + \nabla M\backslash(k-1, l) + /M\phi(m+1-k, n+1-l);$$
$$C'(k, l): \quad \nabla M\phi(m, n) \geq 1 + \nabla M/(k, l) + \backslash M\phi(m+1-k, n-l).$$

For fixed i and j, the oracle will adopt a strategy which maximizes the lower bound given by the righthand side; then we define $\nabla M\phi(m, n)$ to be the minimum of these lower bounds taken over $1 \leq i \leq m$ and $1 \leq j \leq n$. When m or n is zero, $\nabla M\phi(m, n)$ is zero.

For example, suppose that $m = 2$ and $n = 3$, and our oracle is unconstrained. If the first comparison is $A_1{:}B_1$, the oracle may adopt strategy $A'(1, 1)$, requiring $.M.(0, 1) + .M.(2, 2) = 3$ further comparisons. If the first comparison is $A_1{:}B_3$, the oracle may adopt strategy $B(1, 2)$, requiring $.M\backslash(1, 2) + /M.(1, 2) = 4$ further comparisons. No matter what comparison $A_i{:}B_j$ is made first, the oracle can guarantee that at least three further comparisons must be made. Hence $.M.(2, 3) = 4$.

It isn't easy to do these calculations by hand, but a computer can grind out tables of the $\nabla M\phi$ functions rather quickly. There are some obvious symmetries, e.g.,

$$/M.(m, n) = .M\backslash(m, n) = \backslash M.(n, m) = .M/(n, m), \tag{7}$$

by means of which we can reduce the nine functions to just four,

$$.M.(m, n), \qquad /M.(m, n), \qquad /M\backslash(m, n), \qquad \text{and} \qquad /M/(m, n).$$

Table 1 shows the resulting values for all $m, n \leq 10$; our merging oracle has been defined in such a way that

$$.M.(m, n) \leq M(m, n) \qquad \text{for all} \qquad m, n \geq 0. \tag{8}$$

This relation includes Theorem M as a special case, because our oracle will use the simple strategy of that theorem when $|m - n| \leq 1$.

Let us now consider some simple relations satisfied by the M function:

$$M(m, n) = M(n, m); \tag{9}$$
$$M(m, n) \leq M(m, n+1); \tag{10}$$
$$M(k+m, n) \leq M(k, n) + M(m, n); \tag{11}$$

Table 1

LOWER BOUNDS FOR MERGING, FROM THE "ORACLE"

$.M.(m, n)$

m \ n	1	2	3	4	5	6	7	8	9	10
1	1	2	2	3	3	3	3	4	4	4
2	2	3	4	5	5	6	6	6	7	7
3	2	4	5	6	7	7	8	8	9	9
4	3	5	6	7	8	9	10	10	11	11
5	3	5	7	8	9	10	11	12	12	13
6	3	6	7	9	10	11	12	13	14	15
7	3	6	8	10	11	12	13	14	15	16
8	4	6	8	10	12	13	14	15	16	17
9	4	7	9	11	12	14	15	16	17	18
10	4	7	9	11	13	15	16	17	18	19

$/M.(m, n)$

m \ n	1	2	3	4	5	6	7	8	9	10
1	1	2	2	3	3	3	3	4	4	4
2	1	3	4	4	5	5	6	6	7	7
3	1	3	5	6	7	7	8	8	9	9
4	1	4	5	7	8	9	9	10	10	11
5	1	4	6	8	9	10	11	12	12	13
6	1	4	6	8	10	11	12	13	14	14
7	1	4	7	9	10	12	13	14	15	16
8	1	5	7	9	11	13	14	15	16	17
9	1	5	8	10	11	13	15	16	17	18
10	1	5	8	10	12	14	15	17	18	19

$/M\backslash(m, n)$

m \ n	1	2	3	4	5	6	7	8	9	10
1	$-\infty$	2	2	3	3	3	3	4	4	4
2	$-\infty$	2	4	4	5	5	6	6	7	7
3	$-\infty$	2	4	6	6	7	8	8	8	9
4	$-\infty$	2	5	6	8	8	9	10	10	11
5	$-\infty$	2	5	7	8	10	10	11	12	13
6	$-\infty$	2	5	7	9	10	12	13	14	14
7	$-\infty$	2	5	8	10	11	12	14	15	16
8	$-\infty$	2	6	8	10	12	13	15	16	17
9	$-\infty$	2	6	9	10	12	14	16	17	18
10	$-\infty$	2	6	9	11	13	15	16	18	19

$/M/(m, n)$

m \ n	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1
2	1	3	3	4	4	4	4	5	5	5
3	1	3	5	5	6	6	7	7	8	8
4	1	4	5	7	7	8	9	9	9	10
5	1	4	6	7	9	9	10	11	11	12
6	1	4	6	8	9	11	11	12	13	14
7	1	4	7	9	10	11	13	14	15	15
8	1	5	7	9	11	12	14	15	16	17
9	1	5	8	9	11	13	15	16	17	18
10	1	5	8	10	12	14	15	17	18	19

$$M(m, n) \leq \max\left(M(m, n-1) + 1, M(m-1, n) + 1\right); \tag{12}$$

$$M(m, n) \leq \max\left(M(m, n-2) + 1, M(m-1, n) + 2\right), \text{ for } m \geq 1, n \geq 2. \tag{13}$$

Relation (12) comes from the usual merging procedure, if we first compare $A_1 : B_1$. Relation (13) is derived similarly, by first comparing $A_1 : B_2$; if $A_1 > B_2$, we need $M(m, n-2)$ more comparisons, but if $A_1 < B_2$, we can insert A_1 into its proper place and merge $\{A_2, \ldots, A_m\}$ with $\{B_1, \ldots, B_n\}$. Generalizing, we can see that

$$M(m, n) \leq \max\left(M(m, n-k) + 1, M(m-1, n) + 1 + \lceil \log_2 k \rceil\right),$$
$$\text{for} \qquad m \geq 1, \qquad n \geq k, \tag{14}$$

by first comparing $A_1 : B_k$ and using binary search if $A_1 < B_k$.

It turns out that $M(m, n) = .M.(m, n)$ for all $m, n \leq 10$, so Table 1 actually gives the optimum values for merging. This can be proved by using (9)–(14) together with special constructions for $(m, n) = (2, 8)$, $(3, 6)$, and $(5, 9)$ which are given in exercises 8, 9, and 10.

On the other hand, our oracle doesn't always give the best possible lower bounds; the simplest example is $m = 3$, $n = 11$, when $.M.(3, 11) = 9$ but $M(3, 11) = 10$. To see where the oracle has "failed" in this case, we must study the reasons for its decisions; further scrutiny reveals that if $(i, j) \neq (2, 6)$, the oracle can find a strategy which demands 10 comparisons; but when $(i, j) = (2, 6)$, no strategy beats Strategy $A(2, 4)$, leading to the lower bound $1 + .M.(2, 3) + .M.(1, 8) = 9$. It is necessary but not sufficient to finish by merging $\{A_1, A_2\}$ with $\{B_1, B_2, B_3\}$ and $\{A_3\}$ with $\{B_4, \ldots, B_{11}\}$, so the lower bound fails to be sharp in this case.

Similarly it can be shown that $.M.(2, 38) = 10$ while $M(2, 38) = 11$, so our oracle isn't good enough to solve the case $m = 2$. But there is an infinite class of values for which it excels:

Theorem K. $M(m, m + 2) = 2m + 1$ *for* $m \geq 2$;
$M(m, m + 3) = 2m + 2$ *for* $m \geq 4$;
$M(m, m + 4) = 2m + 3$ *for* $m \geq 6$.

Proof. We can in fact prove the result with M replaced by $.M.$; for small m the results have been obtained by computer, so we may assume that m is sufficiently large. We may also assume that the first comparison is $A_i:B_j$ where $i \leq \lceil m/2 \rceil$. If $j \leq i$ we use strategy $A'(i, i)$, obtaining

$$.M.(m, m+d) \geq 1 + .M.(i-1, i) + .M.(m+1-i, m+d-i) = 2m + d - 1$$

by induction on d, $d \leq 4$. If $j > i$ we use strategy $A(i, i + 1)$, obtaining

$$.M.(m, m + d) \geq 1 + .M.(i, i) + .M.(m - i, m + d - i) = 2m + d - 1$$

by induction on m. ∎

The first two parts of Theorem K were obtained by F. Hwang and S. Lin in 1969. This proof makes it reasonable to conjecture that $M(m, m + d) = 2m + d - 1$ for all sufficiently large m, when d is fixed. (Cf. exercise 6.)

Upper bounds. Now let us consider *upper* bounds for $M(m, n)$; good upper bounds correspond to efficient merging algorithms.

When $m = 1$ the merging problem is equivalent to an insertion problem, and there are $n + 1$ places in which A_1 might fall among $B_1, \ldots, B_n$. For this case it is easy to see that *any* extended binary tree with $n + 1$ external nodes is the tree for some merging method! (See exercise 2.) Hence we may choose an optimum binary tree, realizing the information-theoretic lower bound

$$1 + \lfloor \log_2 n \rfloor = M(1, n) = \lceil \log_2 (n + 1) \rceil. \tag{15}$$

Binary search (Section 6.2.1) is, of course, a simple way to attain this value.

The case $m = 2$ is extremely interesting, but considerably harder. It has

been solved completely by R. L. Graham, F. K. Hwang, and S. Lin (see exercises 11, 12, 13); we have

$$M(2, n) = \lceil \log_2 \tfrac{7}{12}(n+1) \rceil + \lceil \log_2 \tfrac{14}{17}(n+1) \rceil. \tag{16}$$

We have seen that the usual merging procedure is optimum when $m = n$, and the rather different binary search procedure is optimum when $m = 1$. What we need is an in-between method which combines the normal merging algorithm with binary search in such a way that the best features of both are retained. Formula (14) suggests the following merging algorithm, due to F. K. Hwang and S. Lin [*SIAM J. Computing* **1** (1972), 31–39]:

Algorithm H (*Binary merging*).

H1. If m or n is zero, stop. If $m \le n$, set $t \leftarrow \lfloor \log_2 (n/m) \rfloor$. If $m > n$, set $t \leftarrow \lfloor \log_2 (m/n) \rfloor$ and go to H4.

H2. Compare $A_m : B_{n+1-2^t}$. If A_m is smaller, set $n \leftarrow n - 2^t$ and return to step H1.

H3. Using binary search (which requires exactly t more comparisons), insert A_m into its proper place among $\{B_{n+1-2^t}, \ldots, B_n\}$. If k is maximal such that $B_k < A_m$, set $m \leftarrow m - 1$ and $n \leftarrow k$. Return to H1.

H4. (Steps H4 and H5 are like H2 and H3, interchanging the roles of m, n, A, B.) If $B_n < A_{m+1-2^t}$, set $m \leftarrow m - 2^t$ and return to step H1.

H5. Insert B_n into its proper place among the A's. If k is maximal such that $A_k < B_n$, set $m \leftarrow k$ and $n \leftarrow n - 1$. Return to H1. ∎

As an example of this algorithm, Table 2 shows the process of merging the three keys {087, 503, 512} with thirteen keys {061, 154, . . . , 908}; eight comparisons are required in this example.

Table 2

EXAMPLE OF BINARY MERGING

A (**boldface** elements are compared)			*B*												Output			
{087	503	512}	{051	154	170	275	426	509	612	653	677	703	765	897	908}			
{087	503	512}	{061	154	170	275	426	509	612	653	677}				{703	765	897	908}
{087	503	512}	{061	154	170	275	426	509	612}				{653	677	703	765	897	908}
{087	503	512}	{061	154	170	275	426	509	612}				{653	677	703	765	897	908}
{087	503}		{061	154	170	275	426	509}			{512	612	653	677	703	765	897	908}
{087	503}		{061	154	170	275	426	509}			{512	612	653	677	793	765	897	908}
{087}			{061	154	170	275	426}		{503	509	512	612	653	677	703	765	897	908}
{087}			{061}		{154	170	275	426	503	509	512	612	653	677	703	765	897	908}
			{061	087	154	170	275	426	503	509	512	612	653	677	703	765	897	908}

Let $H(m, n)$ be the maximum number of comparisons required by Hwang and Lin's algorithm. To calculate $H(m, n)$, we may assume that $k = n$ in step H3 and $k = m$ in step H5, since it is not difficult to prove that $H(m, n) \le H(m, n+1)$ for all $n \ge m$ by induction on n. Thus when $m \le n$ we have

$$H(m, n) = \max\big(H(m, n - 2^t) + 1, \quad H(m - 1, n) + t + 1\big), \quad \text{for} \quad 2^t m \le n < 2^{t+1} m. \tag{17}$$

Replace n by $2n + \epsilon$, $\epsilon = 0$ or 1, to get

$$H(m, 2n + \epsilon) = \max\big(H(m, 2n + \epsilon - 2^{t+1}) + 1, \; H(m - 1, 2n + \epsilon) + t + 2\big), \quad \text{for} \quad 2^t m \le n < 2^{t+1} m;$$

whence it follows by induction on n that

$$H(m, 2n + \epsilon) = H(m, n) + m, \quad \text{for} \quad m \le n, \quad \epsilon = 0 \quad \text{or} \quad 1. \tag{18}$$

It is also easy to see that $H(m, n) = m + n - 1$ when $m \le n < 2m$; hence a repeated application of (18) yields the general formula

$$H(m, n) = m + \lfloor n/2^t \rfloor - 1 + tm, \quad \text{for} \quad m \le n, \quad t = \lfloor \log_2 (n/m) \rfloor. \tag{19}$$

Setting $m = \alpha n$ and $\theta = \log_2 (n/m) - t$ gives

$$H(\alpha n, n) = \alpha n(1 + 2^\theta - \theta - \log_2 \alpha) + O(1), \tag{20}$$

as $n \to \infty$. We know by Eq. 5.3.1–35 that $1.9139 < 1 + 2^\theta - \theta \le 2$; hence (20) may be compared with the information-theoretic lower bound (3). Hwang and Lin have proved (see exercise 17) that

$$H(m, n) < \left\lceil \log_2 \binom{m + n}{m} \right\rceil + \min(m, n). \tag{21}$$

The Hwang-Lin merging algorithm, which might be called "binary merging," does not always give optimum results, but it has the great virtue that it can be programmed rather easily. It reduces to "uncentered binary search" when $m = 1$, and it reduces to the usual merging procedure when $m \approx n$, so it represents an excellent compromise between those two methods. Furthermore, it *is* optimum in many cases (see exercise 16).

Formula (18) suggests that the M function itself might satisfy

$$M(m, n) \le M(m, \lfloor n/2 \rfloor) + m. \tag{22}$$

This is actually true (see exercise 19). Tables of $M(m, n)$ suggest several other plausible relations, such as

$$M(m + 1, n) \ge 1 + M(m, n) \ge M(m, n + 1), \quad \text{for} \quad m \le n; \tag{23}$$

$$M(m + 1, n + 1) \ge 2 + M(m, n); \tag{24}$$

but no proof of these inequalities is known.

EXERCISES

1. [*15*] Find an interesting relation between $M(m, n)$ and the function S defined in Section 5.3.1. [*Hint:* Consider $S(m + n)$.]

▶ **2.** [*22*] When $m = 1$, every merging algorithm without redundant comparisons defines an extended binary tree with $\binom{m+n}{m} = n + 1$ external nodes. Prove that, conversely, every extended binary tree with $n + 1$ external nodes corresponds to some merging algorithm with $m = 1$.

3. [*M24*] Prove that $.M.(1, n) = M(1, n)$ for all n.

4. [*M44*] Is $.M.(m, n) \geq \lceil \log_2 \binom{m+n}{m} \rceil$ for all m and n?

5. [*M30*] Prove that $.M.(m, n) \leq .M\backslash(m, n + 1)$.

6. [*M26*] The stated proof of Theorem K requires that a lot of cases be verified by computer. How can the number of such cases be drastically reduced?

7. [*21*] Prove (11).

▶ **8.** [*24*] Prove that $M(2, 8) \leq 6$, by finding an algorithm which merges two elements with eight others using at most six comparisons.

9. [*27*] Prove that three elements can be merged with six in at most seven steps.

10. [*33*] Prove that five elements can be merged with nine in at most twelve steps. [*Hint:* Experience with the oracle suggests starting by comparing $A_1 : B_2$, then if $A_1 < B_2$ trying $A_5 : B_8$.]

11. [*M40*] (F. Hwang, S. Lin.) Let $g_{2k} = \lfloor 2^k \cdot \frac{17}{14} \rfloor$, $g_{2k+1} = \lfloor 2^k \cdot \frac{12}{7} \rfloor$, for $k \geq 0$, so that we have $(g_0, g_1, g_2, \ldots) = (1, 1, 2, 3, 4, 6, 9, 13, 19, 27, 38, 54, 77, \ldots)$. Prove that it takes more than t comparisons to merge two elements with g_t elements, in the worst case; but two elements can be merged with $g_t - 1$ in at most t steps. [*Hint:* Show that if $n \geq g_{t-1}$ and if we want to merge $\{A_1, A_2\}$ with $\{B_1, B_2, \ldots, B_n\}$ in t comparisons, we can't do better than to compare $A_2 : B_{g_{t-1}}$ on the first step.]

12. [*M21*] Let $R_n(i, j)$ be the least number of comparisons required to sort the distinct objects $\{\alpha, \beta, X_1, X_2, \ldots, X_n\}$, given the relations

$$\alpha < \beta, \qquad X_1 < X_2 < \cdots < X_n, \qquad \alpha < X_{i+1}, \qquad \beta > X_{n-j}.$$

(The condition $\alpha < X_{i+1}$ or $\beta > X_{n-j}$ becomes vacuous when $i \geq n$ or $j \geq n$. Therefore $R_n(n, n) = M(2, n)$.)

Clearly, $R_n(0, 0) = 0$. Prove that

$$R_n(i, j) = 1 + \min\Big(\min_{1 \leq k \leq i} \max\big(R_n(k-1, j), R_{n-k}(i-k, j)\big), \min_{1 \leq k \leq j} \max\big(R_n(i, k-1), R_{n-k}(i, j-k)\big)\Big)$$

for $0 \leq i \leq n$, $0 \leq j \leq n$, $i + j > 0$.

13. [*M42*] (R. L. Graham.) Show that the solution to the recurrence in exercise 12 may be expressed as follows. Define the function $G(x)$, for $0 < x < \infty$, by the rules

$$G(x) = \begin{cases} 1, & 0 < x \leq \frac{5}{7}; \\ \frac{1}{2} + \frac{1}{8}G(8x - 5), & \frac{5}{7} < x \leq \frac{3}{4}; \\ \frac{1}{2}G(2x - 1), & \frac{3}{4} < x \leq 1; \\ 0, & 1 < x < \infty. \end{cases}$$

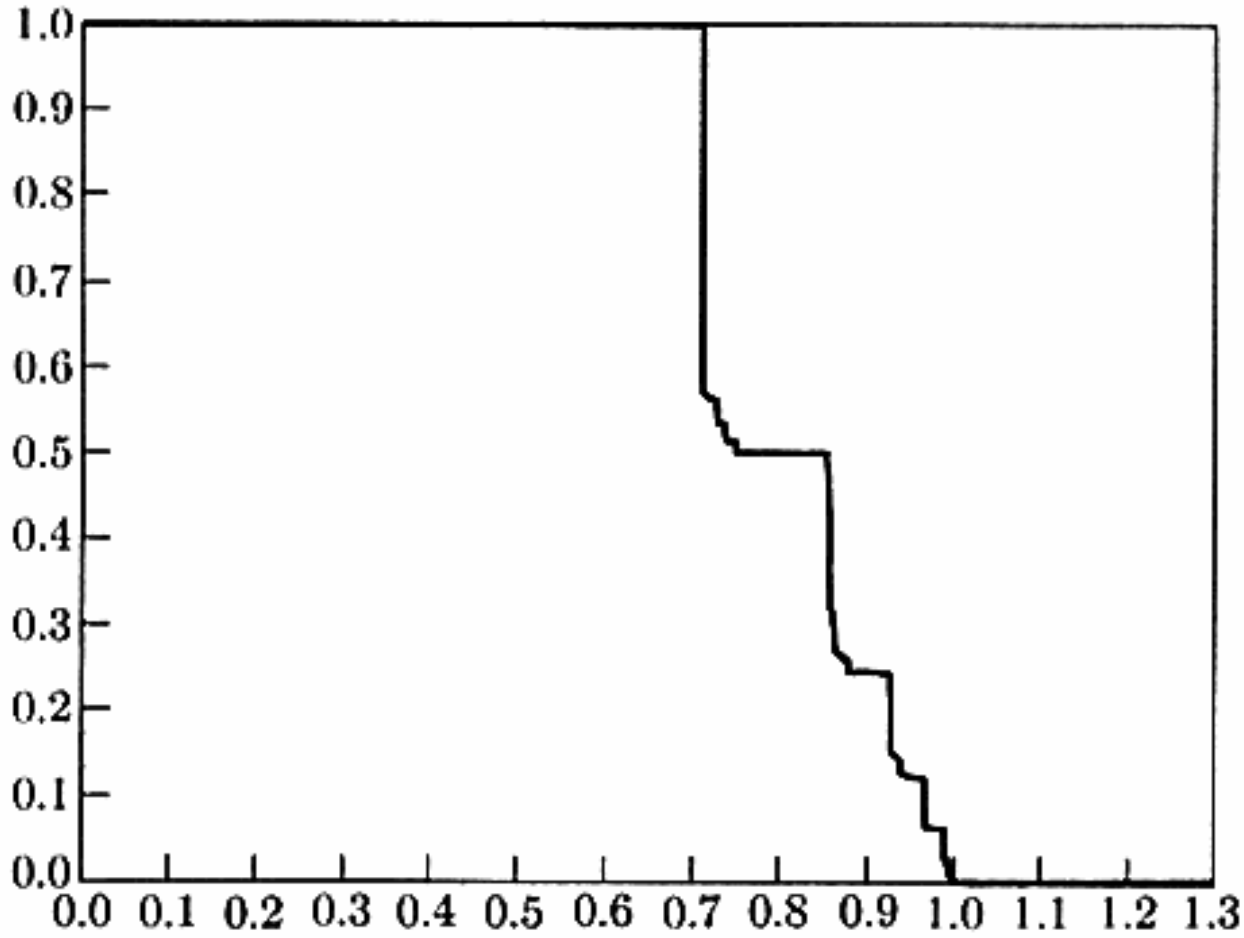


Fig. 38. Graham's function (see exercise 13).

(See Fig. 38.) Since $R_n(i, j) = R_n(j, i)$ and since $R_n(0, j) = M(1, j)$, we may assume that $1 \le i \le j \le n$. Let $p = \lfloor \log_2 i \rfloor$, $q = \lfloor \log_2 j \rfloor$, $r = \lfloor \log_2 n \rfloor$, and let $t = n - 2^r + 1$. Then

$$R_n(i, j) = p + q + S_n(i, j) + T_n(i, j),$$

where S_n and T_n are functions which are either 0 or 1:

$$S_n(i, j) = 1 \quad \text{if and only if } q < r \text{ or } (i - 2^p \ge u \text{ and } j - 2^r \ge u),$$
$$T_n(i, j) = 1 \quad \text{if and only if } p < r \text{ or } (t > \tfrac{6}{7}2^{r-2} \text{ and } i - 2^r \ge v),$$

where $u = 2^p G(t/2^p)$ and $v = 2^{r-2} G(t/2^{r-2})$.

(This may be the most formidable recurrence relation that will ever be solved!)

14. [*46*] (Hwang and Lin.) Let $h_{3k} = 2^k + 2^{k-1} - 1$, $h_{3k+1} = g_{2k} + g_{2k-3} + 2^{k-2}$, $h_{3k+2} = 2g_{2k}$, for $k \ge 2$, except that $h_8 = 9$, with initial values adjusted so that we have $(h_0, h_1, h_2, \ldots) = (1, 1, 2, 2, 3, 4, 5, 7, 9, 11, 14, 18, 23, 29, 38, 47, 59, 76, \ldots)$. Here g_k is the function defined in exercise 11. Prove (or disprove) that $M(3, h_t) > t$, $M(3, h_t - 1) \le t$, for all t.

15. [*12*] Step H1 of the binary merge algorithm may require the calculation of $\lfloor \log_2 (n/m) \rfloor$. Explain how to compute this easily without division or calculation of a logarithm.

16. [*18*] For which m and n is Hwang and Lin's binary merging algorithm optimum, for $1 \le m \le n \le 10$?

17. [*M25*] Prove (21). [*Hint:* The inequality isn't very tight.]

18. [*M40*] Study the *average* number of comparisons used by binary merge.

▶ **19.** [*23*] Prove that the M function satisfies (22).

20. [*20*] Show that if $M(m, n+1) \le M(m+1, n)$ for all $m \le n$, then $M(m, n+1) \le 1 + M(m, n)$ for all $m \le n$.

21. [*M47*] Prove or disprove (23), (24).

22. [*M50*] Study the minimum *average* number of comparisons needed to merge m things with n.

23. [*HM30*] (E. Reingold.) Let $\{A_1, \ldots, A_n\}$ and $\{B_1, \ldots, B_n\}$ be sets containing n elements each. Consider an algorithm which attempts to test *equality* of these two sets solely by making comparisons for equality between elements. Thus, the algorithm asks questions of the form "Is $A_i = B_j$?" for certain i and j, and it branches depending on whether the answer is yes or no.

By defining a suitable oracle, prove that any such algorithm must make at least $\frac{1}{2}n(n+1)$ comparisons in its worst case.

*5.3.3. Minimum-Comparison Selection

A similar class of interesting problems arises when we look for best possible procedures to select the tth largest of n elements.

The history of this question goes back to Rev. C. L. Dodgson's amusing (though serious) essay on lawn tennis tournaments which appeared in *St. James's Gazette*, August 1, 1883, pp. 5–6. Dodgson, who is of course better known as Lewis Carroll, was concerned about the unjust manner in which prizes were (and still are) awarded in tennis tournaments. Consider, for example, Fig. 39 which shows a typical "knockout tournament" between 32 players labeled *01*, *02*, . . . , *32*. In the "finals," player *01* defeats player *05*, so it is clear that player *01* is the champion and he deserves first prize. The inequity arises in connection with the fact that player *05* usually gets second prize, although he may not be the second-best player. It is possible to win second prize even if you are worse than half of the players in the tournament. In fact, as Dodgson observed, the second-best player wins second prize if and only if he and the champion are originally in opposite halves of the tournament; for 2^n players this occurs with probability $2^{n-1}/(2^n - 1)$, so the wrong player receives second prize almost half of the time! If the losers of the semifinal round (players *25* and *17* in Fig. 39) compete for third prize, it is highly unlikely that the third-best player receives third prize.

Dodgson therefore set out to design a tournament that determines the true second- and third-best players, assuming a transitive ranking. (In other words if player A beats player B, and B beats C, then we may assume that A would beat C.) He devised a procedure in which losers are allowed to play further games until they are known to be definitely inferior to three other players. An example of Dodgson's scheme appears in Fig. 40, which is a supplementary tournament to be run in conjunction with Fig. 39. An attempt is made to pair off players with equal records so far, and to avoid matches in which both players have been defeated by the same man. For example, *16* loses to *11* and *13* loses to *12* in Round 1; after *16* loses to *13* in the second round, *16* is eliminated since he is now known to be inferior to *11*, *12*, and *13*. In Round 3 we do not allow *19* to play with *21*, since they have both been defeated by *18* and we could not automatically eliminate the loser of *19* vs. *21*.

It would be nice to report that Lewis Carroll's tournament turns out to be

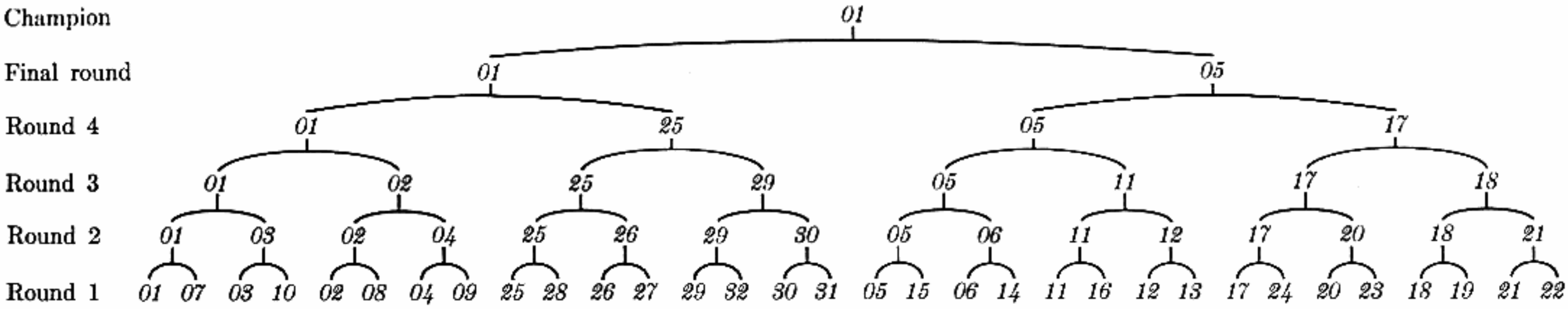


Fig. 39. A knockout tournament with 32 players.

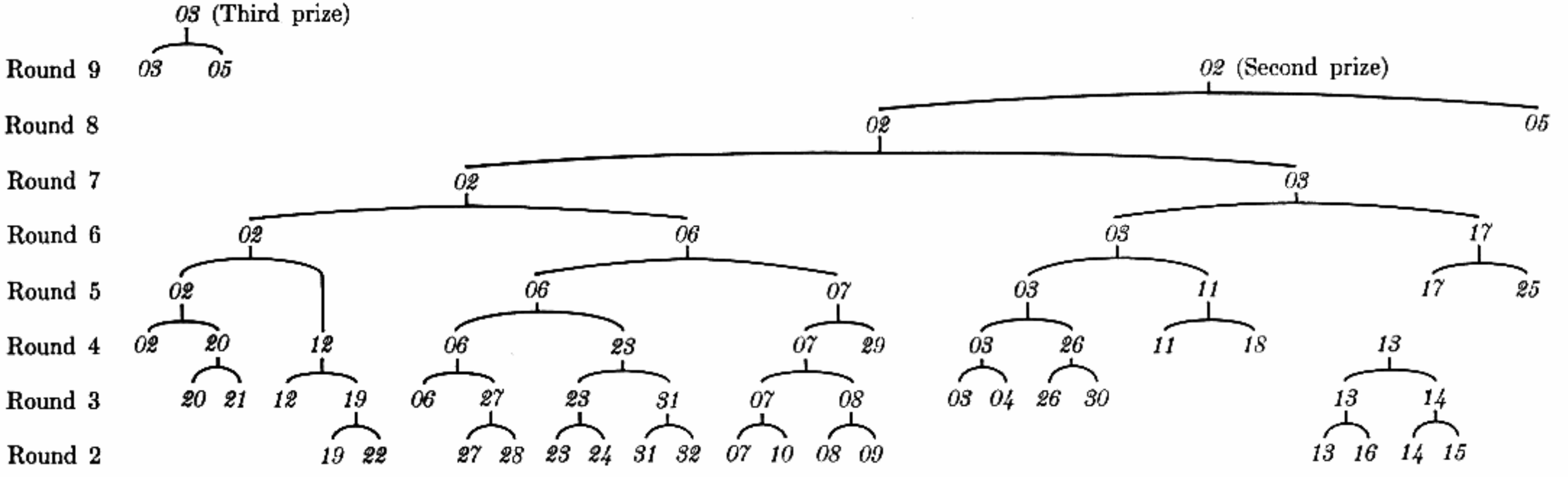


Fig. 40. Lewis Carroll's lawn tennis tournament (played in conjunction with Fig. 39).

optimal, but unfortunately that is not the case. His diary entry for July 23, 1883, says that he composed the essay in about six hours, and he felt "we are now so late in the (tennis) season that it is better it should appear soon than be written well." His procedure makes more comparisons than necessary, and it is not formulated precisely enough to qualify as an algorithm. On the other hand, it has some rather interesting aspects from the standpoint of parallel computation. And it appears to be an excellent plan for a tennis tournament, because he built in some dramatic effects; for example, he specified that the two finalists should sit out round 5, playing an extended match during rounds 6 and 7. But tournament directors presumably thought the proposal was too logical, and so Carroll's system has apparently never been tried. Instead a method of "seeding" is used to keep the best players in different parts of the tree.

In a mathematical seminar during 1929–1930, Hugo Steinhaus posed the problem of finding the *minimum* number of tennis matches required to determine the first and second best players in a tournament, when there are $n \geq 2$ players in all. J. Schreier [*Mathesis Polska* **7** (1932), 154–160] gave a procedure which requires at most $n - 2 + \lceil \log_2 n \rceil$ matches, using essentially the same method as the first two stages in what we have called tree selection sorting (cf. Section 5.2.3, Fig. 23), avoiding redundant comparisons that involve $-\infty$. Schreier also claimed that $n - 2 + \lceil \log_2 n \rceil$ is best possible, but his proof was incorrect, as was another attempted proof by J. Słupecki [*Colloquium Mathematicum* **2** (1951), 286–290]. Thirty-two years went by before a correct, although rather complicated, proof was finally published by S. S. Kislitsyn [*Sibirskiĭ Mat. Zhurnal* **5** (1964), 557–564].

Let $V_t(n)$ denote the minimum number of comparisons needed to determine the tth largest of n elements, for $1 \leq t \leq n$, and let $W_t(n)$ be the minimum number required to determine the largest, second largest, . . . , and tth largest, collectively. By symmetry, we have

$$V_t(n) = V_{n+1-t}(n), \tag{1}$$

and it is obvious that

$$V_1(n) = W_1(n), \tag{2}$$
$$V_t(n) \leq W_t(n), \tag{3}$$
$$W_n(n) = W_{n-1}(n) = S(n). \tag{4}$$

We have observed in Section 5.2.3 that

$$V_1(n) = n - 1. \tag{5}$$

In fact, there is an astonishingly simple proof of this fact, since everyone in a tournament except the champion must lose at least one game! By extending this idea and using an "oracle," we can prove the Schreier-Kislitsyn theorem without much difficulty:

Theorem S. $V_2(n) = W_2(n) = n - 2 + \lceil \log_2 n \rceil$, for $n \geq 2$.

Proof. Assume that n players have participated in a tournament which has determined the second-best player by some given procedure, and let a_j be the number of players who have lost j or more matches. The total number of matches played is then $a_1 + a_2 + a_3 + \cdots$. We cannot determine the second-best player without also determining the champion (cf. exercise 2), so our previous argument shows that $a_1 = n - 1$. To complete the proof, we will show that there is always some sequence of outcomes of the matches which makes $a_2 \geq \lceil \log_2 n \rceil - 1$.

Suppose that at the end of the tournament the champion has played (and beaten) p players; one of these is the second best, and the others must have lost at least one other time, so $a_2 \geq p - 1$. Therefore we can complete the proof by finding an oracle who decides the results of the games in such a way that the champion must play at least $\lceil \log_2 n \rceil$ other people.

Let the oracle declare A to be better than B if A is previously undefeated and B has lost at least once, or if both are undefeated and B has won fewer matches than A at that time. In other circumstances the oracle may make an arbitrary decision consistent with some partial ordering.

Consider the outcome of a complete tournament whose matches have been decided by such an oracle. Let us say that "A supersedes B" if and only if $A = B$ or A supersedes the player who first defeated B. (Only a player's first defeat is relevant in this relation, his subsequent games are ignored. According to the mechanism of the oracle, any player who *first* defeats another must be previously unbeaten.) It follows that a player who won his first p matches supersedes at most 2^p players on the basis of those p contests. (The latter is clear for $p = 0$, and for $p > 0$ the pth match was against someone who was either previously beaten or who supersedes at most 2^{p-1} players.) The champion supersedes everyone, so he must have played at least $\lceil \log_2 n \rceil$ matches. ∎

Thus the problem of finding the second largest is completely resolved in the minimax sense. Exercise 6 shows, in fact, that it is possible to give a simple formula for the minimum number of comparisons needed to find the second largest element of a set when an arbitrary partial ordering of the elements is known.

What if t > 2? In the paper cited above, Kislitsyn went on to consider larger values of t, proving that

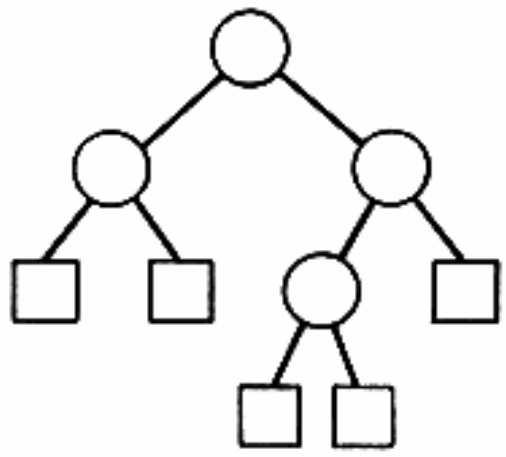
$$W_t(n) \leq n - t + \sum_{n+1-t<j\leq n} \lceil \log_2 j \rceil, \qquad \text{for} \qquad n \geq t. \tag{6}$$

For $t = 1$ and $t = 2$ we have seen that equality actually holds in this formula; for $t = 3$ it can be slightly improved (see exercise 21).

We shall prove Kislitsyn's theorem by showing that the first t stages of *tree selection* require at most $n - t + \sum_{n+1-t<j\leq n} \lceil \log_2 j \rceil$ comparisons (ignoring all of the comparisons which involve $-\infty$). It is interesting to note that the

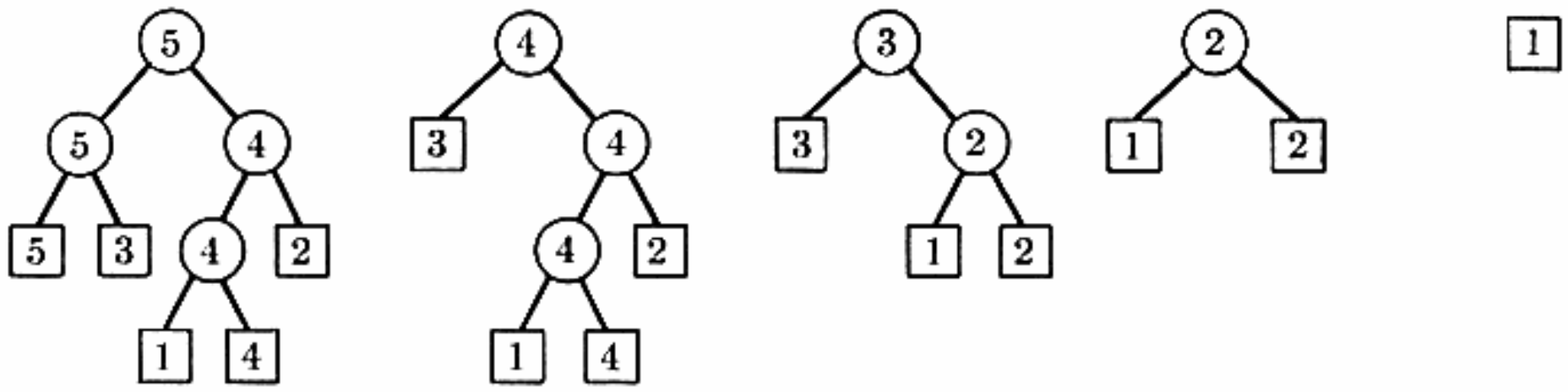
right-hand side of (6) equals $B(n)$ when $t = n - 1$ and $t = n$ (see Eq. 5.3.1–3); hence tree selection and binary insertion yield the same upper bound for the sorting problem, although they are quite different methods.

Let α be an extended binary tree with n external nodes, and let π be a permutation of $\{1, 2, \ldots, n\}$. Place the elements of π into the external nodes, from left to right in symmetric order, and fill in the internal nodes according to the rules of a knockout tournament as in tree selection. When the resulting tree is subjected to repeated selection operations, it defines a sequence $c_{n-1}\ c_{n-2} \ldots c_1$, where c_j is the number of comparisons required to bring element j to the root of the tree when element $j + 1$ has been replaced by $-\infty$. For example, if α is the tree



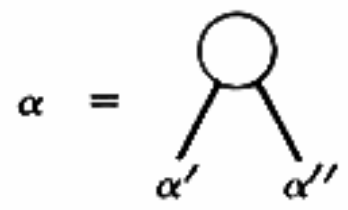
(7)

and if $\pi = 5\ 3\ 1\ 4\ 2$, we obtain the successive trees



If π had been $3\ 1\ 5\ 4\ 2$, the sequence $c_4\ c_3\ c_2\ c_1$ would have been $2\ 1\ 1\ 0$ instead. It is easy to see that c_1 is always zero.

Let $\mu(\alpha, \pi)$ be the multiset $\{c_{n-1}, c_{n-2}, \ldots, c_1\}$ determined by α and π. If



and if elements 1 and 2 do not both appear in α' or both in α'', it is easy to see that

$$\mu(\alpha, \pi) = (\mu(\alpha', \pi') + 1) \uplus (\mu(\alpha'', \pi'') + 1) \uplus \{0\} \tag{8}$$

for appropriate permutations π' and π'', where $(\mu + 1)$ denotes the multiset obtained by adding 1 to each element of μ. On the other hand, if elements 1 and 2 both appear in α', we have

$$\mu(\alpha, \pi) = (\mu(\alpha', \pi') + \epsilon) \uplus (\mu(\alpha'', \pi'') + 1) \uplus \{0\},$$

where $(\mu + \epsilon)$ denotes some multiset obtained by adding 1 to some elements of μ and 0 to the others. A similar formula holds when 1 and 2 both appear in α''. Let us say that multiset μ_1 *dominates* μ_2 if both μ_1 and μ_2 contain the same number of elements, and the kth largest element of μ_1 is greater than or equal to the kth largest element of μ_2 for all k; and let us define $\mu(\alpha)$ to be the "dominant" $\mu(\alpha, \pi)$, taken over all permutations π, in the sense that $\mu(\alpha)$ dominates $\mu(\alpha, \pi)$ for all π and $\mu(\alpha) = \mu(\alpha, \pi)$ for some π. The above formulas show that

$$\mu(\Box) = \emptyset, \qquad \mu\left(\begin{smallmatrix} & \bigcirc & \\ \alpha' & & \alpha'' \end{smallmatrix}\right) = (\mu(\alpha') + 1) \uplus (\mu(\alpha'') + 1) \uplus \{0\}; \tag{9}$$

hence $\mu(\alpha)$ *is the multiset of all distances from the root to the internal nodes of* α.

The reader who has followed the above train of thought will now see that we are ready to prove Kislitsyn's theorem (6); for $W_t(n)$ is less than or equal to $n - 1$ plus the $t - 1$ largest elements of $\mu(\alpha)$, where α is any tree being used in tree selection sorting. We may take α to be the complete binary tree with n external nodes (cf. Section 2.3.4.5), when

$$\begin{aligned} \mu(\alpha) &= \{\lfloor \log_2 1 \rfloor, \lfloor \log_2 2 \rfloor, \ldots, \lfloor \log_2 (n-1) \rfloor\} \\ &= \{\lceil \log_2 2 \rceil - 1, \lceil \log_2 3 \rceil - 1, \ldots, \lceil \log_2 n \rceil - 1\}. \end{aligned} \tag{10}$$

We obtain formula (6) when we consider the $t - 1$ largest elements of this multiset.

Kislitsyn's theorem gives a good upper bound for $W_t(n)$; he remarked that $V_3(5) = 6 < W_3(5) = 7$, but he was unable to find a better bound for $V_t(n)$, in general. A. Hadian and M. Sobel discovered a way to do this using *replacement selection* instead of tree selection (cf. Section 5.4.1). Their formula [Univ. of Minnesota, Dept. of Statistics, report 121 (May, 1969)],

$$V_t(n) \le n - t + (t-1)\lceil \log_2 (n + 2 - t) \rceil, \qquad n \ge t, \tag{11}$$

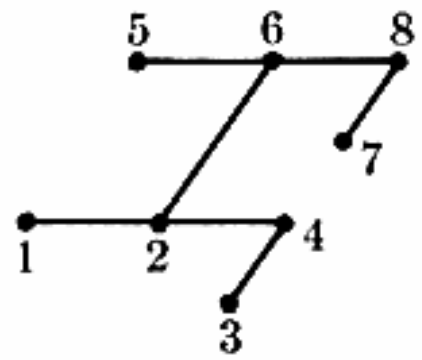
differs from (6) in that each element in the sum of (6) is replaced by the smallest element.

Hadian and Sobel's theorem (11) can be proved by using the following construction: First set up a binary tree for a knockout tournament on $n - t + 2$ items. (This takes $n - t + 1$ comparisons.) The largest item is greater than $n - t + 1$ others, so it can't be tth largest. Replace it, where it appears at an external node of the tree, by one of the $t - 2$ elements held in reserve, and find the largest element of the resulting $n - t + 2$; this requires at most $\lceil \log_2 (n + 2 - t) \rceil$ comparisons, because we need to recompute only one path in the tree. Repeat this operation $t - 2$ times in all, for each element held in reserve. Finally, replace the currently largest element by $-\infty$, and determine the largest of the remaining $n + 1 - t$; this requires at most $\lceil \log_2 (n + 2 - t) \rceil - 1$ comparisons, and it brings the tth largest element of the original set to the root of the tree. Summing the comparisons yields (11).

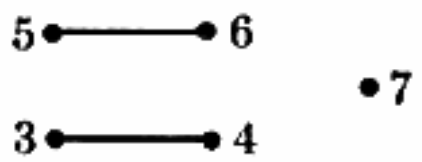
In relation (11) we should of course replace t by $n + 1 - t$ on the right-hand side whenever $n + 1 - t$ gives a better value (as when $n = 6$, $t = 3$).

Curiously, the formula gives a smaller bound for $V_7(13)$ than it does for $V_6(13)$. The upper bound in (11) is exact for $n \leq 6$, but as n and t get larger it is possible to obtain much better estimates of $V_t(n)$.

For example, the following elegant method (due to David G. Doren) can be used to show that $V_4(8) \leq 12$. Let the elements be $X_1, \ldots, X_8$; first compare $X_1{:}X_2$ and $X_3{:}X_4$ and the two winners, and do the same to $X_5{:}X_6$ and $X_7{:}X_8$ and their winners. Relabel elements so that $X_1 < X_2 < X_4 > X_3$, $X_5 < X_6 < X_8 > X_7$, then compare $X_2{:}X_6$; by symmetry assume that $X_2 < X_6$, so that we have the configuration



(Now X_1 and X_8 are out of contention and we must find the third largest of $\{X_2, \ldots, X_7\}$.) Compare $X_2{:}X_7$, and discard the smaller; in the worst case we have $X_2 < X_7$ and we must find the third largest of



This can be done in $V_3(5) - 2 = 4$ more steps, since the $V_3(5) = 6$ procedure of (11) begins by comparing two disjoint pairs of elements.

Other tricks of this kind can be used to produce the results shown in Table 1; no general method is evident as yet, and the values listed in the table for $n \geq 8$ may be subject to further improvement. Note that $V_3(10) \leq 14$, so (11) does not always give equality when $t = 3$. The fact that $V_4(7) = 10$ shows that (11) is already off by 2 when $n = 7$.

Table 1

BEST UPPER BOUNDS KNOWN FOR $V_t(n)$

n	$V_1(n)$	$V_2(n)$	$V_3(n)$	$V_4(n)$	$V_5(n)$	$V_6(n)$	$V_7(n)$	$V_8(n)$	$V_9(n)$	$V_{10}(n)$
1	0									
2	1	1								
3	2	3	2							
4	3	4	4	3						
5	4	6	6	6	4					
6	5	7	8	8	7	5				
7	6	8	10	10*	10	8	6			
8	7	9	11	12	12	11	9	7		
9	8	11	12	14	15*	14	12	11	8	
10	9	12	14*	15	17	17	15	14*	12	9

*Exercises 10–12 give constructions which improve on Eq. (11) in these cases.

A linear method. When n is odd and $t = \lceil n/2 \rceil$, the tth largest (and tth smallest) element is called the median. According to (11), we can find the median of n elements in $\approx \frac{1}{2} n \log_2 n$ comparisons; but this is only about twice as fast as sorting, even though we are asking for much less information. For several years, concerted efforts were made by a number of people to find an improvement over (11) when t and n are large; finally in 1971, Manuel Blum discovered a method which needed only $O(n \log \log n)$ steps. Blum's approach to the problem suggested a new class of techniques which led to the following construction due to R. Rivest and R. Tarjan:

Theorem L. $V_t(n) \leq 15n - 163$ *for* $1 \leq t \leq n$, *when* $n > 32$.

Proof. The theorem is trivial when n is small, since $V_t(n) \leq S(n) \leq 10n \leq 15n - 163$ for $32 < n \leq 2^{10}$. By adding at most 13 dummy "$-\infty$" elements, we may assume that $n = 7(2q + 1)$ for some integer $q \geq 73$. The following method may now be used to select the tth largest:

Step 1. Divide the elements into $2q + 1$ groups of seven elements each, and sort each of the groups. This takes at most $13(2q + 1)$ comparisons.

Step 2. Find the median of the $2q + 1$ median elements obtained in Step 1, and call it x. By induction on q, this takes at most $V_{q+1}(2q + 1) \leq 30q - 148$ comparisons.

Step 3. The $n - 1$ elements other than x have now been partitioned into three sets (see Fig. 41):

$4q + 3$ elements known to be greater than x (Region B);
$4q + 3$ elements known to be less than x (Region C);
$6q$ elements whose relation to x is unknown (Regions A, D).

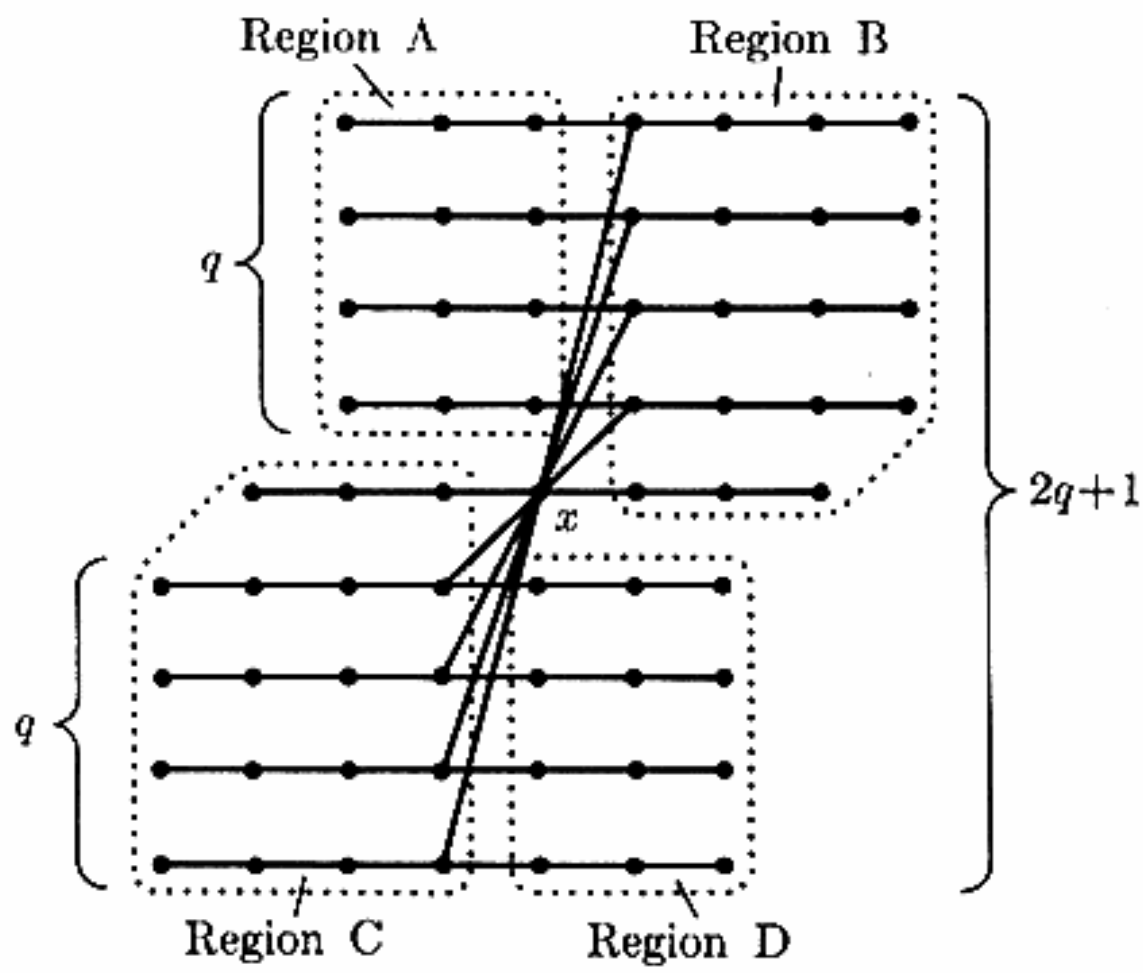


Fig. 41. The selection algorithm of Rivest and Tarjan ($q = 4$).

By making $4q$ additional comparisons, we can tell exactly which of the elements in regions A and D are less than x. (We first test x against the middle element of each triple.)

Step 4. We have now found r elements greater than x and $n - 1 - r$ elements less than x, for some r. If $t = r + 1$, x is the answer; if $t < r + 1$, we need to find the tth largest of the r large elements; and if $t > r + 1$, we need to find the $(t - 1 - r)$th largest of the $n - 1 - r$ small elements. The point is that r and $n - 1 - r$ are both less than or equal to $10q + 3$ (the size of regions A and D, plus either B or C). By induction on q this step therefore requires at most $15(10q + 3) - 163$ comparisons.

The total number of comparisons comes to at most

$$13(2q + 1) + 30q - 148 + 4q + 15(10q + 3) - 163 = 15(14q - 6) - 163.$$

Since we started with at least $14q - 6$ elements, the proof is complete. ∎

The method used in this proof is a little crude because it throws away good information in Step 4. Careful refinements by V. Pratt, R. Rivest, and R. Tarjan have shown that the constant 15 can be reduced to 5.43.

The average number. Instead of minimizing the *maximum* number of comparisons, we can ask instead for an algorithm which minimizes the *average* number of comparisons, assuming random order. As usual, this problem is considerably harder, and it is still unsolved even in the case $t = 2$. Claude Picard mentioned the problem in his book *Théorie des Questionnaires* (1965), and an extensive exploration was undertaken by Milton Sobel [*Univ. of Minnesota, Dept. of Statistics*, report 113 (November, 1968)].

Sobel constructed the procedure of Fig. 42, which finds the second largest of six elements using only $6\frac{1}{2}$ comparisons on the average. In the worst case, 8 comparisons are required, and this is worse than $V_2(6) = 7$; but all known procedures for this problem which require at most 7 comparisons use at least $6\frac{2}{3}$ comparisons on the average. Thus it is likely that no procedure which finds the second largest of six elements will be optimal in both the minimax and the minimean senses simultaneously.

Let $\overline{V}_t(n)$ denote the minimum average number of comparisons needed to find the tth largest of n elements. The following table shows the best upper bounds known for $\overline{V}_2(n)$, as computed by Sobel:

$$\begin{array}{rccccccccc} n = & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \overline{V}_2(n) \le & 1 & 2\frac{2}{3} & 4 & 5\frac{4}{15} & 6\frac{1}{2} & 7\frac{17}{21} & 9 & 10\frac{1}{15} & 11\frac{4}{135} \end{array} \tag{12}$$

Sobel has conjectured that

$$\overline{V}_2(n) \ge n - 2 + \lfloor 2 \log_2 n \rfloor / 2. \tag{13}$$

R. W. Floyd discovered in 1970 that the median of n elements can be found

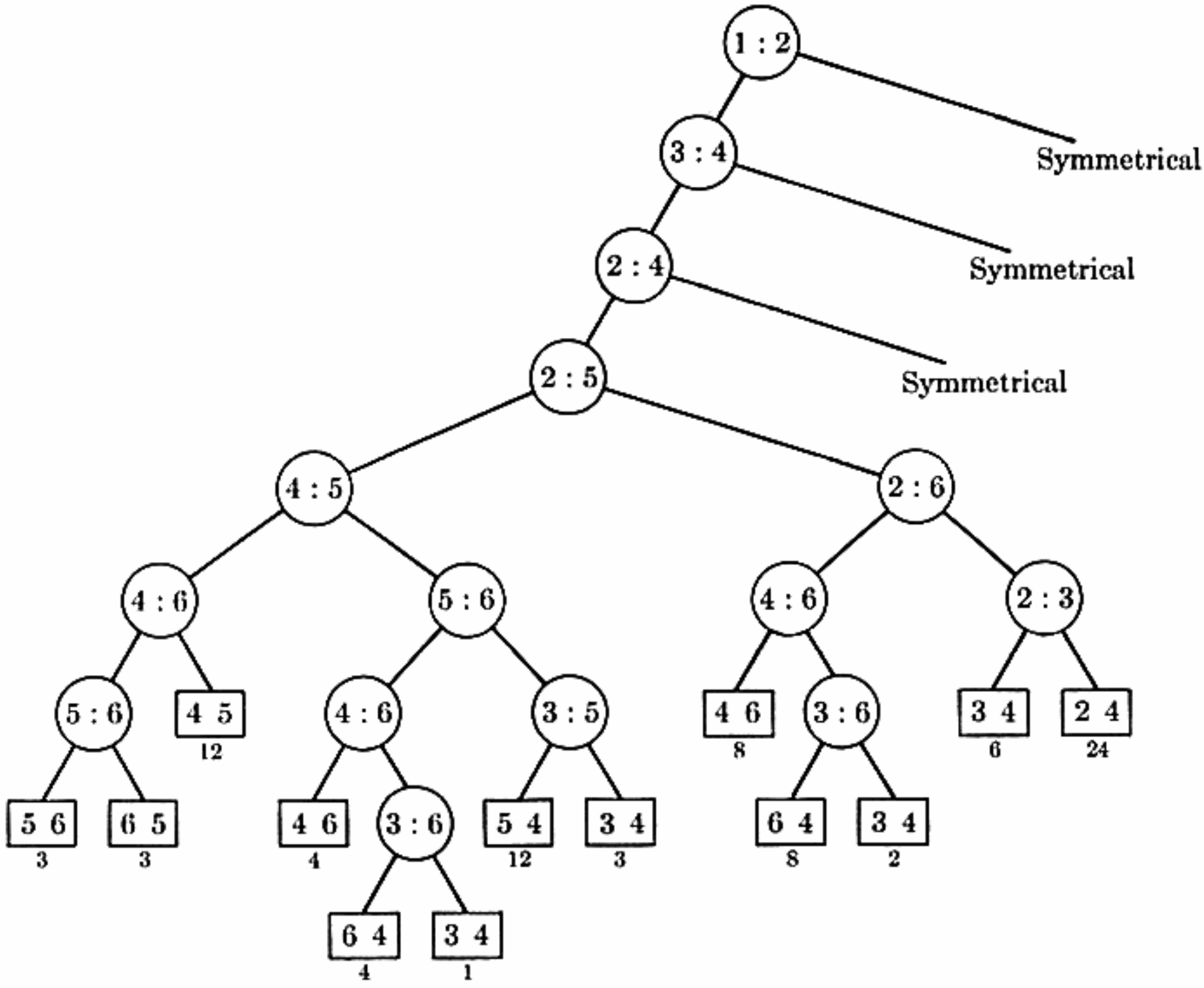


Fig. 42. A procedure which selects the second largest of $\{X_1, X_2, X_3, X_4, X_5, X_6\}$, using $6\frac{1}{2}$ comparisons on the average. Each "symmetrical" branch is identical to its brother, with names permuted in some appropriate manner. External nodes contain "j k" when X_j is known to be the second largest and X_k the largest; the number of permutations leading to this node appears immediately below it.

with only $\frac{3}{2}n + O(n^{2/3} \log n)$ comparisons, on the average. (See exercise 13.) He proved in fact that

$$\overline{V}_t(n) \leq n + t + f(n), \qquad \text{where} \qquad \lim_{n\to\infty} f(n)/n = 0. \tag{14}$$

This result is conjectured to be the best possible asymptotic formula, but no satisfactory lower bound has yet been obtained.

EXERCISES

1. [*15*] In Lewis Carroll's tournament (Figs. 39 and 40), why does player *13* get eliminated even though he has won his match in Round 3?

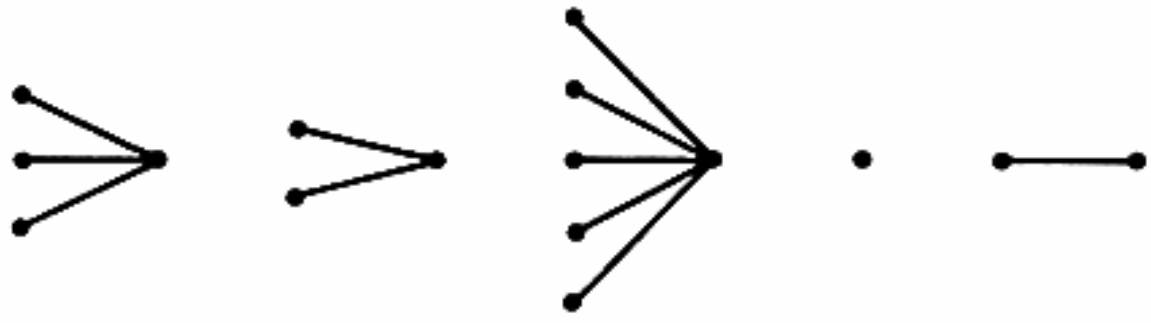
▶ **2.** [*M25*] Prove that after we have found the tth largest of n elements by a sequence of comparisons, we also know which $t - 1$ elements are greater than it, and which $n - t$ elements are less than it.

3. [*M21*] Prove that $V_t(n) \geq V_t(n-1) + 1$, for $1 \leq t \leq n$.

4. [*M20*] Prove that $W_t(n) \geq \lceil \log_2 n^{\underline{t}} \rceil$, where $n^{\underline{t}} = n(n-1)\ldots(n+1-t)$.

5. [*10*] Prove that $W_3(n) \leq V_3(n) + 1$.

▸ **6.** [*M26*] (R. W. Floyd.) Given n distinct elements $\{X_1, \ldots, X_n\}$ and relations $X_i < X_j$ for certain pairs (i, j), we wish to find the second largest element. If we know that $X_i < X_j$ and $X_i < X_k$ for $j \neq k$, X_i cannot possibly be the second largest, so it can be eliminated. The resulting relations now have a form such as



namely, m groups of elements which can be represented by a vector $(l_1, l_2, \ldots, l_m)$; the jth group contains $l_j + 1$ elements, one of which is known to be greater than the others. For example, the above configuration can be described by the vector (3, 2, 4, 0, 1); when no relations are known we have a vector of n zeros.

Let $f(l_1, l_2, \ldots, l_m)$ be the minimum number of comparisons needed to find the second largest element of such a partially ordered set. Prove that

$$f(l_1, l_2, \ldots, l_m) = m - 2 + \lceil \log_2 (2^{l_1} + 2^{l_2} + \cdots + 2^{l_m}) \rceil.$$

[*Hint:* Show that the best strategy is always to compare the largest elements of the two smallest groups, until reducing m to unity; use induction on $l_1 + l_2 + \cdots + l_m + 2m$.]

7. [*M20*] Prove (8).

8. [*M21*] Kislitsyn's formula (6) is based on tree selection sorting using the complete binary tree with n external nodes. Would a tree selection method based on some *other* tree give a better bound, for any t and n?

▸ **9.** [*20*] Draw a comparison tree for finding the median of five elements in at most six steps, using the replacement-selection method of Hadian and Sobel [cf. (11)].

10. [*35*] Show that the median of seven elements can be found in at most 10 steps.

11. [*28*] (Hadian and Sobel.) Show that the median of nine elements can be found in at most 15 steps, by extending the method of Doren stated in the text.

12. [*21*] (Hadian and Sobel.) Prove that $V_3(n) \leq V_3(n-1) + 2$. [*Hint:* Start by discarding the smallest of $\{X_1, X_2, X_3, X_4\}$.]

▸ **13.** [*HM28*] (R. W. Floyd.) Show that if we start by finding the median element of $\{X_1, \ldots, X_{n^{2/3}}\}$, using a recursively defined method, we can go on to find the median of $\{X_1, \ldots, X_n\}$ with an average of $\frac{3}{2}n + O(n^{2/3} \log n)$ comparisons.

▸ **14.** [*20*] (M. Sobel.) Show that it is possible to find the largest two elements of $\{X_1, X_2, X_3, X_4, X_5\}$, without necessarily knowing their relative order, using at most five comparisons.

15. [*22*] (I. Pohl.) Suppose that we are interested in minimizing space instead of time. What is the minimum number of data words needed in memory in order to compute the tth largest of n elements, if each element fills one word and if the elements are input one at a time into a single register?

▸**16.** [*25*] (I. Pohl.) Show that we can find both the maximum and the minimum of a set of n elements, using at most $\lceil \frac{3}{2}n \rceil - 2$ comparisons; and the latter number cannot be lowered. [*Hint:* Any stage in such an algorithm can be represented as a quadruple (a, b, c, d), where a elements have never been compared, b have won but never lost, c have lost but never won, d have both won and lost. Construct a suitable oracle.]

17. [*20*] (R. W. Floyd.) Show that it is possible to select, in order, both the k largest and the l smallest elements of a set of n elements, using at most $\lceil \frac{3}{2}n \rceil - k - l + \sum_{n+1-k<j\le n} \lceil \log_2 j \rceil + \sum_{n+1-l<j\le n} \lceil \log_2 j \rceil$ comparisons.

18. [*M20*] If groups of size 5, not 7, had been used in the proof of Theorem L, what theorem would have been obtained?

19. [*M44*] Find the exact value of $\overline{V}_2(6)$. Can this be achieved by a procedure which never makes more than seven comparisons?

20. [*M47*] Prove (or disprove) Sobel's conjecture (13).

21. [*25*] (S. Lin.) Prove that $W_3(2^k + 2) \le 2^k + 2k$, when $k \ge 3$.

22. [*M47*] Determine the values of $V_3(n)$ and $W_3(n)$ for all n.

23. [*M49*] What is the asymptotic value of $V_{\lceil n/2 \rceil}(n)$, as $n \to \infty$?

24. [*M48*] What is the asymptotic value of $\overline{V}_{\lceil n/2 \rceil}(n)$, as $n \to \infty$?

5.3.4. Networks for Sorting

In this section we shall study a "constrained" type of sorting which is especially interesting because of its applications and its rich underlying theory. The new constraint is to insist on a *homogeneous* sequence of comparisons, in the sense that whenever we compare K_i versus K_j the subsequent comparisons for the case $K_i < K_j$ are exactly the same as for the case $K_i > K_j$, but with i and j interchanged. Figure 43(a) shows a comparison tree in which this homogeneity condition is satisfied. (Note that the same number of comparisons is made on each level, so there are 2^m outcomes after m comparisons have been made; since $n!$ is not a power of 2, it follows that some of the comparisons will be redundant, in the sense that one of their subtrees can never arise in practice. In other words, some branches of the tree must make more comparisons than necessary in order to ensure that all of the corresponding branches of the tree will sort properly.)

Since each path from top to bottom of such a tree determines the entire tree, such a sorting scheme is most easily represented as a *network* as in Fig. 42(b). The boxes in such a network represent "comparator modules" which have two inputs (represented as lines coming into the module from above) and two outputs (represented as lines leading downward); the left-hand output is the smaller of the two inputs, and the right-hand output is the larger. At the bottom of the network, K_1' is the smallest of $\{K_1, K_2, K_3, K_4\}$, K_2' the second smallest, etc. It is not difficult to prove that any sorting network corresponds to a homogeneous comparison tree in the above sense, and any homogeneous tree corresponds to a network of comparator modules.

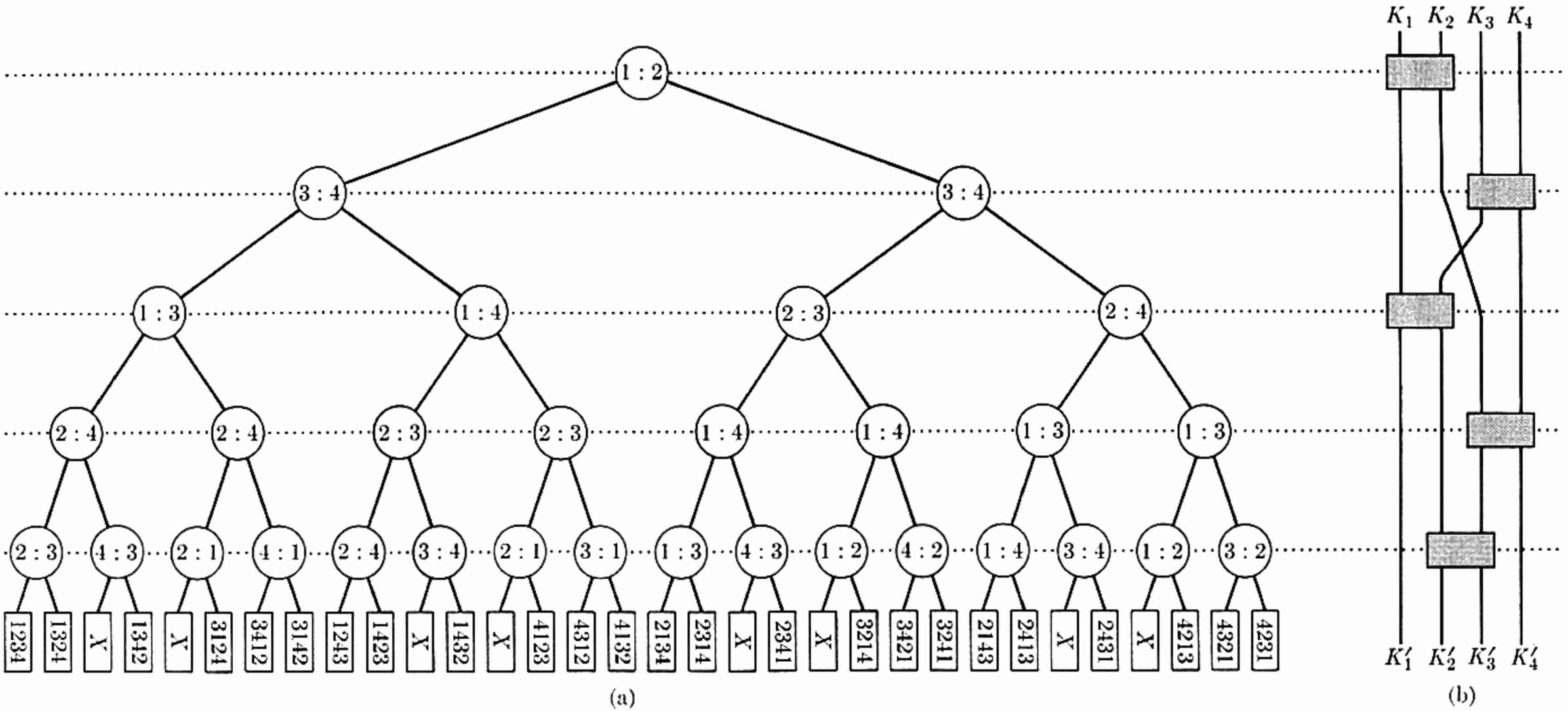


Fig. 43. (a) A homogeneous comparison tree. (b) The corresponding network.

Incidentally, we may note that comparator modules are fairly easy to manufacture, from an engineering point of view. For example, assume that the lines contain binary numbers, where one bit enters each module per unit time, most significant bit first. Each comparator module has three states, and behaves as follows:

Time t		Time $(t+1)$	
State	Inputs	State	Outputs
0	0 0	0	0 0
0	0 1	1	0 1
0	1 0	2	0 1
0	1 1	0	1 1
1	x y	1	x y
2	x y	2	y x

Initially all modules are in state 0 and are outputting 0 0. A module enters either state 1 or state 2 as soon as its inputs differ. Numbers which begin to be transmitted at the top of Fig. 43(b) at time t will begin to be output at the bottom, in sorted order, at time $t+3$, if a suitable delay element is attached to the K_1' and K_4' lines.

In order to develop the theory of sorting networks it is convenient to represent them in a slightly different way, illustrated in Fig. 44. Here numbers enter at the left, and comparator modules are represented by vertical connections between two lines; each comparator causes an interchange of its inputs, if necessary, so that the larger number appears on the lower line after passing the comparator. At the right of the diagram all the numbers are in order from top to bottom.

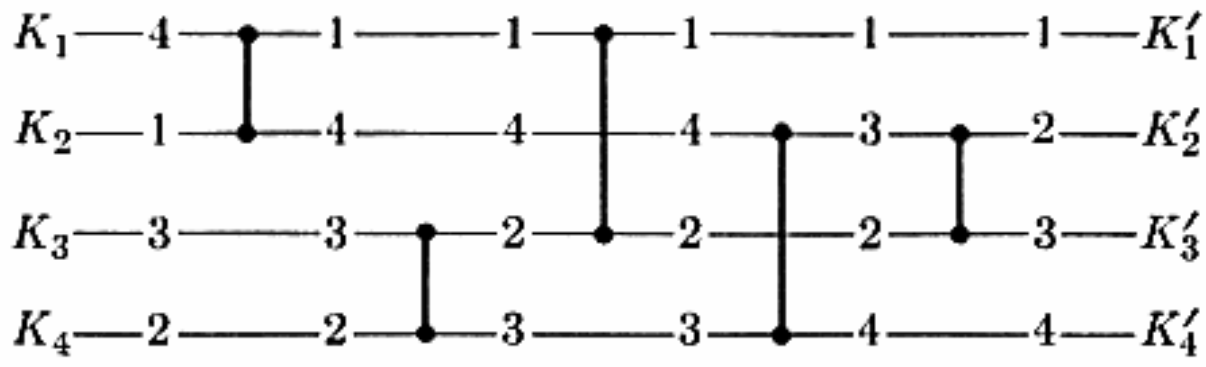


Fig. 44. Another way to represent the network of Fig. 43, as it sorts the sequence $\langle 4, 1, 3, 2\rangle$.

Our previous studies of optimal sorting have concentrated on minimizing the number of comparisons with little or no regard for any underlying data movement or for the complexity of the decision structure that may be necessary. In this respect sorting networks have some advantages, since the data can be maintained in n locations and the decision structure is "straight line"; there is no need to remember the results of previous comparisons—the plan is immutably fixed in advance. Another important advantage of sorting networks is that it may be possible to overlap several of the operations, performing them

simultaneously (on a suitable machine). For example, the five steps in Figs. 43 and 44 can be collapsed into three when simultaneous nonoverlapping comparisons are allowed, since the first two and the second two can be combined; we shall exploit this property of sorting networks later in this section. Thus sorting networks can be very useful, although it is not at all obvious that efficient n-element sorting networks can be constructed for large n; we may find that many additional comparisons are needed in order to keep the decision structure homogeneous.

There are two simple ways to construct a sorting network for $n + 1$ elements when an n-element network is given, using either the principle of *insertion* or the principle of *selection*. Figure 45(a) shows how the $(n + 1)$st element can be inserted into its proper place after the first n elements have been sorted; and part (b) of the figure shows how the largest element can be selected before we proceed to sort the remaining ones. Repeated application of Fig. 45(a) gives the network analog of straight insertion sorting (Algorithm 5.2.1S), and repeated application of Fig. 45(b) yields the network analog of the bubble sort (Algorithm 5.2.2B). Figure 46 shows the corresponding six-element networks. It is interesting to note that when we collapse either network together to allow simultaneous operations, both methods reduce to the same "triangular" $(2n - 3)$-stage procedure (Fig. 47).

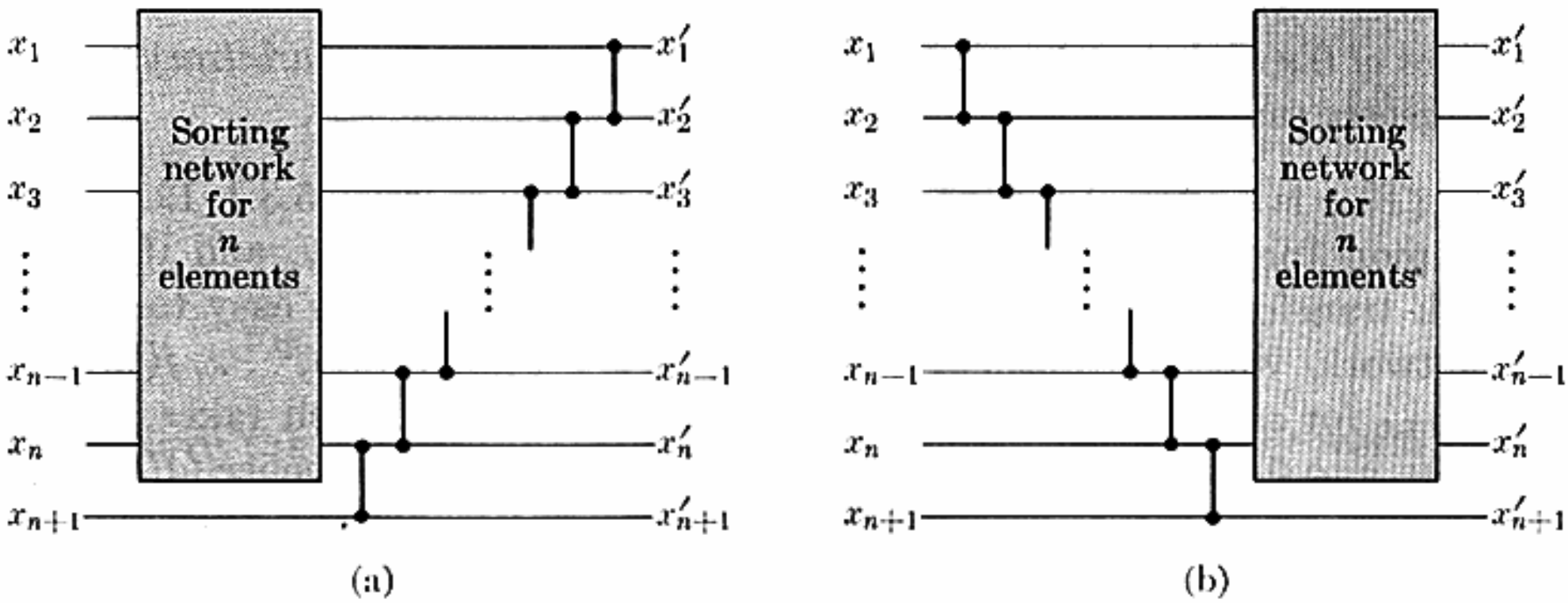


Fig. 45. Making $(n + 1)$-sorters from n-sorters: (a) insertion, (b) selection.

Fig. 46. Network analogs of elementary internal sorting schemes, obtained by applying Fig. 45 repeatedly: (a) straight insertion, (b) bubble sort.

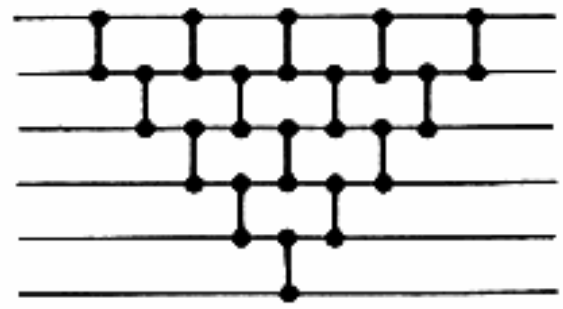
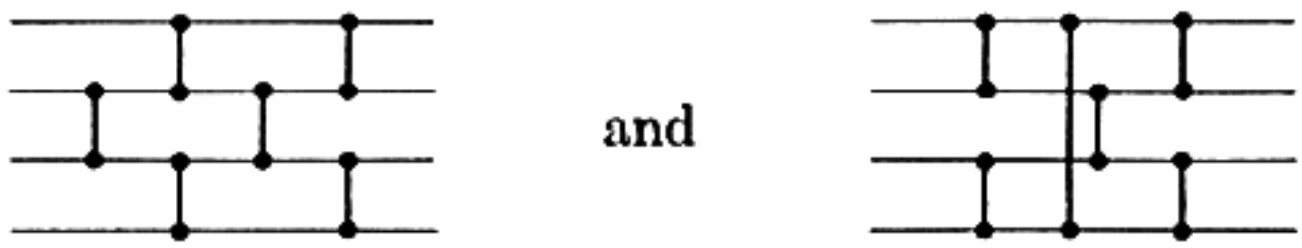


Fig. 47. With parallelism, straight insertion = bubble sort!

It is easy to prove that the network of Figs. 43 and 44 will sort any set of four numbers into order, since the first four comparators route the smallest and the largest elements to the correct places, and the last comparator puts the remaining two elements in order. But it is not always so easy to tell whether or not a given network will sort all possible input sequences; for example, both



are valid 4-element sorting networks, but the proofs of their validity are not trivial. It would be sufficient to test each n-element network on all $n!$ permutations of n distinct numbers, but in fact we can get by with far fewer tests:

Theorem Z (*Zero-one principle*). *If a network with n input lines sorts all 2^n sequences of 0's and 1's into nondecreasing order, it will sort any arbitrary sequence of n numbers into nondecreasing order.*

Proof. (This is a special case of Bouricius's theorem, exercise 5.3.1–12.) If $f(x)$ is any monotonic function, with $f(x) \le f(y)$ whenever $x \le y$, and if a given network transforms $\langle x_1, \ldots, x_n \rangle$ into $\langle y_1, \ldots, y_n \rangle$, then it is easy to see that the network will transform $\langle f(x_1), \ldots, f(x_n) \rangle$ into $\langle f(y_1), \ldots, f(y_n) \rangle$. If $y_i > y_{i+1}$ for some i, consider the monotonic function f which takes all numbers $< y_i$ into 0 and all numbers $\ge y_i$ into 1; this defines a sequence $\langle f(x_1), \ldots, f(x_n) \rangle$ of 0's and 1's which is not sorted by the network. Hence if all $0 - 1$ sequences are sorted, we have $y_i \le y_{i+1}$ for $1 \le i < n$. ∎

The zero-one principle is quite helpful in the construction of sorting networks. As a nontrivial example, we can derive a generalized version of Batcher's "merge exchange" sort (Algorithm 5.2.2M). The idea is to sort $m + n$ elements by sorting the first m and the last n independently, then applying an (m, n)-*merging network* to the result. An (m, n)-merging network can be constructed inductively as follows:

a) If $m = 0$ or $n = 0$, the network is empty. If $m = n = 1$, the network is a single comparator module.
b) If $mn > 1$, let the sequences to be merged be $\langle x_1, \ldots, x_m \rangle$ and $\langle y_1, \ldots, y_n \rangle$. Merge the "odd sequences" $\langle x_1, x_3, \ldots, x_{2\lceil m/2 \rceil - 1} \rangle$ and $\langle y_1, y_3, \ldots, y_{2\lceil n/2 \rceil - 1} \rangle$, obtaining the sorted result $\langle v_1, v_2, \ldots, v_{\lceil m/2 \rceil + \lceil n/2 \rceil} \rangle$; and merge

the "even sequences" $\langle x_2, x_4, \ldots, x_{\lfloor m/2 \rfloor}\rangle$ and $\langle y_2, y_4, \ldots, y_{\lfloor n/2 \rfloor}\rangle$, obtaining the sorted result $\langle w_1, w_2, \ldots, w_{\lfloor m/2 \rfloor+\lfloor n/2 \rfloor}\rangle$. Finally, apply the comparison-interchange operations

$$w_1{:}v_2, \quad w_2{:}v_3, \quad w_3{:}v_4, \quad \ldots, \quad w_{\lfloor m/2 \rfloor+\lfloor n/2 \rfloor}{:}v^* \tag{1}$$

to the sequence

$$\langle v_1, w_1, v_2, w_2, v_3, w_3, \ldots, v_{\lfloor m/2 \rfloor+\lfloor n/2 \rfloor}, w_{\lfloor m/2 \rfloor+\lfloor n/2 \rfloor}, v^*, v^{**}\rangle; \tag{2}$$

the result will be sorted. (!) (Here $v^* = v_{\lfloor m/2 \rfloor+\lfloor n/2 \rfloor+1}$ does not exist if both m and n are even, and $v^{**} = v_{\lfloor m/2 \rfloor+\lfloor n/2 \rfloor+2}$ does not exist unless both m and n are odd; the total number of comparator modules indicated in (1) is $\lfloor (m + n - 1)/2. \rfloor$)

Batcher's (m, n)-merging network is called the *odd-even merge*. A (4, 7)-merge constructed according to these principles is illustrated in Fig. 48.

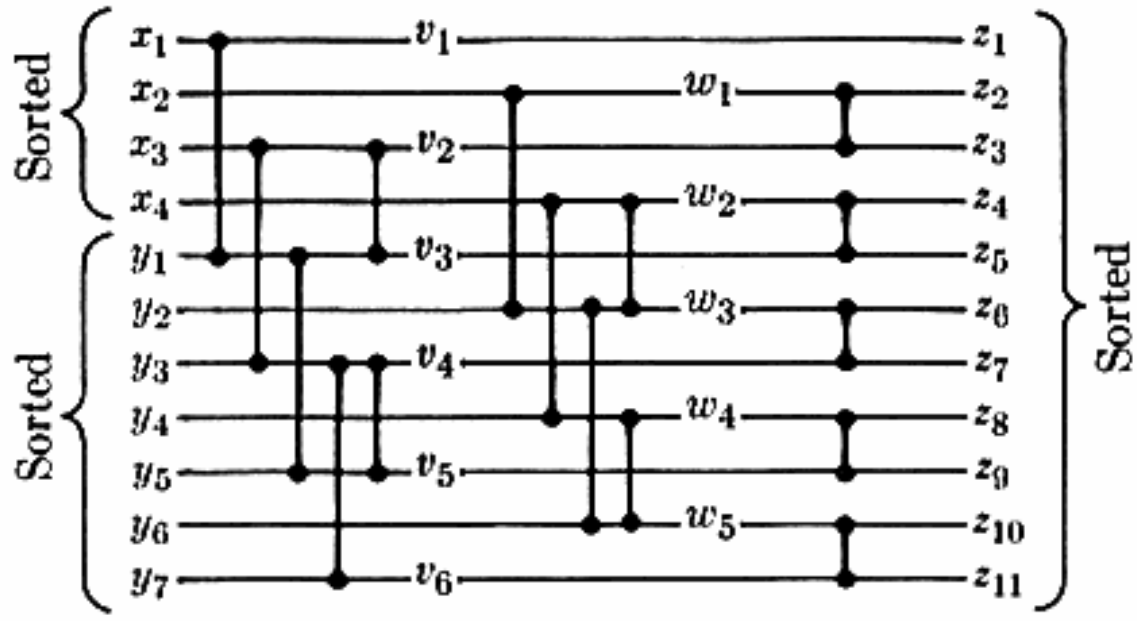


Fig. 48. The odd-even merge, when $m = 4$ and $n = 7$.

To prove that this rather strange merging procedure actually works, when $mn > 1$, we use the zero-one principle, testing it on all sequences of 0's and 1's. After the initial m-sort and n-sort, the sequence $\langle x_1, \ldots, x_m\rangle$ will consist of k 0's followed by $m - k$ 1's, and the sequence $\langle y_1, \ldots, y_n\rangle$ will be l 0's followed by $n - l$ 1's, for some k and l. Hence the sequence $\langle v_1, v_2, \ldots\rangle$ will consist of exactly $\lceil k/2 \rceil + \lceil l/2 \rceil$ 0's, followed by 1's; and $\langle w_1, w_2, \ldots\rangle$ will consist of $\lfloor k/2 \rfloor + \lfloor l/2 \rfloor$ 0's, followed by 1's. Now here's the point:

$$(\lceil k/2 \rceil + \lceil l/2 \rceil) - (\lfloor k/2 \rfloor + \lfloor l/2 \rfloor) = 0, \quad 1, \quad \text{or} \quad 2. \tag{3}$$

If this difference is 0 or 1, the sequence (2) is already in order, and if the difference is 2 one of the comparison-interchanges in (1) will fix everything up. This completes the proof. (Note that the zero-one principle reduces the merging problem from a consideration of $\binom{m+n}{m}$ cases to only $(m + 1)(n + 1)$, represented by the two parameters k and l.)

Let $C(m, n)$ be the number of comparator modules used in the odd-even merge for m and n, not counting the initial m-sort and n-sort; we have

$$C(m, n) = \begin{cases} mn, & \text{if} \quad mn \le 1; \\ C(\lceil m/2 \rceil, \lceil n/2 \rceil) + C(\lfloor m/2 \rfloor, \lfloor n/2 \rfloor) & \\ \quad + \lfloor (m + n - 1)/2 \rfloor, & \text{if} \quad mn > 1. \end{cases} \tag{4}$$

This is not an especially simple function of m and n, in general, but by noting that $C(1, n) = n$ and that

$$\begin{aligned} &C(m + 1, n + 1) - C(m, n) \\ &\qquad = 1 + C(\lfloor m/2 \rfloor + 1, \lfloor n/2 \rfloor + 1) - C(\lfloor m/2 \rfloor, \lfloor n/2 \rfloor), \qquad \text{if} \quad mn \ge 1, \end{aligned}$$

we can derive the relation

$$C(m + 1, n + 1) - C(m, n) = t + 2 + \lfloor n/2^{t+1} \rfloor,$$
$$\text{if} \quad n \ge m \ge 1 \quad \text{and} \quad t = \lfloor \log_2 m \rfloor. \tag{5}$$

Consequently,

$$C(m, m + r) = B(m) + m + R_m(r), \qquad \text{for} \quad m \ge 0, r \ge 0, \tag{6}$$

where $B(m)$ is the "binary insertion" function $\sum_{1 \le k \le m} \lceil \log_2 k \rceil$ of Eq. 5.3.1–3, and where $R_m(r)$ denotes the sum of the first m terms of the series

$$\left\lfloor \frac{r+0}{1} \right\rfloor + \left\lfloor \frac{r+1}{2} \right\rfloor + \left\lfloor \frac{r+2}{4} \right\rfloor + \left\lfloor \frac{r+3}{4} \right\rfloor + \left\lfloor \frac{r+4}{8} \right\rfloor + \cdots + \left\lfloor \frac{r+j}{2^{\lfloor \log_2 j \rfloor + 1}} \right\rfloor + \cdots. \tag{7}$$

In particular, when $r = 0$, we have the important special case

$$C(m, m) = B(m) + m. \tag{8}$$

Furthermore if $t = \lceil \log_2 m \rceil$,

$$\begin{aligned} R_m(r + 2^t) &= R_m(r) + 1 \cdot 2^{t-1} + 2 \cdot 2^{t-2} + \cdots + 2^{t-1} \cdot 2^0 + m \\ &= R_m(r) + m + t \cdot 2^{t-1}. \end{aligned}$$

Hence $C(m, n + 2^t) - C(m, n)$ has a simple form, and

$$C(m, n) = \left(\frac{t}{2} + \frac{m}{2^t} \right) n + O(1), \qquad \text{for } m \text{ fixed}, \quad n \to \infty, \quad t = \lceil \log_2 m \rceil; \tag{9}$$

the $O(1)$ term is an eventually periodic function of n, with period length 2^t. The asymptotic value of $C(n, n)$ as $n \to \infty$ is $n \log_2 n$, by Eq. (8) and exercise 5.3.1–15.

Minimum-comparison networks. Let $\hat{S}(n)$ be the minimum number of comparators needed in a sorting network for n elements; clearly $\hat{S}(n) \ge S(n)$,

where $S(n)$ is the minimum number of comparisons needed in an unrestricted sorting procedure (cf. Section 5.3.1). We have $\hat{S}(4) = 5 = S(4)$, so the new constraint causes no loss of efficiency when $n = 4$; but already when $n = 5$ it turns out that $\hat{S}(5) = 9$ while $S(5) = 7$. The problem of determining $\hat{S}(n)$ seems to be even harder than the problem of determining $S(n)$; even the asymptotic behavior of $\hat{S}(n)$ is still unknown.

It is interesting to trace the history of this problem, since each step was forged with some difficulty. Sorting networks were first explored by P. N. Armstrong, R. J. Nelson, and D. J. O'Connor, about 1954 [see *U.S. Patent 3029413*]; they showed that $\hat{S}(n+1) \leq \hat{S}(n) + n$. In the words of their patent attorney, "By the use of skill, it is possible to design economical n-line sorting switches using a reduced number of two-line sorting switches"; they gave special constructions for $4 \leq n \leq 8$, using 5, 9, 12, 18, and 19 comparators, respectively. In Nelson's subsequent work on the problem together with R. C. Bose, a general procedure for constructing sorting networks was devised prior to 1960, showing that $\hat{S}(2^n) \leq 3^n - 2^n$ for all n; hence $\hat{S}(n) = O(n^{\log_2 3}) = O(n^{1.585})$. Bose and Nelson published their interesting method in *JACM* **9** (1962), 282–296, where they conjectured that it was best possible; T. N. Hibbard [*JACM* **10** (1963), 142–150] found a similar but slightly simpler method which used the same number of comparisons, thereby reinforcing the conjecture.

In 1964, R. W. Floyd and D. E. Knuth found a new way to approach the problem, leading to an asymptotic bound of the form $\hat{S}(n) = O(n^{1+c/\sqrt{\log n}})$. Working independently, K. E. Batcher discovered the general merging strategy outlined above; using a number of comparators defined by

$$c(1) = 0, \quad c(n) = c(\lceil n/2 \rceil) + c(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil, \lfloor n/2 \rfloor) \quad \text{for} \quad n \geq 2, \tag{10}$$

he proved that (see exercise 5.2.2–14)

$$c(2^t) = (t^2 - t + 4)2^{t-2} - 1;$$

and it follows that $\hat{S}(n) = O(n(\log n)^2)$. Neither Batcher nor Floyd and Knuth published their constructions until some time later [*Notices of the Amer. Math. Soc.* **14** (1967), 283; *Proc. AFIPS Spring Joint Computer Conference* **32** (1968), 307–314].

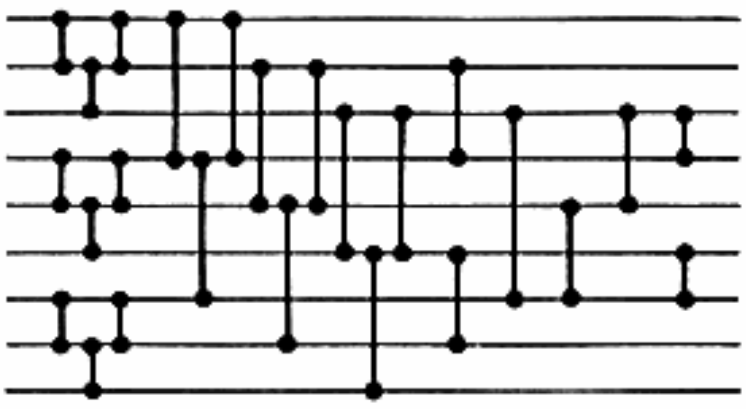
Several people have found ways to improve on the number of comparators used by Batcher's merge-exchange construction; the following table shows the best upper bounds currently known for $\hat{S}(n)$:

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
$c(n) =$	0	1	3	5	9	12	16	19	26	31	37	41	48	53	59	63	(11)
$\hat{S}(n) \leq$	0	1	3	5	9	12	16	19	25	29	35	39	46	51	56	60	

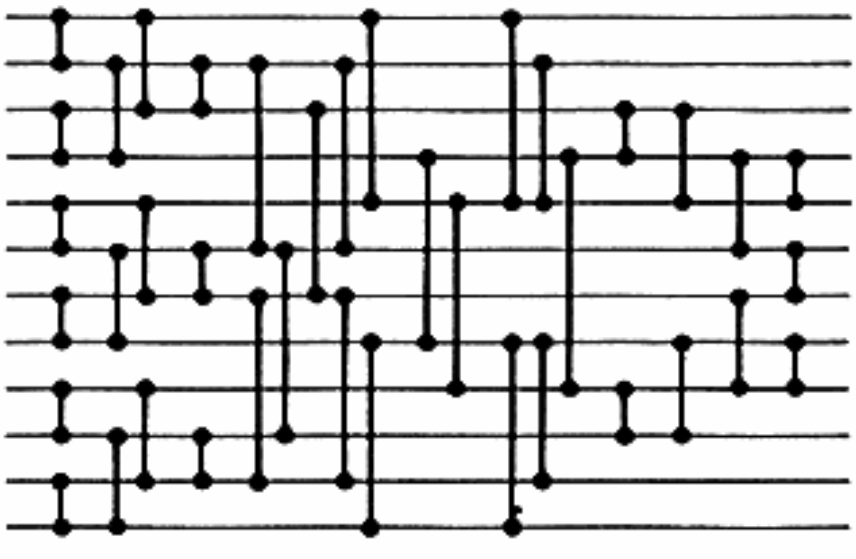
Since $\hat{S}(n) < c(n)$ for $8 < n \leq 16$, merge exchange is nonoptimal for all $n > 8$. When $n \leq 8$, merge exchange is equivalent in the number of comparators to

the construction of Bose and Nelson. Floyd and Knuth proved in 1964–1966 that the above values of $\hat{S}(n)$ are *exact* for $n \leq 8$ [see *A Survey of Combinatorial Theory* (North-Holland, 1972), Chapter 15]; the values of $\hat{S}(n)$ for $n > 8$ are still not known.

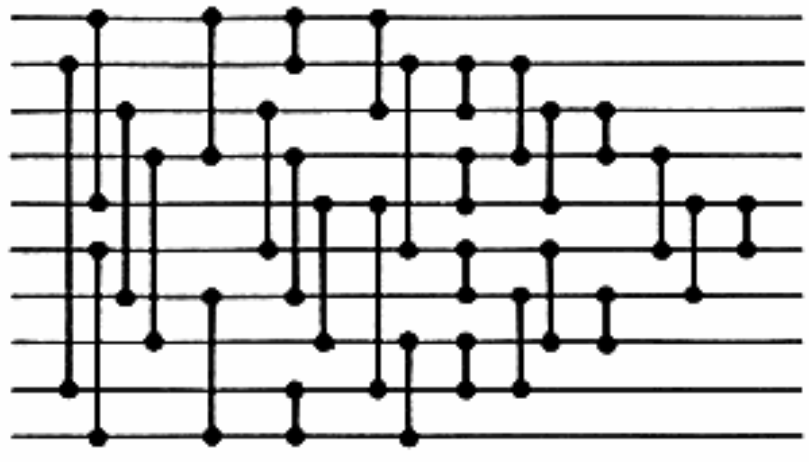
Constructions which lead to the above values for $\hat{S}(n)$ are shown in Fig. 49. The network for $n = 9$, based on an interesting three-way merge, was found by R. W. Floyd in 1964; its validity can be established by using the general principle described in exercise 27. The network for $n = 10$ was discovered by A. Waksman in 1969, by regarding the inputs as permutations of $\{1, 2, \ldots, 10\}$ and trying to reduce as much as possible the number of possible values that can appear on each line at a given stage, while maintaining some symmetry.



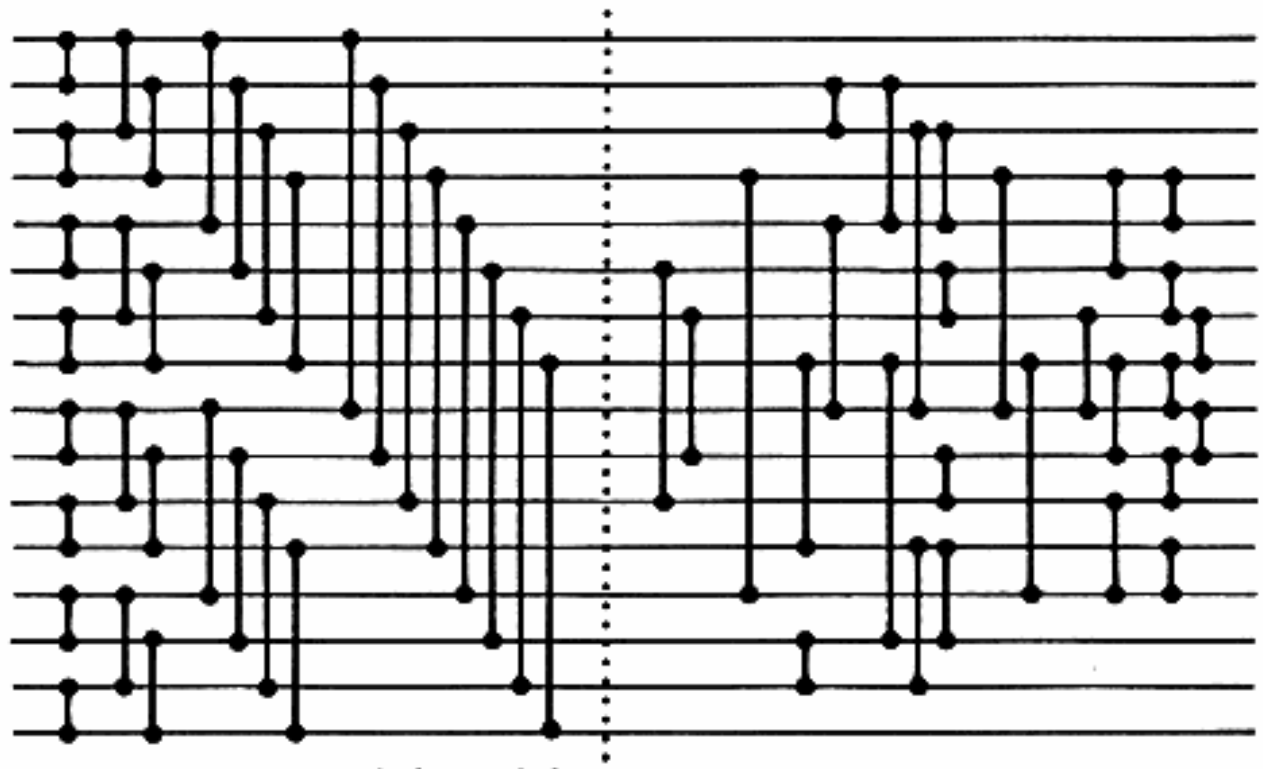
$n = 9$ 25 modules, delay 10



$n = 12$ 39 modules, delay 9



$n = 10$ 29 modules, delay 9



$n = 16$ 60 modules, delay 10

Fig. 49. Efficient sorting networks.

A 62-comparator sorting network for 16 elements was found by G. Shapiro in 1969, and this was rather surprising since Batcher's method (63 comparisons) would appear to be at its best when n is a power of 2. Soon after hearing of Shapiro's construction, M. W. Green tripled the amount of surprise by finding the 60-comparison sorter in Fig. 49. The first portion of Green's construction is fairly easy to understand; after the 32 comparison/interchanges to the left of the dotted line have been made, the lines can be labeled with the 16 subsets of $\{a, b, c, d\}$, in such a way that the line labeled s is known to contain a number less than or equal to the contents of the line labeled t whenever s is a subset of t. The state of the sort at this point is discussed further in exercise 32. Comparisons made on subsequent levels of Green's network become increasingly mysterious, however, and as yet nobody has seen how to generalize the construction in order to obtain correspondingly efficient networks for higher values of n.

Shapiro and Green also discovered the network shown for $n = 12$. When $n = 11$, 13, 14, or 15, good networks can be found by removing the bottom line of the network for $n + 1$, together with all comparators touching that line.

For the best networks currently known as $n \to \infty$, see the Ph.D. thesis of D. Van Voorhis (Stanford University, 1971); his networks require asymptotically $\frac{1}{4}n(\log_2 n)^2 - \alpha n \log_2 n$ comparators, where $\alpha = \frac{1}{4} + \frac{1}{6}\sum_{k\geq 0} 2^{-2^k-k} \approx 0.356852$.

Minimum-time networks. In physical realizations of sorting networks, and on parallel computers, it is possible to do nonoverlapping comparison-exchanges at the same time; therefore it is natural to try to minimize the delay time. A moment's reflection shows that the delay time of a sorting network is equal to the maximum number of comparators in contact with any "path" through the network, if we define a path to consist of any left-to-right route which possibly switches lines at the comparators. We can put a sequence number on each comparator indicating the earliest time it can be executed; this is one higher than the maximum of the sequence numbers of the comparators which precede it. (See Fig. 50(a); part (b) of the figure shows the same network redrawn so that each comparison is done at the earliest possible moment.)

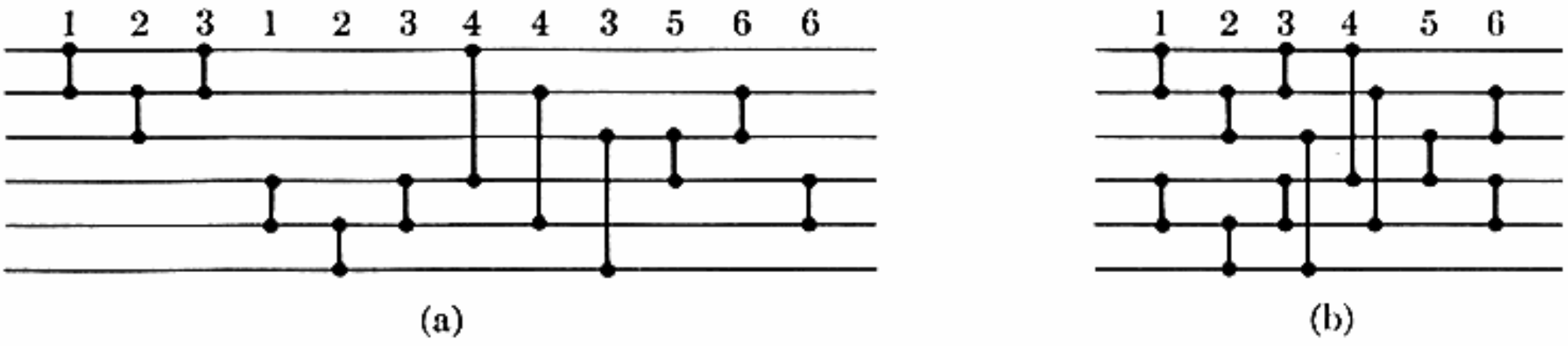


Fig. 50. Doing each comparison at the earliest possible time.

Batcher's odd-even merging network described above takes $T_B(m, n)$ units of time, where $T_B(m, 0) = T_B(0, n) = 0$, $T_B(1, 1) = 1$, and

$$T_B(m, n) = 1 + \max\,(T_B(\lfloor m/2 \rfloor, \lfloor n/2 \rfloor), \quad T_B(\lceil m/2 \rceil, \lceil n/2 \rceil)) \quad \text{for} \quad mn \geq 2.$$

We can use these relations to prove that $T_B(m, n + 1) \geq T_B(m, n)$, by induction; hence $T_B(m, n) = 1 + T_B(\lceil m/2 \rceil, \lceil n/2 \rceil)$ for $mn \geq 2$, and it follows that

$$T_B(m, n) = 1 + \lceil \log_2 \max\,(m, n) \rceil, \quad \text{for} \quad mn \geq 1. \tag{12}$$

Exercise 5 shows that Batcher's sorting method therefore has a delay time of

$$\binom{1 + \lceil \log_2 n \rceil}{2}. \tag{13}$$

Let $\hat{T}(n)$ be the minimum achievable delay time in any sorting network for n elements. It is possible to improve some of the networks described above so that they have less delay time but use no more comparators, as shown for $n = 6$ and $n = 9$ in Fig. 51, and for $n = 10$ in exercise 7. Still smaller delay time can be achieved if we add one or two extra modules, as shown in the remarkable networks for $n = 10$, 12, and 16 in Fig. 51. These constructions yield the following upper bounds on $\hat{T}(n)$ for small n:

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\hat{T}(n) \leq$	0	1	3	3	5	5	6	6	7	7	8	8	9	9	9	9

(14)

For $n \leq 8$ the values given here are known to be exact (see exercise 4). The networks in Fig. 51 merit careful study, because it is by no means obvious that they sort; they were discovered in 1969–1971 by G. Shapiro ($n = 6$, 9, 12) and D. Van Voorhis ($n = 10$, 16).

Merging networks. Let $\hat{M}(m, n)$ denote the minimum number of comparator modules needed in a network that merges m elements $x_1 \leq \cdots \leq x_m$ with n elements $y_1 \leq \cdots \leq y_n$ to form the sorted sequence $z_1 \leq \cdots \leq z_{m+n}$. At present no merging networks have been discovered which are superior to the odd-even merge described above; hence the function $C(m, n)$ in (6) represents the best upper bound known for $\hat{M}(m, n)$.

R. W. Floyd has discovered an interesting way to find *lower* bounds for this merging problem.

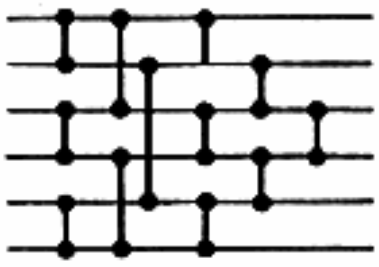
Theorem F. *$\hat{M}(2n, 2n) \geq 2\hat{M}(n, n) + n$, for all $n \geq 1$.*

Proof. Consider a network with $\hat{M}(2n, 2n)$ comparator modules, capable of sorting all input sequences $\langle z_1, \ldots, z_{4n} \rangle$ such that $z_1 \leq z_3 \leq \cdots \leq z_{4n-1}$ and $z_2 \leq z_4 \leq \cdots \leq z_{4n}$. We may assume that each module replaces (z_i, z_j) by $(\min\,(z_i, z_j), \max\,(z_i, z_j))$, for some $i < j$ (see exercise 16). The comparators can therefore be divided into three classes:

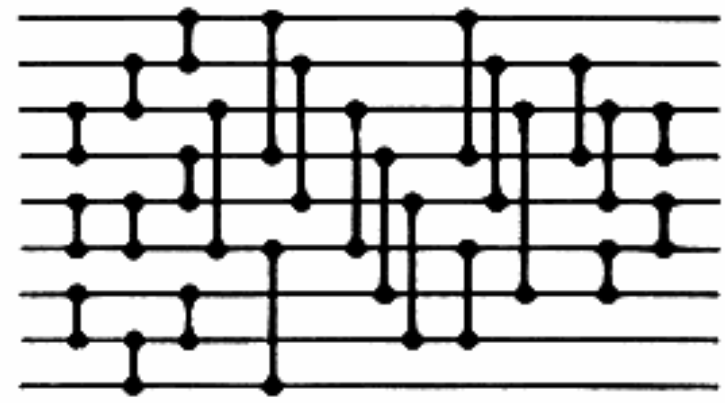
a) $i \leq 2n$ and $j \leq 2n$.

b) $i > 2n$ and $j > 2n$.

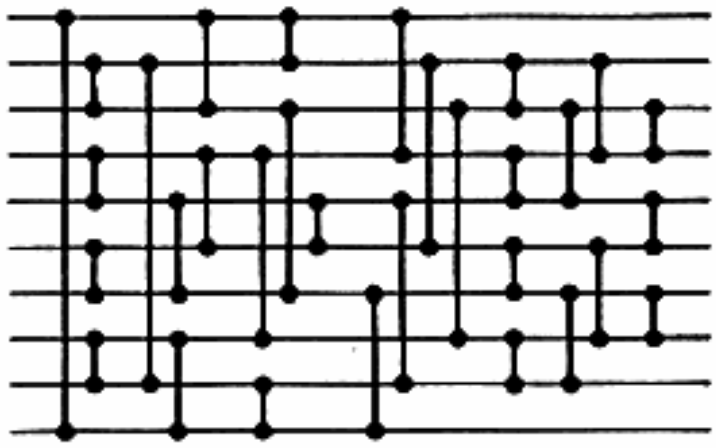
c) $i \leq 2n$ and $j > 2n$.



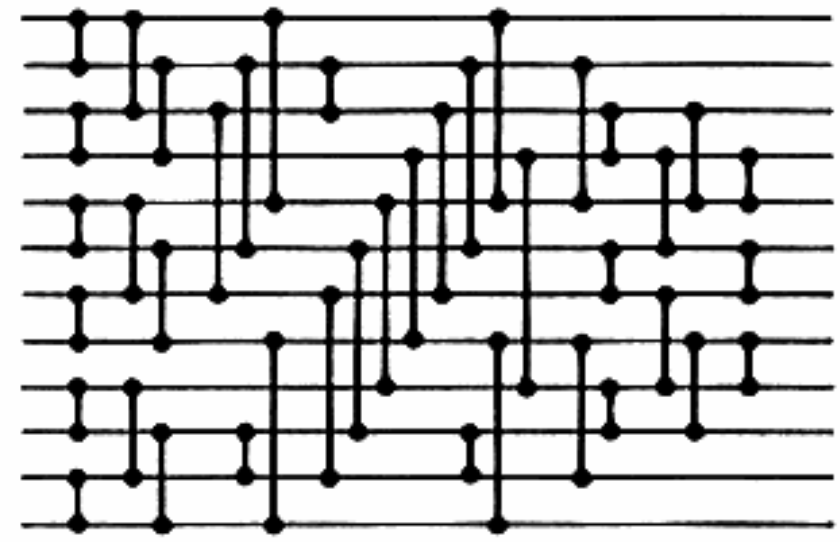
$n = 6$ 12 modules, delay 5



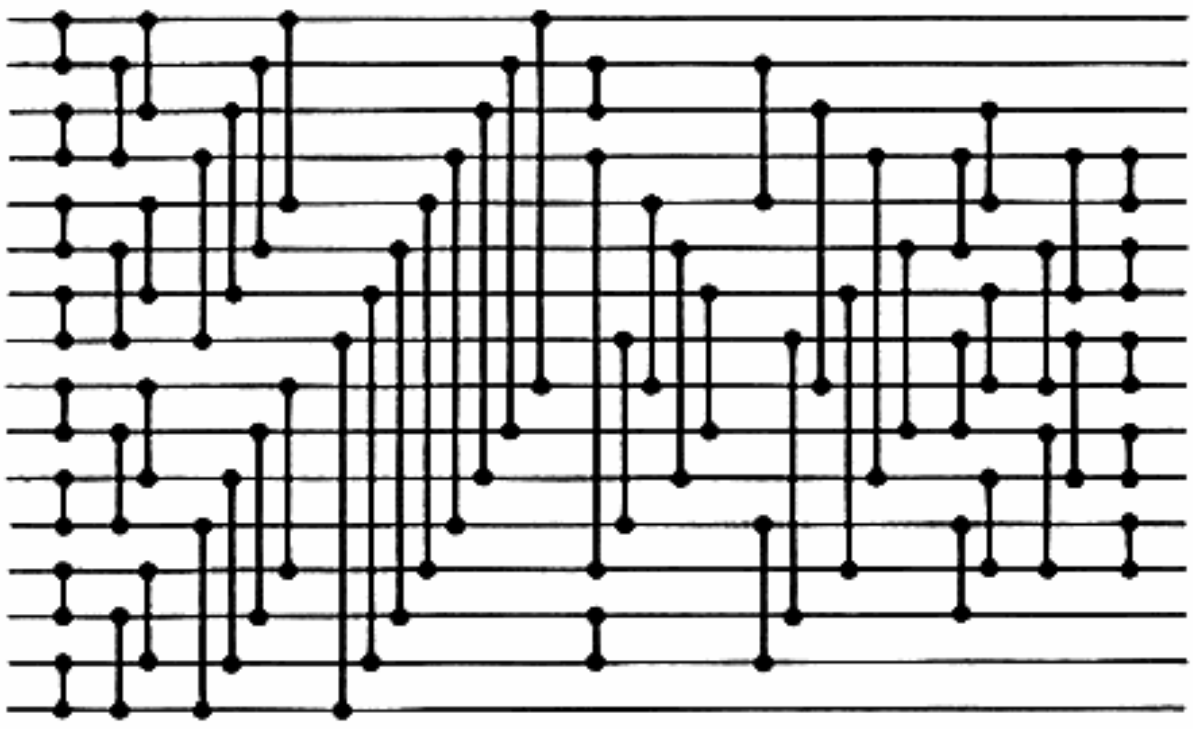
$n = 9$ 25 modules, delay 8



$n = 10$ 31 modules, delay 7



$n = 12$ 40 modules, delay 8



$n = 16$ 61 modules, delay 9

Fig. 51. Sorting networks which are unusually fast, when comparisons are performed in parallel.

Class (a) must contain at least $\hat{M}(n, n)$ comparators, since $z_{2n+1}, z_{2n+2}, \ldots, z_{4n}$ may be already in their final position when the merge starts; similarly, there are at least $\hat{M}(n, n)$ comparators in class (b). Furthermore the input sequence $\langle 0, 1, 0, 1, \ldots, 0, 1\rangle$ shows that class (c) contains at least n comparators, since n zeros must move from $\{z_{2n+1}, \ldots, z_{4n}\}$ to $\{z_1, \ldots, z_{2n}\}$. ∎

Repeated use of Theorem F proves that $\hat{M}(2^m, 2^m) \geq \frac{1}{2}(m + 2)2^m$; hence $\hat{M}(n, n) \geq \frac{1}{2}n \log_2 n + O(n)$. Merging *without* the network restriction requires only $M(n, n) = 2n - 1$ comparisons; hence we have proved that merging with networks is intrinsically harder than merging in general. The odd-even merge

shows that $\hat{M}(n, n) \leq C(n, n) = n \log_2 n + O(n)$, so the asymptotic behavior of $\hat{M}(n, n)$ is known within a factor of 2. (The exact value of $\hat{M}(n, n)$ is known for $n \leq 5$, see exercise 9.)

Bitonic sorting. When simultaneous comparisons are allowed, we have seen in Eq. (12) that the odd-even merge uses $\lceil \log_2 (2n) \rceil$ units of delay time, when $1 \leq m \leq n$. Batcher has devised another type of network for merging, called a *bitonic sorter*, which lowers the delay time to $\lceil \log_2 (m + n) \rceil$ although it requires more comparator modules.

Let us say that a sequence $\langle z_1, \ldots, z_p \rangle$ of p numbers is *bitonic* if $z_1 \geq \cdots \geq z_k \leq \cdots \leq z_p$ for some k, $1 \leq k \leq p$. (Compare this with the ordinary definition of "monotonic" sequences.) A bitonic sorter of order p is a comparator network capable of sorting any bitonic sequence of length p into nondecreasing order. The problem of merging $x_1 \leq \cdots \leq x_m$ with $y_1 \leq \cdots \leq y_n$ is a special case of the bitonic sorting problem, since merging can be done by applying a bitonic sorter of order $m + n$ to the sequence $\langle x_m, \ldots, x_1, y_1, \ldots, y_n \rangle$.

Note that when the sequence $\langle z_1, \ldots, z_p \rangle$ is bitonic, so are all of its subsequences. Shortly after Batcher discovered the odd-even merging networks, he observed that we can construct a bitonic sorter of order p in an analogous way, by first sorting the bitonic subsequences $\langle z_1, z_3, z_5, \ldots \rangle$ and $\langle z_2, z_4, z_6, \ldots \rangle$ independently, then comparing and interchanging $z_1 : z_2$, $z_3 : z_4$, ... (See exercise 10 for a proof.) If $C'(p)$ is the corresponding number of comparator modules, we have

$$C'(p) = C'(\lceil p/2 \rceil) + C'(\lfloor p/2 \rfloor) + \lfloor p/2 \rfloor, \qquad \text{for} \qquad p \geq 2; \tag{15}$$

and the delay time is clearly $\lceil \log_2 p \rceil$. Figure 52 shows the bitonic sorter of order 7 constructed in this way: it can be used as a (3, 4)- as well as a (2, 5)-merging network, with three units of delay; the odd-even merge for $m = 2$ and $n = 5$ has one less comparator but one more level of delay.

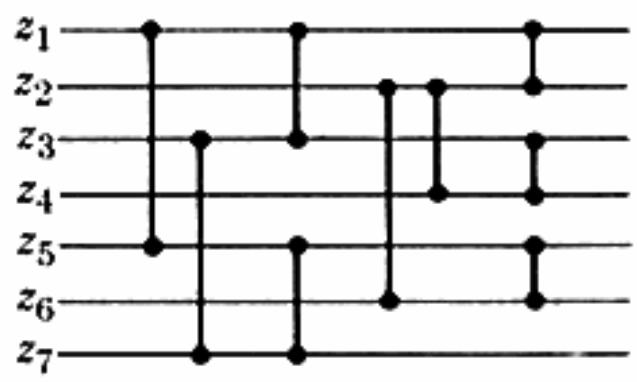


Fig. 52. Batcher's bitonic sorter of order 7.

Batcher's bitonic sorter of order 2^k is particularly interesting; it consists of k levels of 2^{k-1} comparators each. If we number the input lines $z_0, z_1, \ldots, z_{2^k-1}$, element z_i is compared to z_j on level l if and only if i and j differ only in the lth most significant bit of their binary representations. This simple structure leads to parallel sorting networks which are as fast as merge exchange (Algorithm 5.2.2M) but considerably easier to implement. (See exercises 11 and 13.)

When $m = n$, it is not difficult to show that both the odd-even merge and Batcher's bitonic sorter provide the absolute minimum delay time achievable in any merging network. The nth smallest output of an (n, n) merging network must depend upon all $2n$ of the inputs, and if it can be computed in l steps, it involves at most 2^l of the inputs; hence $2^l \geq 2n$, $l \geq \lceil \log_2 (2n) \rceil$.

When $m < n$, the nth output of an (m, n) merging network depends on $2m + 1$ of the inputs (cf. exercise 29), so the same argument gives a minimum delay time of $\lceil \log_2 (2m + 1) \rceil$ for merging in this case. Batcher has shown [Report GER-14122 (Akron, Ohio: Goodyear Aerospace Corporation, 1968)] that this minimum delay time is achievable if we allow "multiple fanout" in the network, i.e., the splitting of lines so that the same number is fed to many modules at the same time. For example, his network which is capable of merging one item with n others in only two levels of delay is illustrated for $n = 6$ in Fig. 53. Of course, networks with multiple fanout do not conform to our conventions, and it is fairly easy to see that any $(1, n)$ merging network without multiple fanout must have a delay time of $\log_2 (n + 1)$ or more. (See exercise 14.)

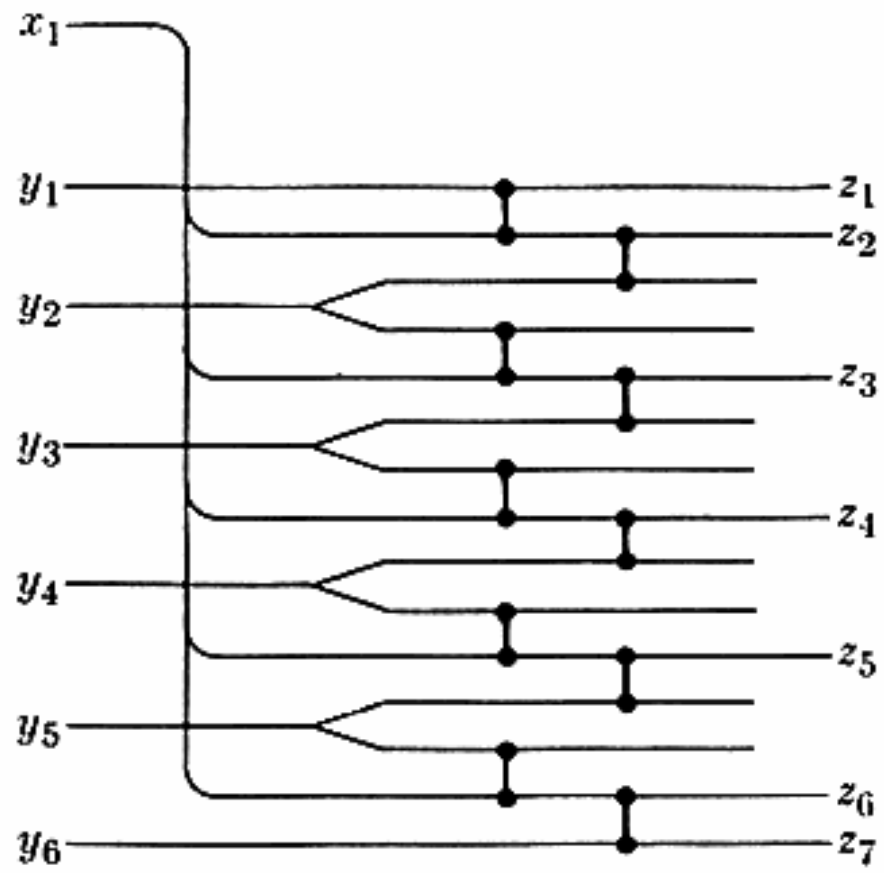


Fig. 53. Merging one item with six others, with multiple fanout, in order to achieve the minimum possible delay time.

Selection networks. We can also use networks to approach the problem of Section 5.3.3. Let $\hat{U}_t(n)$ denote the minimum number of comparators required in a network which moves the t largest of n distinct inputs into t specified output lines; they are allowed to appear in any order on these output lines. Let $\hat{V}_t(n)$ denote the minimum number of comparators required to move the tth largest of n distinct inputs into a specified output line; and let $\hat{W}_t(n)$ denote the minimum number of comparators required to move the t largest of n distinct inputs into t specified output lines in nondecreasing order. It is not difficult to see (cf. exercise 17) that

$$\hat{U}_t(n) \leq \hat{V}_t(n) \leq \hat{W}_t(n). \tag{16}$$

Suppose first that we have $2t$ elements $\langle x_1, \ldots, x_{2t}\rangle$ and we wish to select the largest t; V. E. Alekseyev [*Kibernetika* **5**, 5 (1969), 99–103] has observed that we can do the job by first sorting $\langle x_1, \ldots, x_t\rangle$ and $\langle x_{t+1}, \ldots, x_{2t}\rangle$, then comparing and interchanging

$$x_1 : x_{2t}, \qquad x_2 : x_{2t-1}, \qquad \ldots, \qquad x_t : x_{t+1}. \tag{17}$$

Since none of these pairs can contain more than one of the largest t elements (why?), Alekseyev's procedure must select the largest t elements.

If we want to select the t largest of nt elements, we can apply the procedure $n - 1$ times (eliminating t elements each time); hence

$$\hat{O}_t(nt) \le (n-1)(2\hat{S}(t) + t). \tag{18}$$

Alekseyev has also derived an interesting *lower* bound for the selection problem.

Theorem A. $\hat{O}_t(n) \ge (n-t)\lceil \log_2(t+1)\rceil$.

Proof. It is most convenient to consider the equivalent problem of selecting the *smallest* t elements. We can attach numbers (l, u) to each line of a comparator network, as shown in Fig. 54, where l and u denote respectively the minimum and maximum values that can appear at that position when the input is a permutation of $\{1, 2, \ldots, n\}$. Let l_i and l_j be the lower bounds on lines i and j before a comparison of $x_i : x_j$, and let l_i' and l_j' be the corresponding lower bounds after the comparison. It is obvious that $l_i' = \min(l_i, l_j)$, and exercise 24 proves the (nonobvious) relation

$$l_j' \le l_i + l_j. \tag{19}$$

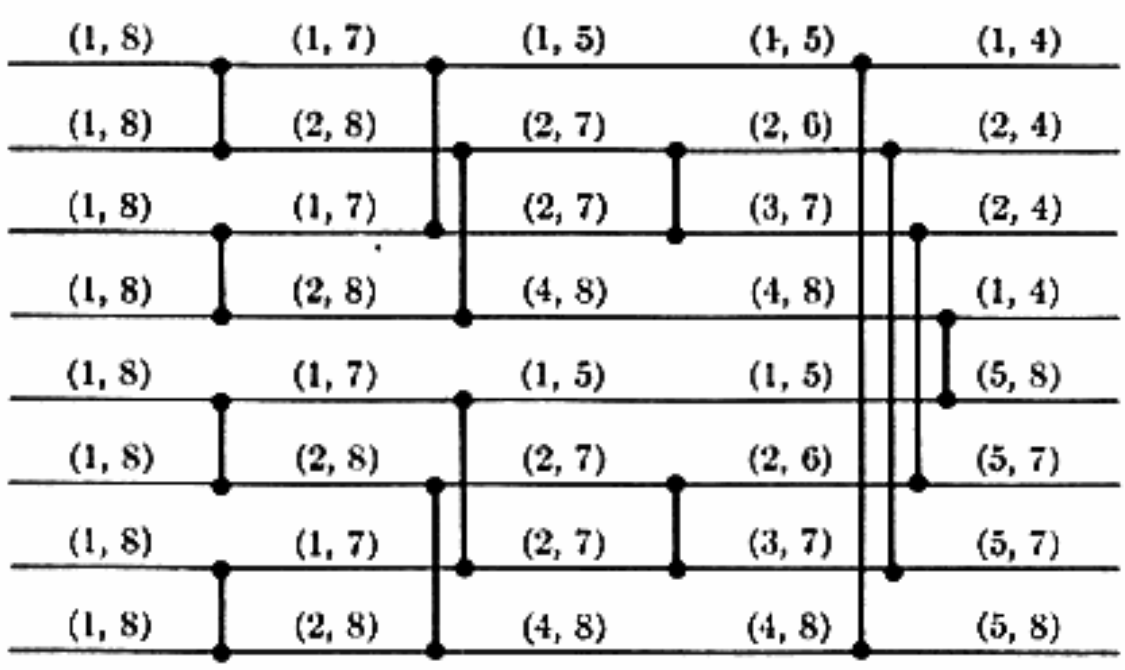


Fig. 54. Separating the largest four from the smallest four. (Numbers on these lines are used in the proof of Theorem A.)

Now let us reinterpret the network operations in another way (see Fig. 55): All input lines are assumed to contain zero, and each "comparator" now places the smaller of its inputs on the upper line and the larger *plus one* on the lower line. The resulting numbers $\langle m_1, m_2, \ldots, m_n \rangle$ have the property that

$$2^{m_i} \geq l_i \tag{20}$$

throughout the network, since this holds initially and it is preserved by each comparator because of (19). Furthermore, the final value of

$$m_1 + m_2 + \cdots + m_n$$

is the total number of comparators in the network, since each comparator adds unity to this sum.

If the network selects the smallest t numbers, $n - t$ of the l_i are $\geq t + 1$; hence $n - t$ of the m_i must be $\geq \lceil \log_2 (t + 1) \rceil$. ∎

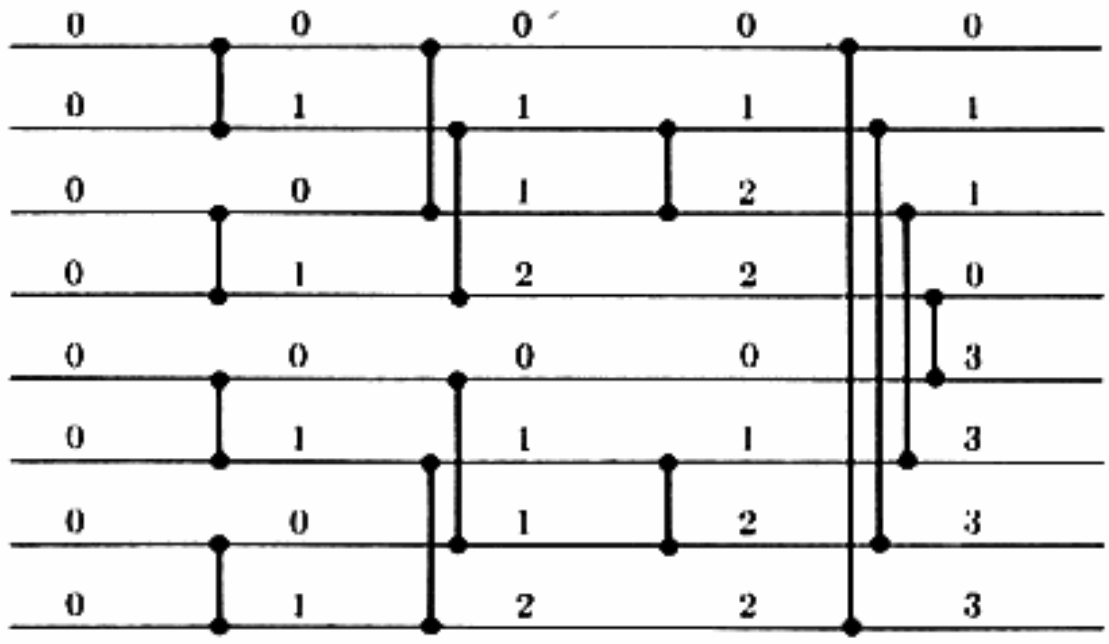


Fig. 55. Another interpretation for the network of Fig. 54.

The lower bound in Theorem A turns out to be exact when $t = 1$ and when $t = 2$ (see exercise 19). Table 1 gives some values of $\hat{U}_t(n)$, $\hat{V}_t(n)$, and $\hat{W}_t(n)$ for small t and n.

Table 1

COMPARISONS NEEDED IN SELECTION NETWORKS $(\hat{U}_t(n), \hat{V}_t(n), \hat{W}_t(n))$

	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$	$t = 6$
$n = 1$	(0, 0, 0)					
$n = 2$	(1, 1, 1)	(0, 1, 1)				
$n = 3$	(2, 2, 2)	(2, 3, 3)	(0, 2, 3)			
$n = 4$	(3, 3, 3)	(4, 5, 5)	(3, 5, 5)	(0, 3, 5)		
$n = 5$	(4, 4, 4)	(6, 7, 7)	(6, 7, 8)	(4, 7, 9)	(0, 4, 9)	
$n = 6$	(5, 5, 5)	(8, 9, 9)	(8, 10, 10)	(8, 10, 12)	(5, 9, 12)	(0, 5, 12)

EXERCISES—First Set

Several of the following exercises develop the theory of sorting networks in detail, and it is convenient to introduce some notation. We let $[i:j]$ stand for a comparison/interchange module. A network with n inputs and r comparator modules is written $[i_1:j_1][i_2:j_2]\ldots[i_r:j_r]$, where each of the i's and j's is $\leq n$; we shall call it an *n-network* for short. A network is called *standard* if $i_q < j_q$ for $1 \leq q \leq r$. Thus, for example, Fig. 44 on page 222 depicts a standard 4-network which is denoted by the comparator sequence [1:2][3:4][1:3][2:4][2:3].

The text's convention for drawing network diagrams represents only standard networks; all comparators $[i:j]$ are represented by a line from i to j, where $i < j$. When nonstandard networks must be drawn, we can use an *arrow* from i to j, indicating that the larger number goes to the point of the arrow. For example, Fig. 56 illustrates a nonstandard network for 16 elements, whose comparators are [1:2][4:3][5:6][8:7] etc. Exercise 11 proves that Fig. 56 is a sorting network.

If $x = \langle x_1, \ldots, x_n\rangle$ is an n-vector and α is an n-network, we write $x\alpha$ for the vector of numbers $\langle (x\alpha)_1, \ldots, (x\alpha)_n\rangle$ produced by the network. For brevity, we also let $a \vee b = \max(a, b)$, $a \wedge b = \min(a, b)$, $\bar{a} = 1 - a$. Thus $(x[i:j])_i = x_i \wedge x_j$, $(x[i:j])_j = x_i \vee x_j$, and $(x[i:j])_k = x_k$ when $i \neq k \neq j$. We say α is a *sorting network* iff $(x\alpha)_i \leq (x\alpha)_{i+1}$ for all x and for $1 \leq i < n$.

The symbol $e^{(i)}$ stands for a vector which is 1 in position i, 0 elsewhere; thus $(e^{(i)})_j = \delta_{ij}$. The symbol D_n stands for the set of all 2^n n-place vectors of 0's and 1's, and P_n stands for the set of all $n!$ vectors which are permutations of $\{1, 2, \ldots, n\}$. We write $x \wedge y$ and $x \vee y$ for the vectors $\langle x_1 \wedge y_1, \ldots, x_n \wedge y_n\rangle$ and $\langle x_1 \vee y_1, \ldots, x_n \vee y_n\rangle$, and we write $x \leq y$ if $x_i \leq y_i$ for all i. Thus $x \leq y$ iff $x \vee y = y$ iff $x \wedge y = x$. If x and y are in D_n, we say that x *covers* y if $x = y \vee e^{(i)} \neq y$ for some i. Finally for all x in D_n we let $\nu(x)$ be the number of 1's in x, and $\zeta(x)$ the number of 0's; thus $\nu(x) + \zeta(x) = n$.

1. [*20*] Draw a network diagram for the odd-even merge when $m = 3$ and $n = 5$.

2. [*22*] Show that V. Pratt's sorting algorithm (exercise 5.2.1–30) leads to a sorting network for n elements which has approximately $(\log_2 n)(\log_3 n)$ levels of delay. Draw the corresponding network for $n = 12$.

3. [*M20*] (K. E. Batcher.) Find a simple relation between $C(m, m-1)$ and $C(m, m)$.

▸ **4.** [*M23*] Prove that $\hat{T}(6) = 5$.

5. [*M21*] Prove that (13) is the delay time associated with the sorting network outlined in (10).

6. [*28*] Let $T(n)$ be the minimum number of stages needed to sort by making *simultaneous disjoint comparisons* (without the network constraint); each such set of comparisons can be represented as a node containing a set of pairs $i_1:j_1$, $i_2:j_2$, $\ldots$, $i_r:j_r$ where $i_1, j_1, i_2, j_2, \ldots, i_r, j_r$ are distinct, with 2^r branches below this node for the respective cases

$$\langle K_{i_1} < K_{j_1}, K_{i_2} < K_{j_2}, \ldots, K_{i_r} < K_{j_r}\rangle,$$
$$\langle K_{i_1} > K_{j_1}, K_{i_2} < K_{j_2}, \ldots, K_{i_r} < K_{j_r}\rangle, \qquad \text{etc.}$$

Prove that $T(5) = T(6) = 5$.

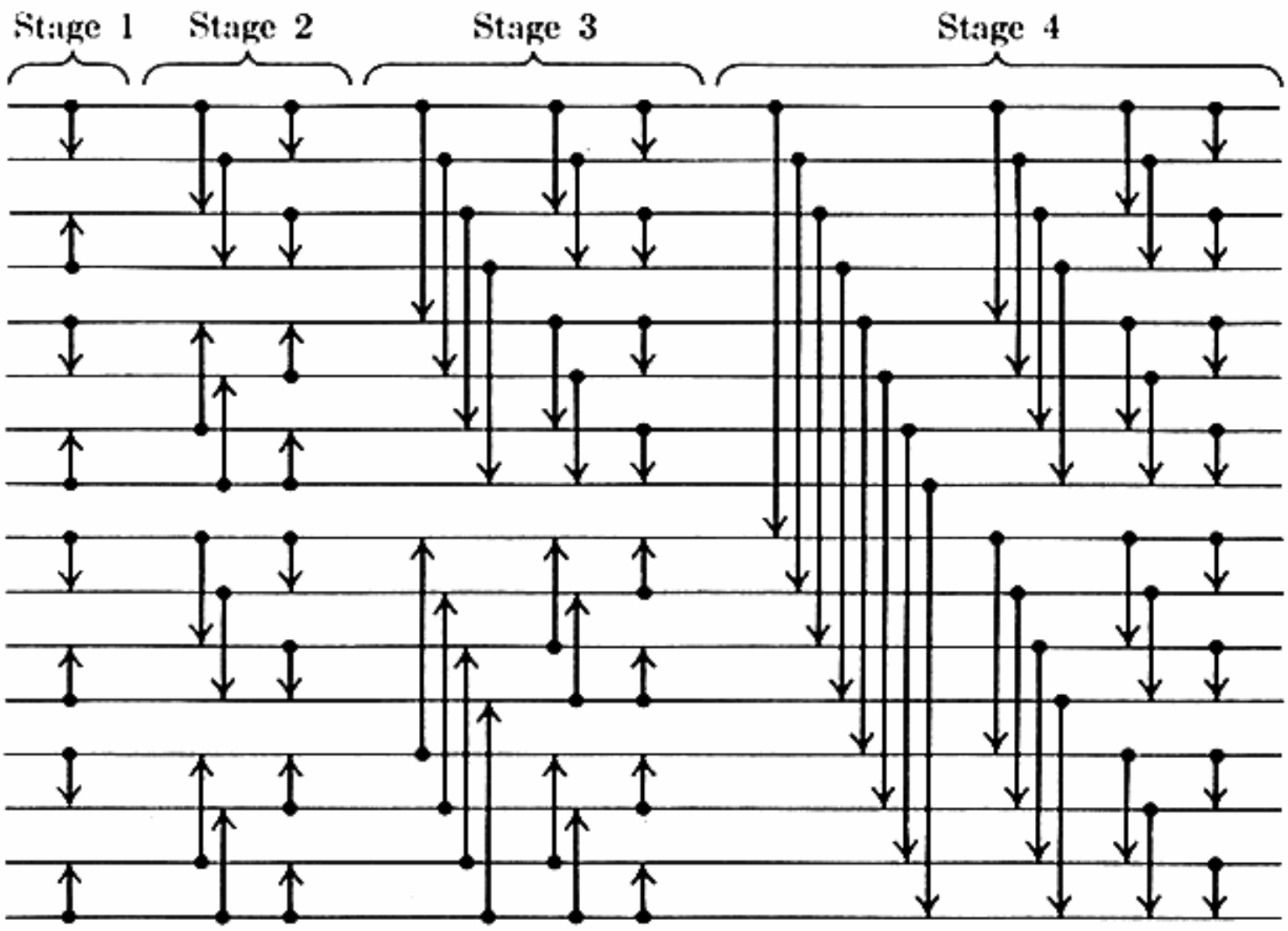


Fig. 56. A nonstandard sorting network based on bitonic sorting.

7. [*25*] Show that if the final comparator of the network for $n = 10$ in Fig. 49 is moved just ahead of the second- and third-last comparators, the network will still sort.

8. [*M20*] Prove that $\hat{M}(m_1 + m_2, n_1 + n_2) \geq \hat{M}(m_1, n_1) + \hat{M}(m_2, n_2) + \min(m_1, n_2)$, for $m_1, m_2, n_1, n_2 \geq 0$.

9. [*M25*] (R. W. Floyd.) Prove that $\hat{M}(3, 3) = 6$, $\hat{M}(4, 4) = 9$, $\hat{M}(5, 5) = 13$.

10. [*M22*] Prove that Batcher's bitonic sorter, as defined in the remarks preceding (15), is valid. [*Hint:* It is only necessary to prove that all sequences consisting of k 1's followed by l 0's followed by $n - k - l$ 1's will be sorted.]

11. [*M23*] Prove that Batcher's bitonic sorter of order 2^p will not only sort sequences $\langle z_0, z_1, \ldots, z_{2^p-1}\rangle$ for which $z_0 \geq \cdots \geq z_k \leq \cdots \leq z_{2^p-1}$, it also will sort any sequence for which $z_0 \leq \cdots \leq z_k \geq \cdots \geq z_{2^p-1}$. [As a consequence, the network in Fig. 56 will sort 16 elements, since each stage separated by dotted lines consists of bitonic sorters or reverse-order bitonic sorters, applied to sequences that have been sorted in opposite directions.]

12. [*M20*] Prove or disprove: If x and y are bitonic sequences of the same length, so are $x \vee y$ and $x \wedge y$.

▸**13.** [*24*] (H. S. Stone.) Show that a sorting network for 2^t elements can be constructed by following the pattern illustrated for $t = 4$ in Fig. 57. Each of the t^2 steps in this scheme consists of a "perfect shuffle" of the first 2^{t-1} elements with the last 2^{t-1}, followed by simultaneous operations performed on 2^{t-1} pairs of adjacent elements. Each of the latter operations is either "0" (no operation), "+" (a standard comparator module), or "−" (a reverse comparator module). The sorting proceeds in t stages of t steps each; during the last stage all operations are "+". During stage s, for $s < t$, we do $t - s$ steps in which all operations are "0", followed by s steps in which the operations within step 8 consist alternately of 2^q "+" followed by 2^q "−", for $q = 1, 2, \ldots, s$.

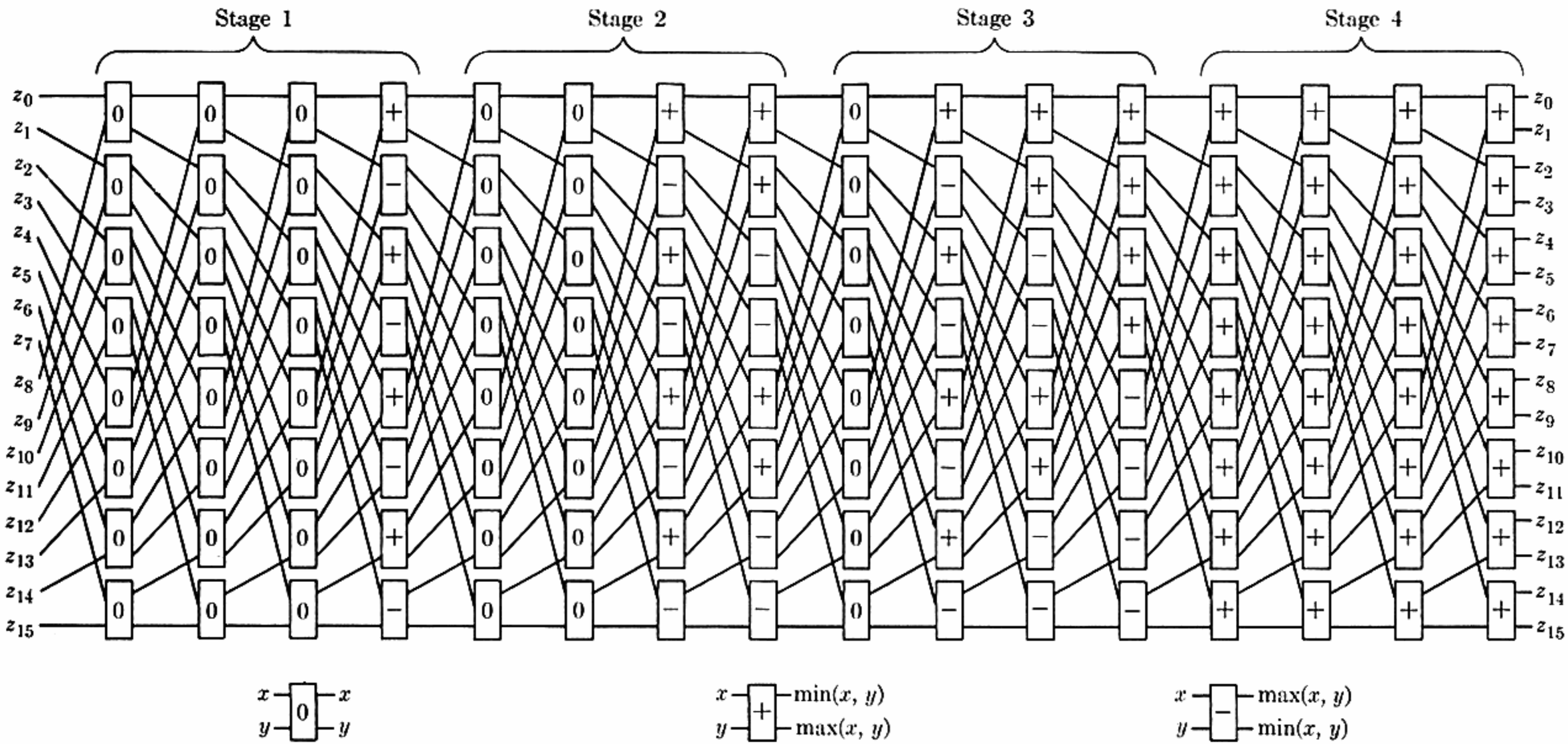


Fig. 57. Sorting 16 elements with "perfect shuffles."

[Note that this sorting scheme could be performed by a fairly simple device whose circuitry performs one "shuffle-and-operate" step and feeds the output lines back into the input. The first three steps in Fig. 57 could of course be eliminated; they have been retained only to make the pattern clear. Stone notes that the same pattern "shuffle/operate" occurs in several other algorithms, such as the fast Fourier transform (cf. Section 4.6.4).]

14. [*M20*] Prove that any $(1, n)$-merging network without multiple fanout must have at least $\lceil \log_2 (n+1) \rceil$ levels of delay.

15. [*20*] Find a nonstandard sorting network for four elements which has only five comparator modules.

16. [*M22*] Prove that the following algorithm transforms any sorting network $[i_1{:}j_1] \ldots [i_r{:}j_r]$ into a standard sorting network:

T1. Let q be the smallest index such that $i_q > j_q$. If no such index exists, stop.

T2. Change all occurrences of i_q to j_q, and all occurrences of j_q to i_q, in all comparators $[i_s{:}j_s]$ for $q \le s \le r$. Return to T1. ∎

For example, the network [4:1][3:2][1:3][2:4][1:2][3:4] is first transformed into [1:4][3:2][4:3][2:1][4:2][3:1], then [1:4][2:3][4:2][3:1][4:3][2:1], then [1:4][2:3][2:4][3:1][2:3][4:1], etc., until the standard network [1:4][2:3][2:4][1:3][1:2][3:4] is obtained.

17. [*M25*] Let D_{tn} be the set of all $\binom{n}{t}$ sequences $\langle x_1, \ldots, x_n \rangle$ of 0's and 1's having exactly t 1's. Show that $\hat{U}_t(n)$ is the minimum number of comparators needed in a network which sorts all the elements of D_{tn}; $\hat{V}_t(n)$ is the minimum number needed to sort $D_{tn} \cup D_{(t-1)n}$; and $\hat{W}_t(n)$ is the minimum number needed to sort $\bigcup_{0 \le k \le t} D_{kn}$.

▸ **18.** [*M20*] Prove that a network which finds the median of $2t - 1$ elements requires at least $(t-1)\lceil \log_2 (t+1) \rceil + \lceil \log_2 t \rceil$ comparator modules. [*Hint:* See the proof of Theorem A.]

19. [*M22*] Prove that $\hat{U}_2(n) = 2n - 4$ and $\hat{V}_2(n) = 2n - 3$, for all $n \ge 2$.

20. [*24*] Prove that $\hat{V}_3(5) = 7$.

21. [*M15*] Let α be any n-network, and let x and y be n-vectors. Prove that $x \le y$ implies that $x\alpha \le y\alpha$.

22. [*M15*] Prove that if x and y are n-vectors of real numbers, then $x \cdot y \le (x\alpha) \cdot (y\alpha)$. (Here $x \cdot y$ is the dot product, $x_1y_1 + \cdots + x_ny_n$.)

23. [*M17*] Let α be an n-network. Prove that there is a permutation $p \in P_n$ such that $(p\alpha)_i = j$ if and only if there are vectors x, y in D_n such that x covers y, $(x\alpha)_i = 1$, $(y\alpha)_i = 0$, and $\zeta(y) = j$.

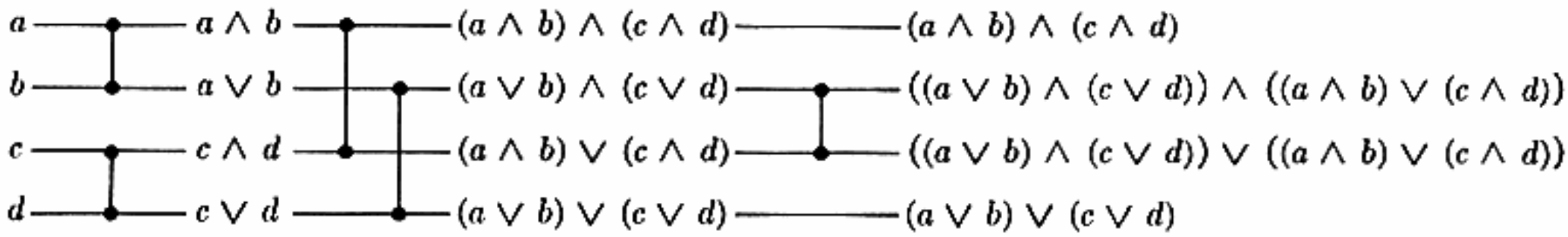
▸ **24.** [*M21*] (V. E. Alekseyev.) Let α be an n-network, and for $1 \le k \le n$ let $l_k = \min \{(p\alpha)_k \mid p \text{ in } P_n\}$, $u_k = \max \{(p\alpha)_k \mid p \text{ in } P_n\}$ denote the lower and upper bounds on the range of values which may appear in line k of the output. Let l'_k and u'_k be defined similarly for the network $\alpha' = \alpha[i{:}j]$. Prove that $l'_i = l_i \wedge l_j$, $l'_j \le l_i + l_j$, $u'_i \ge u_i + u_j - (n+1)$, $u'_j = u_i \vee u_j$. [*Hint:* Given vectors x and y in D_n with $(x\alpha)_i = (y\alpha)_j = 0$, $\zeta(x) = l_i$, $\zeta(y) = l_j$, find a vector z in D_n with $(z\alpha')_j = 0$, $\zeta(z) \le l_i + l_j$.]

25. [*M30*] Let l_k and u_k be as defined in exercise 24. Prove that the set $\{(p\alpha)_k \mid p \text{ in } P_n\}$ includes all integers between l_k and u_k inclusive.

26. [*M24*] (R. W. Floyd.) Let α be an n-network. Prove that the set $D_n\alpha = \{x\alpha \mid x \text{ in } D_n\}$ can be determined from the set $P_n\alpha = \{p\alpha \mid p \text{ in } P_n\}$; and conversely, $P_n\alpha$ can be determined from $D_n\alpha$.

▶**27.** [*M20*] Let x and y be vectors, and let $x\alpha$ and $y\alpha$ be sorted. Prove that $(x\alpha)_i \le (y\alpha)_j$ if and only if, for every choice of j elements from y we can choose i elements from x, such that every chosen x element is $\le$ some chosen y element. Use this principle to prove that *if we sort the rows of any matrix, then sort the columns, the rows will remain in order.*

▶**28.** [*M20*] The following diagram shows that it is possible to write down formulas for the contents of all lines in a sorting network in terms of the inputs:



Using the commutative laws $x \wedge y = y \wedge x$, $x \vee y = y \vee x$, the associative laws $x \wedge (y \wedge z) = (x \wedge y) \wedge z$, $x \vee (y \vee z) = (x \vee y) \vee z$, the distributive laws $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$, the absorption laws $x \wedge (x \vee y) = x \vee (x \wedge y) = x$, and the idempotent laws $x \wedge x = x \vee x = x$, we can reduce the formulas at the right of this network to $(a \wedge b \wedge c \wedge d)$, $(a \wedge b \wedge c) \vee (a \wedge b \wedge d) \vee (a \wedge c \wedge d) \vee (b \wedge c \wedge d)$, $(a \wedge b) \vee (a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d) \vee (c \wedge d)$, $a \vee b \vee c \vee d$, respectively.

Prove that, in general, the tth largest element of $\{x_1, \ldots, x_n\}$ is given by the "elementary symmetric function"

$$\sigma_t(x_1, \ldots, x_n) = \vee\{x_{i_1} \wedge x_{i_2} \wedge \ldots \wedge x_{i_t} \mid 1 \le i_1 < i_2 < \cdots < i_t \le n\}.$$

[There are $\binom{n}{t}$ terms being $\vee$'d together. Thus the problem of finding minimum-cost sorting networks is equivalent to the problem of computing the elementary symmetric functions with a minimum number of "and/or" circuits, where at every stage we replace two quantities ϕ and ψ by $\phi \wedge \psi$ and $\phi \vee \psi$.]

29. [*M20*] Given that $x_1 \le x_2 \le x_3$ and $y_1 \le y_2 \le y_3 \le y_4 \le y_5$, and that $z_1 \le z_2 \le \cdots \le z_8$ is the result of merging the x's with the y's, find formulas for each of the z's in terms of the x's and the y's, using the operators $\wedge$ and $\vee$.

30. [*HM24*] Prove that any formula involving $\wedge$ and $\vee$ and the independent variables $\{x_1, \ldots, x_n\}$ can be reduced using the identities in exercise 28 to a "canonical" form $\tau_1 \vee \tau_2 \vee \cdots \vee \tau_k$, where $k \ge 1$, each τ_i has the form $\wedge\{x_j \mid j \text{ in } S_i\}$ where S_i is a subset of $\{1, 2, \ldots, n\}$, and no set S_i is included in S_j for $i \ne j$. Prove also that two such canonical forms are equal for all $x_1, \ldots, x_n$ if and only if they are identical (up to order).

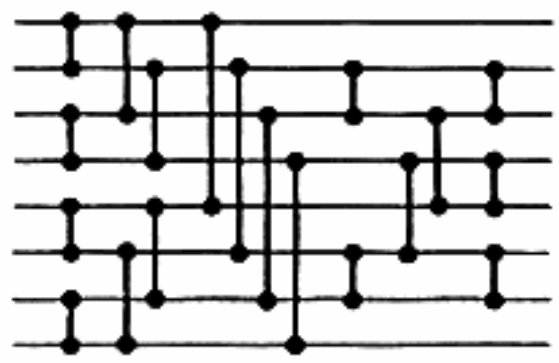
31. [*M24*] (R. Dedekind, 1897.) Let δ_n be the number of distinct canonical forms on $x_1, \ldots, x_n$ in the sense of exercise 30. Thus $\delta_1 = 1$, $\delta_2 = 4$, and $\delta_3 = 18$. What is δ_4?

32. [*M28*] (M. W. Green.) Let $G_1 = \{00, 01, 11\}$, and let G_{n+1} be the set of all strings $\theta\phi\psi\omega$ such that θ, ϕ, ψ, ω have length 2^{n-1} and $\theta\phi$, $\psi\omega$, $\theta\psi$, and $\phi\omega$ are in G_n.

Let α be the network consisting of the first four levels of the 16-sorter shown in Fig. 48. Show that $D_{16}\alpha = G_4$, and prove that it has exactly $\delta_4 + 2$ elements. (Cf. exercise 31.)

▸ **33.** [*M22*] Not all δ_n of the functions of $\langle x_1, \ldots, x_n \rangle$ in exercise 31 can appear in comparator networks. In fact, prove that the function $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4)$ cannot appear as an output of any comparator network on $\langle x_1, \ldots, x_n \rangle$.

34. [*23*] Is the following a sorting network?



35. [*20*] Prove that any *standard* sorting network must contain each of the "adjacent" comparators $[i:i+1]$, for $1 \leq i < n$, at least once.

▸ **36.** [*22*] The network of Fig. 47 involves only *adjacent* comparisons $[i:i+1]$; let us call such a network *primitive*. (a) Prove that a primitive sorting network for n elements must have at least $\binom{n}{2}$ comparators. [*Hint:* Consider the inversions of a permutation.] (b) (R. W. Floyd, 1964.) Let α be a primitive network for n elements, and let x be a vector such that $(x\alpha)_i > (x\alpha)_j$ for some $i < j$. Prove that $(y\alpha)_i > (y\alpha)_j$, where y is the vector $\langle n, n-1, \ldots, 1 \rangle$. (c) As a consequence of (b), a primitive network is a sorting network if and only if it sorts the single vector $\langle n, n-1, \ldots, 1 \rangle$!

37. [*M22*] The *odd-even transposition sort* for n numbers, $n \geq 3$, is a network n levels deep with $\frac{1}{2}n(n-1)$ comparators, arranged in a brick-like pattern as shown in Fig. 58. (When n is even, there are two possibilities.) Such a sort is especially easy to implement in hardware, since only two kinds of actions are performed alternately. Prove that such a network is, in fact, a valid sorting network. [*Hint:* See exercise 36.]

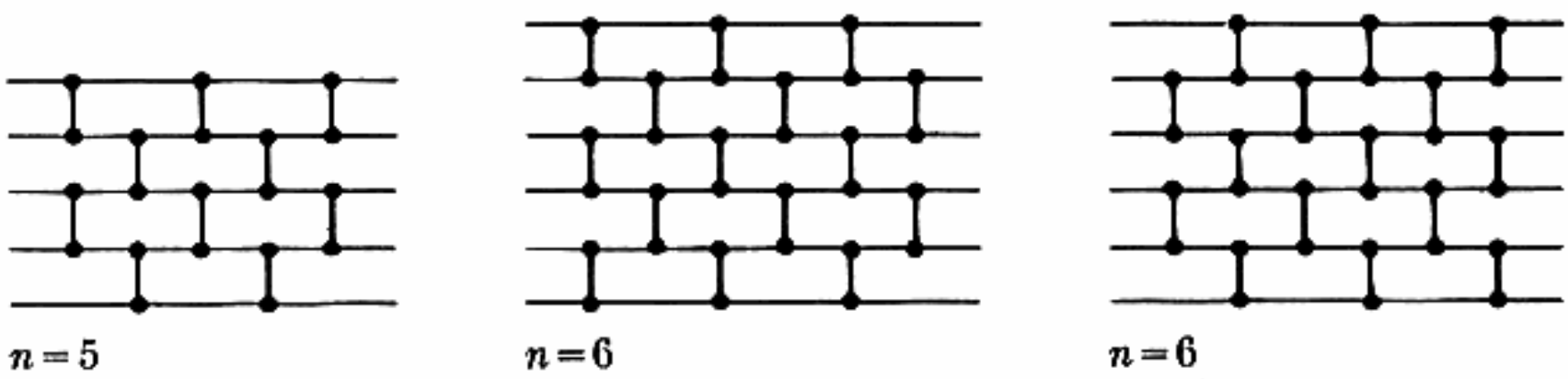


Fig. 58. The odd-even transposition sort.

38. [*29*] We may interpret sorting networks in another way, letting each line carry a multiset of m numbers, instead of a single number; under this interpretation, the operation $[i:j]$ replaces x_i and x_j, respectively, by $x_i \Cap x_j$ and $x_i \Cup x_j$, the least m and the greatest m of the $2m$ numbers $x_i \uplus x_j$. (See Fig. 59 for an illustration of this interpretation when $m = 2$.) If a and b are multisets of m numbers each, we say that $a \ll b$ iff $a \Cap b = a$ (equivalently, $a \Cup b = b$; the largest element of a is less than or equal to the smallest of b). Thus $a \Cap b \ll a \Cup b$.

Let α be an n-network, and let $x = \langle x_1, \ldots, x_n \rangle$ be a vector in which each x_i is a multiset of m elements. Prove that if $(x\alpha)_i$ is not $\ll (x\alpha)_j$ in the above interpretation, there is a vector y in D_n such that $(y\alpha)_i = 1$ and $(y\alpha)_j = 0$. [Consequently, a sorting

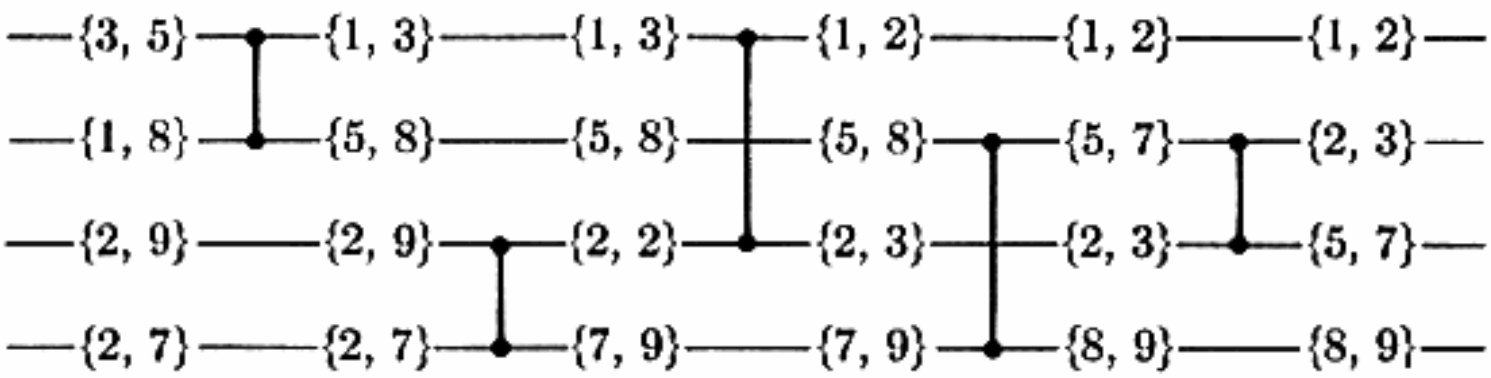


Fig. 59. Another interpretation of the sorting network in Fig. 44: each comparator module performs a merge operation.

network for n elements becomes a sorting network for mn elements if we replace comparisons by m-way merges. Figure 60 shows an 8-element sorter constructed from a 4-element sorter by using this observation.]

▸**39.** [*M23*] Show that, in the notation of exercise 38, $(x \curlywedge y) \curlywedge z = x \curlywedge (y \curlywedge z)$ and $(x \curlyvee y) \curlyvee z = x \curlyvee (y \curlyvee z)$; however $(x \curlyvee y) \curlywedge z$ is *not* always equal to $(x \curlywedge z) \curlyvee (y \curlywedge z)$, and $(x \curlywedge y) \curlyvee (x \curlywedge z) \curlyvee (y \curlywedge z)$ does *not* always equal the middle m elements of $x \uplus y \uplus z$. Find a correct formula, in terms of x, y, z and the $\curlywedge$ and $\curlyvee$ operations, for those middle elements.

▸**40.** [*M25*] (R. L. Graham.) The comparator $[i:j]$ is called *redundant* in the network $\alpha_1[i:j]\alpha_2$ if either $(x\alpha_1)_i \leq (x\alpha_1)_j$ for all vectors x, or $(x\alpha_1)_i \geq (x\alpha_1)_j$ for all vectors x. Prove that if α is a network with r irredundant comparators, there are at least r distinct ordered pairs (i, j) of distinct indices such that $(x\alpha)_i \leq (x\alpha)_j$ for all vectors x. (Consequently, a network with no redundant comparators contains at most $\binom{n}{2}$ modules.)

▸**41.** [*M27*] (V. E. Alekseyev.) Let $\alpha = [i_1:j_1] \ldots [i_r:j_r]$ be an n-network; for $1 \leq s \leq r$ we define $\alpha^s = [i'_1:j'_1] \ldots [i'_{s-1}:j'_{s-1}][i_s:j_s] \ldots [i_r:j_r]$, where the i'_k and j'_k are obtained from i_k and j_k by changing i_s to j_s and changing j_s to i_s wherever they appear. For example, if $\alpha = [1:2][3:4][1:3][2:4][2:3]$, then $\alpha^4 = [1:4][3:2][1:3][2:4][2:3]$.

a) Prove that $D_n\alpha = D_n(\alpha^s)$.

b) Prove that $(\alpha^s)^t = (\alpha^t)^s$.

c) A *conjugate* of α is any network of the form $(\ldots((\alpha^{s_1})^{s_2}) \ldots)^{s_k}$. Prove that α has at most 2^{r-1} conjugates.

d) Let $g_\alpha(x) = 1$ if $x \in D_n\alpha$, 0 if $x \notin D_n\alpha$; and let

$$f_\alpha(x) = (\bar{x}_{i_1} \vee x_{j_1}) \wedge \cdots \wedge (\bar{x}_{i_r} \vee x_{j_r}).$$

Prove that $g_\alpha(x) = \bigvee \{f_{\alpha'}(x) \mid \alpha' \text{ is a conjugate of } \alpha\}$.

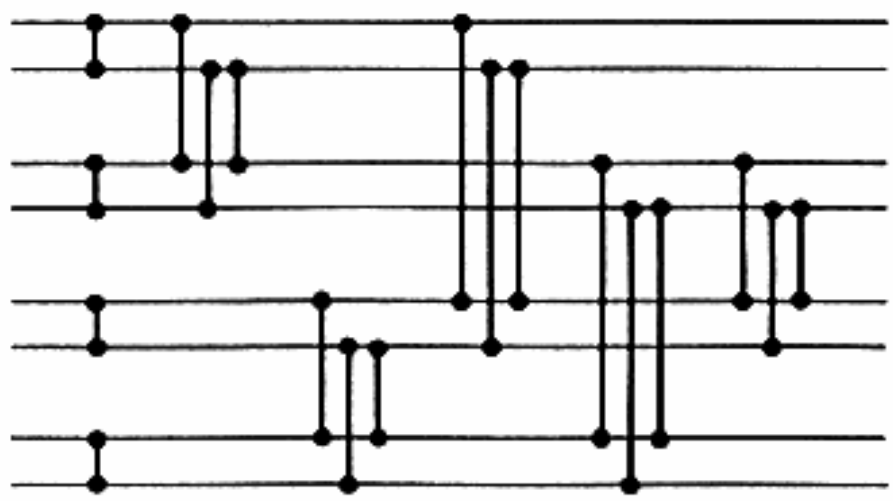


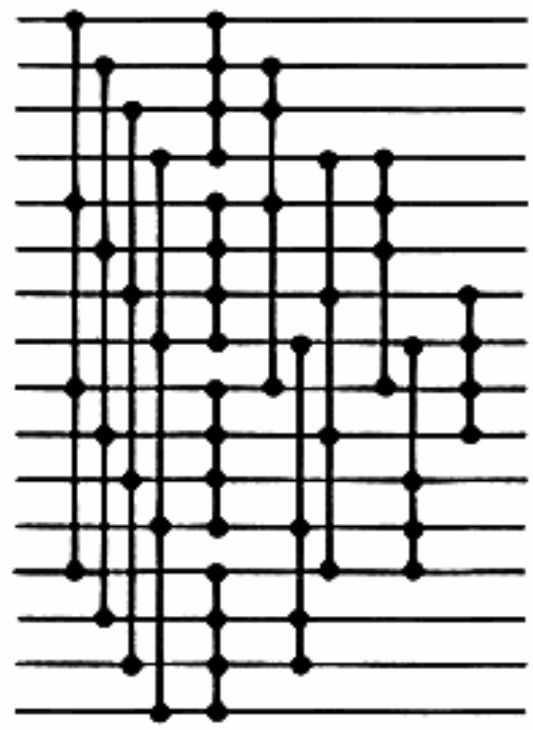
Fig. 60. An 8-sorter constructed from a 4-sorter, by using the merging interpretation.

c) Let G_α be the directed graph with vertices $\{1, \ldots, n\}$, and arcs $i_s \rightarrow j_s$ for $1 \leq s \leq r$. Prove that α is a sorting network if and only if $G_{\alpha'}$ has an oriented path from i to $i+1$ for $1 \leq i < n$ and for all α' conjugate to α. [This condition is somewhat remarkable, since G_α does not depend on the order of the comparators in α.]

▶ **42.** [*25*] (D. Van Voorhis.) Prove that $\hat{S}(n) \geq \hat{S}(n-1) + \lceil \log_2 n \rceil$.

43. [*23*] A *permutation network* is a sequence of modules $[i_1:j_1] \ldots [i_r:j_r]$ where each module $[i:j]$ can be set by external controls to pass its inputs unchanged or to switch x_i and x_j (irrespective of the values of x_i and x_j), and such that each permutation of the inputs is achievable on the output lines by some setting of the modules. Every sorting network is clearly a permutation network, but the converse is not true: Find a permutation network for five elements which has only eight modules.

44. [*46*] Study the properties of sorting networks which are made from m-sorter modules instead of 2-sorters. (For example, G. Shapiro has constructed the network



which sorts 16 elements using fourteen 4-sorters. Is this best possible? Are there efficient ways to sort m^2 elements with m-sorter modules, for all m?)

45. [*48*] Find an (m, n) merging network with less than $C(m, n)$ comparators, or prove that none exists.

46. [*48*] Find an (m, n) merging network with less than $\lceil \log_2 (m+n) \rceil$ levels of delay, or prove that none exists.

47. [*48*] Study the class of sorting schemes that can be implemented with perfect shuffle networks as in Fig. 57, with other patterns of "0", "+", and "−" operations.

48. [*HM49*] Explore the properties of the $\hat{\wedge}$ and $\hat{\vee}$ operations defined in exercise 38. Is it possible to characterize all of the identities in this algebra in some nice way, or to derive them all from a finite set of identities? In this regard, identities such as $x \mathbin{\hat{\wedge}} x \mathbin{\hat{\wedge}} x = x \mathbin{\hat{\wedge}} x$, or $x \mathbin{\hat{\wedge}} (x \mathbin{\hat{\vee}} (x \mathbin{\hat{\wedge}} (x \mathbin{\hat{\vee}} y))) = x \mathbin{\hat{\wedge}} (x \mathbin{\hat{\vee}} y)$, which hold only for $m \leq 2$, are of comparatively little interest; consider only the identities which are true for *all* m.

49. [*M49*] What is the asymptotic behavior of the function $T(n)$ defined in exercise 6? Is $T(n) < \hat{T}(n)$ for some n?

50. [*50*] Find the exact value of $\hat{S}(n)$ for some $n > 8$.

51. [*M50*] Prove that the asymptotic value of $\hat{S}(n)$ is not $O(n \log n)$.

EXERCISES—Second Set

The following exercises deal with several different types of optimality questions related to sorting. The first few problems are based on an interesting "multi-head" generalization of the bubble sort, investigated by P. N. Armstrong and R. J. Nelson as early as 1954. [See *U.S. Patents 3029413, 3034102.*] Let $1 = h_1 < h_2 < \cdots < h_m = n$ be an increasing sequence of integers; we shall call it a "head sequence" of length m and span n, and we shall use it to define a special kind of sorting method. The sorting of records $R_1 \ldots R_N$ proceeds in several passes, and each pass consists of $N + n - 1$ steps. On step j, for $j = 1 - n, 2 - n, \ldots, N - 1$, the records $R_{j+h[1]}, R_{j+h[2]}, \ldots, R_{j+h[m]}$ are examined and rearranged if necessary so that their keys are in order. (We say that $R_{j+h[1]}, \ldots, R_{j+h[m]}$ are "under the read-write heads." When $j + h[k]$ is < 1 or $> N$, record $R_{j+h[k]}$ is left out of consideration; in effect, the keys $K_0, K_{-1}, K_{-2}, \ldots$ are treated as $-\infty$ and $K_{N+1}, K_{N+2}, \ldots$ are treated as $+\infty$. Therefore step j is actually trivial when $j \le -h[m-1]$ or $j > N - h[2]$.)

For example, the following table shows one pass of a sort when $m = 3$, $N = 9$, and $h_1 = 1$, $h_2 = 2$, $h_3 = 4$:

	K_{-2}	K_{-1}	K_0	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9	K_{10}	K_{11}	K_{12}
$j = -3$	—	—		3	1	4	5	9	2	6	8	7			
$j = -2$		—	—	3	1	4	5	9	2	6	8	7			
$j = -1$			—	3	1	4	5	9	2	6	8	7			
$j = 0$				1	3	4	5	9	2	6	8	7			
$j = 1$				1	3	4	5	9	2	6	8	7			
$j = 2$				1	3	2	4	9	5	6	8	7			
$j = 3$				1	3	2	4	6	5	9	8	7			
$j = 4$				1	3	2	4	5	6	9	8	7			
$j = 5$				1	3	2	4	5	6	7	8	9			
$j = 6$				1	3	2	4	5	6	7	8	9	—		
$j = 7$				1	3	2	4	5	6	7	8	9		—	
$j = 8$				1	3	2	4	5	6	7	8	9	—		—

Note that when $m = 2$, $h_1 = 1$, and $h_2 = 2$, this "multi-head" method reduces to the bubble sort (Algorithm 5.2.2B).

52. [*21*] (James Dugundji.) Prove that if $h[k+1] = h[k] + 1$ for some k, $1 \le k < m$, the multihead sorter defined above will eventually sort any input file in a finite number of passes. But if $h[k+1] \ge h[k] + 2$ for $1 \le k < m$, the input might *never* become sorted.

▸**53.** [*30*] (Armstrong and Nelson.) Given that $h[k+1] \le h[k] + k$ for $1 \le k < m$, and $N \ge n - 1$, prove that the largest $n - 1$ elements always move to their final destination on the first pass. [*Hint:* Use the zero-one principle; when sorting 0's and 1's, with less than n 1's, prove that it is impossible to have all heads sensing a 1 unless all 0's lie to the left of the heads.]

Prove that sorting will be complete in at most $\lceil (N-1)/(n-1) \rceil$ passes when the heads satisfy the given conditions. Is there an input file which requires this many passes?

54. [*26*] If $n = N$, prove that the first pass can be guaranteed to place the smallest key into position R_1 if and only if $h[k+1] \leq 2h[k]$ for $1 \leq k < m$.

55. [*34*] (J. Hopcroft.) A "perfect sorter" for N elements is a multi-head sorter with $N = n$ which always finishes in one pass. Exercise 53 proves that the sequence $\langle h_1, h_2, h_3, h_4, \ldots, h_m\rangle = \langle 1, 2, 4, 7, \ldots, 1 + \binom{m}{2}\rangle$ gives a perfect sorter for $N = \binom{m}{2} + 1$ elements, using $m = (\sqrt{8N-7}+1)/2$ heads. For example, the head sequence $\langle 1, 2, 4, 7, 11, 16, 22\rangle$ is a perfect sorter for 22 elements.

Prove that, in fact, the head sequence $\langle 1, 2, 4, 7, 11, 16, 23\rangle$ is a perfect sorter for 23 elements.

56. [*49*] Study the largest N for which m-head perfect sorters exist, given m. Is $N = O(m^2)$?

57. [*23*] (V. Pratt.) When each head h_k is in position 2^{k-1} for $1 \leq k \leq m$, how many passes are necessary to sort the sequence $z_1\, z_2 \ldots z_{2^m-1}$ of 0's and 1's where $z_j = 0$ iff j is a power of 2?

58. [*24*] (*Uniform sorting.*) The tree of Fig. 34 in Section 5.3.1 makes the comparison 2:3 in both branches on level 1, and on level 2 it compares 1:3 in each branch unless that comparison would be redundant. In general, we can consider the class of all sorting algorithms which are uniform in that way; assuming that the $M = \binom{N}{2}$ pairs $\{(a, b) \mid 1 \leq a < b \leq N\}$ have been arranged into a sequence

$$(a_1, b_1)\ (a_2, b_2), \ldots, (a_M, b_M),$$

we can successively make each of the comparisons $K_{a_1}{:}K_{b_1}$, $K_{a_2}{:}K_{b_2}$, ... whose outcome is not already known. Each of the $M!$ arrangements of the (a, b) pairs defines a uniform sorting algorithm. The concept of uniform sorting is due to H. L. Beus [*JACM* **17** (1970), 482–495], whose work has suggested the next few exercises.

It is convenient to define uniform sorting formally by means of graph theory. Let G be the directed graph on the vertices $\{1, 2, \ldots, N\}$, having no arcs. For $i = 1, 2, \ldots, M$ we add arcs to G as follows:

Case 1. G contains a path from a_i to b_i. Add the arc $a_i \rightarrow b_i$ to G.

Case 2. G contains a path from b_i to a_i. Add the arc $b_i \rightarrow a_i$ to G.

Case 3. G contains no path from a_i to b_i or b_i to a_i. Compare $K_{a_i}{:}K_{b_i}$; then add the arc $a_i \rightarrow b_i$ to G if $K_{a_i} \leq K_{b_i}$, the arc $b_i \rightarrow a_i$ if $K_{a_i} > K_{b_i}$.

We are concerned primarily with the number of key comparisons made by a uniform sorting algorithm, not with the mechanism by which redundant comparisons are actually avoided; the graph G need not be constructed explicitly, it is used here merely to help define the concept of uniform sorting.

We shall also consider *restricted uniform sorting*, in which only paths of length 2 are counted in cases 1, 2, and 3 above. (A restricted uniform sorting algorithm may make some redundant comparisons, but exercise 59 shows that the analysis is somewhat simpler in the restricted case.)

Prove that the restricted uniform algorithm is the same as the uniform algorithm when the sequence of pairs is taken in lexicographic order

$$(1, 2)(1, 3)(1, 4) \ldots (1, N)(2, 3)(2, 4) \ldots (2, N) \ldots (N-1, N).$$

Show in fact that both algorithms are equivalent to "quicksort" (Algorithm 5.2.2Q) when the keys are distinct and when quicksort's redundant comparisons are removed as in exercise 5.2.2–24. (Disregard the order in which the comparisons are actually made in quicksort, consider only which pairs of keys are compared.)

59. [*M38*] Given a pair sequence $(a_1, b_1) \ldots (a_M, b_M)$ as in exercise 58, let c_i be the number of pairs (j, k) such that $j < k < i$ and (a_i, b_i), (a_j, b_j), (a_k, b_k) forms a triangle. (a) Prove that the average number of comparisons made by the restricted uniform sorting algorithm is $\sum_{1 \le i \le M} 2/(c_i + 2)$. (b) Use the results of (a) and exercise 58 to determine the average number of irredundant comparisons performed by quicksort. (c) The following pair sequence is inspired by (but not equivalent to) merge sorting:

$$(1, 2)(3, 4)(5, 6) \ldots (1, 3)(1, 4)(2, 3)(2, 4)(5, 7) \ldots (1, 5)(1, 6)(1, 7)(1, 8)(2, 5) \ldots$$

Does the uniform method based on this sequence do more or less comparisons than quicksort, on the average?

60. [*M29*] In the worst case, quicksort does $\binom{N}{2}$ comparisons. Do all restricted uniform sorting algorithms (in the sense of exercise 57) perform $\binom{N}{2}$ comparisons in their worst case?

61. [*M48*] (H. L. Beus.) Does quicksort have the minimum average number of comparisons, over all (restricted) uniform sorting algorithms?

62. [*25*] The Ph.D. thesis "Electronic Data Sorting" by Howard B. Demuth (Stanford University: October, 1956) was perhaps the first paper to deal in any detail with questions of computational complexity. Demuth considered several abstract models for sorting devices, and established lower and upper bounds on the mean and maximum execution times achievable with each model. His simplest model, the "circular non-reversible memory" (Fig. 61), is the subject of this exercise.

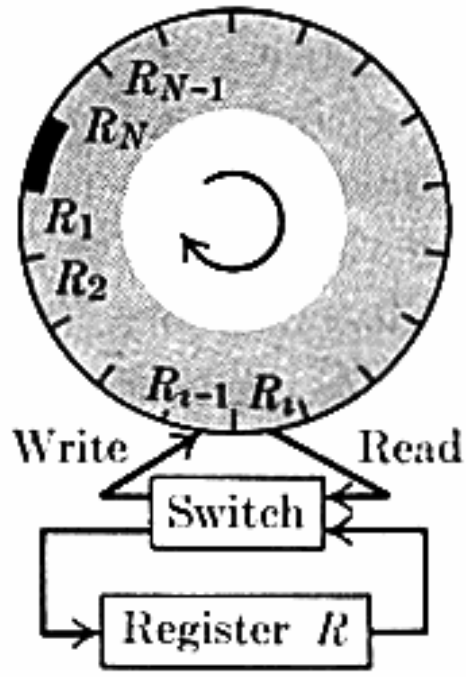


Fig. 61. A device for which the bubble-sort strategy is optimum.

Consider a machine which sorts $R_1\ R_2 \ldots R_N$ in a number of passes, where each pass contains the following $N + 1$ steps:

Step 1. Set $R \leftarrow R_1$. (R is an internal machine register.)

Step i, for $1 < i \le N$.

Either (a) Set $R_{i-1} \leftarrow R$, $R \leftarrow R_i$;
or (b) Set $R_{i-1} \leftarrow R_i$, leaving R unchanged.

Step $N + 1$. Set $R_N \leftarrow R$.

The problem is to find a way to choose between alternatives (a) and (b) each time, in order to minimize the number of passes required to sort.

Prove that the "bubble sort" technique is optimum for this model. In other words, show that the strategy which selects alternative (a) whenever $R \le R_i$ and alternative (b) whenever $R > R_i$ will achieve the minimum number of passes.

5.4. EXTERNAL SORTING

Now it is time for us to study the interesting problems which arise when the number of records to be sorted is larger than our computer can hold in its high-speed internal memory. External sorting is quite different from internal sorting, even though the problem in both cases is to sort a given file into nondecreasing order, since efficient storage accessing on external files is rather severely limited. The data structures must be arranged so that comparatively slow peripheral memory devices (tapes, disks, drums, etc.) can quickly cope with the requirements of the sorting algorithm. Consequently most of the internal sorting techniques we have studied (insertion, exchange, selection) are virtually useless for external sorting, and it is necessary to reconsider the whole question.

Suppose, for example, that we have a file of 5000 records $R_1 R_2 \ldots R_{5000}$ to sort, and that each record R_i is 20 words long (although the keys K_i are not necessarily this long). If only 1000 of these records will fit in the internal memory of a particular computer at one time, what shall we do?

One fairly obvious solution is to start by sorting each of the five subfiles $R_1 \ldots R_{1000}$, $R_{1001} \ldots R_{2000}$, $\ldots$, $R_{4001} \ldots R_{5000}$, independently, then to merge the resulting subfiles together. Fortunately the process of merging uses only very simple data structures, namely linear lists which are traversed in a sequential manner as stacks or as queues; hence merging can be done without difficulty on the least expensive external memory devices.

The process just described—internal sorting followed by "external merging"—is very commonly used, and we shall devote most of our study of external sorting to variations on this theme.

The ascending sequences of records, which are produced by the initial internal sorting phase, are often called *strings* in the published literature about sorting; this terminology is fairly widespread, but it unfortunately conflicts with even more widespread usage in other branches of computer science, where "strings" are *arbitrary* sequences of symbols. Our study of permutations has already given us a perfectly good name for the sorted segments of a file, which are conventionally called ascending runs or simply *runs*. Therefore we shall consistently use the word "runs" to describe sorted portions of a file. In this way it is possible to distinguish between "strings of runs" and "runs of strings" without ambiguity. (Of course, "runs of a program" means something else again; we can't have everything.)

Let us first consider the process of external sorting when *magnetic tapes* are used for auxiliary storage. Perhaps the simplest and most appealing way to merge with tapes is the *balanced 2-way merge*, following the central idea which was used in Algorithms 5.2.4N, S, and L. We use four "working tapes" in this process. During the first phase, ascending runs produced by internal sorting are placed alternately on Tapes 1 and 2, until the input is exhausted. Then Tapes 1 and 2 are rewound to their beginnings, and we merge the runs from these tapes, obtaining new runs which are twice as long as the original ones; the new runs are written alternately on Tapes 3 and 4 as they are being formed.

(If Tape 1 contains one more run than Tape 2, an extra "dummy" run of length 0 is assumed to be present on Tape 2.) Then all tapes are rewound, and the contents of Tapes 3 and 4 are merged into quadruple-length runs recorded alternately on Tapes 1 and 2. The process continues, doubling the length of runs each time, until only one run is left (namely the entire sorted file). If S runs were produced during the internal sorting phase, and if $2^{k-1} < S \le 2^k$, this balanced 2-way merge procedure makes exactly $k = \lceil \log_2 S \rceil$ merging passes over all the data.

For example, in the above situation where 5000 records are to be sorted with an internal memory capacity of 1000, we have $S = 5$. The initial distribution phase of the sorting process places five runs on tape as follows:

$$\begin{array}{ll} \text{Tape 1} & R_1 \ldots R_{1000};\ R_{2001} \ldots R_{3000};\ R_{4001} \ldots R_{5000}. \\ \text{Tape 2} & R_{1001} \ldots R_{2000};\ R_{3001} \ldots R_{4000}. \\ \text{Tape 3} & \text{(empty)} \\ \text{Tape 4} & \text{(empty)} \end{array} \tag{1}$$

The first pass of merging then produces longer runs on Tapes 3 and 4, as it reads Tapes 1 and 2, as follows:

$$\begin{array}{ll} \text{Tape 3} & R_1 \ldots R_{2000};\ R_{4001} \ldots R_{5000}. \\ \text{Tape 4} & R_{2001} \ldots R_{4000}. \end{array} \tag{2}$$

(A dummy run has implicitly been added at the end of Tape 2, so that the last run $R_{4001} \ldots R_{5000}$ on Tape 1 is merely copied onto Tape 3.) After all tapes are rewound, the next pass over the data produces

$$\begin{array}{ll} \text{Tape 1} & R_1 \ldots R_{4000}. \\ \text{Tape 2} & R_{4001} \ldots R_{5000}. \end{array} \tag{3}$$

(Again that run $R_{4001} \ldots R_{5000}$ was simply copied; but if we had started with 8000 records, Tape 2 would have contained $R_{4001} \ldots R_{8000}$ at this point.) Finally, after another spell of rewinding, $R_1 \ldots R_{5000}$ is produced on Tape 3, and the sorting is complete.

Balanced merging can easily be generalized to the case of T tapes, for any $T \ge 3$. Choose any number P with $1 \le P < T$, and divide the T tapes into two "banks," with P tapes on the left bank and $T - P$ on the right. Distribute the initial runs as evenly as possible onto the P tapes in the left bank; then do a P-way merge from the left to the right, followed by a $(T - P)$-way merge from the right to the left, etc., until sorting is complete. The best choice of P usually turns out to be $\lceil T/2 \rceil$ (see exercises 3, 4).

Balanced two-way merging is the special case $T = 4$, $P = 2$. Let us reconsider the example above using more tapes, taking $T = 6$ and $P = 3$.

The initial distribution now gives us

$$\begin{array}{ll} \text{Tape 1} & R_1 \ldots R_{1000};\ R_{3001} \ldots R_{4000}. \\ \text{Tape 2} & R_{1001} \ldots R_{2000};\ R_{4001} \ldots R_{5000}. \\ \text{Tape 3} & R_{2001} \ldots R_{3000}. \end{array} \tag{4}$$

And the first merging pass produces

$$\begin{array}{ll} \text{Tape 4} & R_1 \ldots R_{3000}. \\ \text{Tape 5} & R_{3001} \ldots R_{5000}. \\ \text{Tape 6} & \text{(Empty)} \end{array} \tag{5}$$

(A dummy run has been assumed on Tape 3.) The second merging pass completes the job, placing $R_1 \ldots R_{5000}$ on Tape 1. In this special case $T = 6$ is essentially the same as $T = 5$, since the sixth tape is used only when $S \geq 7$.

Three-way merging actually requires somewhat more computer processing time than two-way merging, but this is generally negligible compared to the time needed to read, write, and rewind the tapes; we can get a fairly good estimate of the running time by considering only the amount of tape motion. The example in (4) and (5) required only two passes over the data, compared to three passes when $T = 4$, so the merging takes only about two-thirds as long when $T = 6$.

Balanced merging is quite simple, but if we look more closely, we find immediately that it isn't the *best* way to handle the particular cases treated above! Instead of going from (1) to (2) and rewinding all of the tapes, we should have stopped the first merging pass after Tapes 3 and 4 contained $R_1 \ldots R_{2000}$ and $R_{2001} \ldots R_{4000}$, respectively, with Tape 1 poised ready to read $R_{4001} \ldots R_{5000}$. Then Tapes 2, 3, 4 could be rewound and we could complete the sort by doing a three-way merge onto Tape 2. The total number of records read from tape during this procedure would be only $4000 + 5000 = 9000$, compared to $5000 + 5000 + 5000 = 15{,}000$ in the balanced scheme. A smart computer would be able to figure this out!

Indeed, when we have five runs and four tapes we can do even better by distributing them as follows:

$$\begin{array}{ll} \text{Tape 1} & R_1 \ldots R_{1000};\ R_{3001} \ldots R_{4000}. \\ \text{Tape 2} & R_{1001} \ldots R_{2000};\ R_{4001} \ldots R_{5000}. \\ \text{Tape 3} & R_{2001} \ldots R_{3000}. \\ \text{Tape 4} & \text{(Empty)} \end{array}$$

Then a three-way merge to Tape 4, followed by a rewind of Tapes 3 and 4, followed by a three-way merge to Tape 3, would complete the sort with only $3000 + 5000 = 8000$ records read.

And, of course, if we had six tapes we could put the initial runs on Tapes 1 through 5 and complete the sort in one pass by doing a five-way merge to

Tape 6. These considerations indicate that simple balanced merging isn't the best, and it is interesting to look for improved merging patterns.

Subsequent portions of this chapter investigate external sorting more deeply. In Section 5.4.1, we consider the "internal sorting" phase which produces the initial runs; of particular interest is the technique of "replacement selection," which takes advantage of the order present in most data to produce long initial runs that actually exceed the internal memory capacity by a significant amount. Section 5.4.1 also discusses a suitable data structure for multiway merging.

The most important merging patterns are discussed in Sections 5.4.2 through 5.4.5. It is convenient to have a rather naive conception of tape sorting as we learn the characteristics of these patterns, before we come to grips with the harsh realities of real tape drives and real data that is to be sorted. For example, we may blithely assume (as we did above) that the original input records appear magically during the initial distribution phase; in fact, these input records probably will occupy one of our tapes, and they may even fill several tape reels since tapes aren't of infinite length! It is best to ignore such mundane considerations until after an academic understanding of the classical merging patterns has been gained. Then Section 5.4.6 brings the discussion down to earth by discussing real-life constraints which strongly influence the choice of a merging pattern. Section 5.4.6 compares the basic merging patterns of Sections 5.4.2 through 5.4.5, using a variety of assumptions which arise in practice.

Some other approaches to external sorting, *not* based on merging, are discussed in Sections 5.4.7 and 5.4.8. Finally Section 5.4.9 completes our survey of external sorting by treating the important problem of sorting on bulk memories such as disks and drums.

EXERCISES

1. [*15*] The text suggests internal sorting first, followed by external merging. Why don't we do away with the internal sorting phase, simply merging the records into longer and longer runs right from the start?

2. [*10*] What will the sequence of tape contents be, analogous to (1) through (3), when the example records $R_1\ R_2 \ldots R_{5000}$ are sorted using a 3-tape balanced method with $P = 2$? Compare this to the 4-tape merge; how many passes are made over all the data, after the initial distribution of runs?

3. [*20*] Show that the balanced $(P, T - P)$-way merge applied to S initial runs takes $2k$ passes, when $P^k(T - P)^{k-1} < S \leq P^k(T - P)^k$; and it takes $2k + 1$ passes, when $P^k(T - P)^k < S \leq P^{k+1}(T - P)^k$.

Give simple formulas for (a) the exact number of passes, as a function of S, when $T = 2P$; and (b) the approximate number of passes, as $S \to \infty$, for general P and T.

4. [*HM15*] What value of P, for $1 \leq P < T$, makes $P(T - P)$ a maximum?

In Section 5.2.4, we studied internal sorting methods based on two-way merging, the process of combining two ordered sequences into a single ordered sequence. It is not difficult to extend this to the notion of P-way merging, where P input runs are combined into a single output run.

Let's assume that we have been given P ascending runs, i.e., sequences of records whose keys are in nondecreasing order. The obvious way to merge them is to look at the first record of each run and to select the record whose key is smallest; this record is transferred to the output and removed from the input, and the process is repeated. At any given time we need to look at only P keys (one from each input run) and select the smallest. If two or more keys are smallest, an arbitrary one is selected.

When P isn't too large, it is convenient to make this selection by simply doing $P - 1$ comparisons to find the smallest of the current keys. But when P is, say, 8 or more, we can save work by using a *selection tree* as described in Section 5.2.3; then only about $\log_2 P$ comparisons are needed each time, once the tree has been set up.

Consider, for example, the case of four-way merging, with a two-level selection tree:

Step 1.
$$087\begin{cases}087\begin{cases}087 & 503 & \infty\\ 170 & 908 & \infty\end{cases}\\ 154\begin{cases}154 & 426 & 653 & \infty\\ 612 & \infty\end{cases}\end{cases}$$

Step 2.
$$087\ \ 154\begin{cases}170\begin{cases}503 & \infty\\ 170 & 908 & \infty\end{cases}\\ 154\begin{cases}154 & 426 & 653 & \infty\\ 612 & \infty\end{cases}\end{cases}$$

Step 3.
$$087\ \ 154\ \ 170\begin{cases}170\begin{cases}503 & \infty\\ 170 & 908 & \infty\end{cases}\\ 426\begin{cases}426 & 653 & \infty\\ 612 & \infty\end{cases}\end{cases}$$

$\vdots$

Step 9.
$$087\ \ 154\ \ 170\ \ 426\ \ 503\ \ 612\ \ 653\ \ 908\ \ \infty\ \begin{cases}\infty\ \begin{cases}\infty\\ \infty\end{cases}\\ \infty\ \begin{cases}\infty\\ \infty\end{cases}\end{cases}$$

An additional key "∞" has been placed at the end of each run in this example, so that the merging terminates gracefully. Since external merging generally deals with very long runs, the addition of records with ∞ keys does not add

substantially to the length of the data or to the amount of work involved in merging, and such "sentinel" records frequently serve as a useful way to delimit the runs on a file.

Each step after the first in the above process consists of replacing the smallest element by the succeeding element in its run, and changing the corresponding path in the selection tree. Thus the three positions of the tree which contain 087 in Step 1 are changed in Step 2; the three positions containing 154 in Step 2 are changed in Step 3; and so on. The process of replacing one key by another in the selection tree is called *replacement selection*.

We can look at this four-way merge in several ways. From one standpoint it is equivalent to three two-way merges performed concurrently as coroutines; each node in the selection tree represents one of the sequences involved in concurrent merging processes. The selection tree is also essentially operating as a priority queue, with a "smallest in, first out" discipline.

As in Section 5.2.3 we could use a heap instead of a selection tree, to implement the priority queue. (The heap would, of course, be arranged so that the *smallest* element appears at the top, instead of the largest, reversing the order of Eq. 5.2.3–3.) Since a heap does not have a fixed size, we could therefore avoid the use of "∞" keys; merging would be complete when the heap becomes empty. On the other hand, external sorting applications usually deal with comparatively long records and keys, so that the heap is filled with pointers to keys instead of the keys themselves; we shall see below that selection trees can be represented by pointers in such a convenient manner that they are probably superior to heaps in this situation.

A tree of "losers." Figure 62 shows the complete binary tree with 12 external (square) nodes and 11 internal (circular) nodes; the external nodes have been

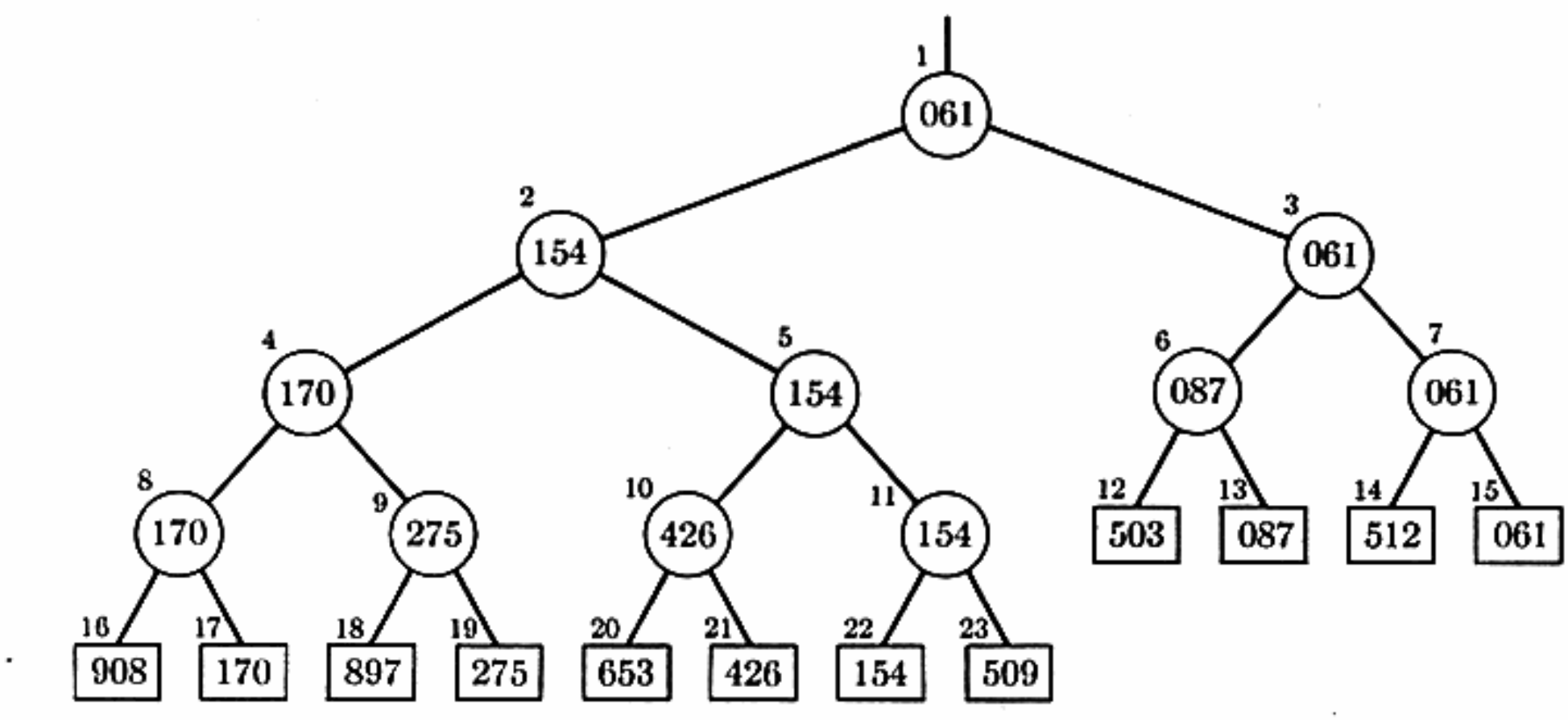


Fig. 62. A tournament to select the smallest key, using a complete binary tree whose nodes are numbered from 1 to 23.

filled with keys, and the internal nodes have been filled with the "winners" if the tree is regarded as a tournament to select the smallest key. The smaller numbers above each node show the traditional way to allocate consecutive storage positions for complete binary trees.

When the smallest key, 061, is to be replaced by another key in the selection tree of Fig. 62, we will have to look at the keys 512, 087, and 154, and no other existing keys, in order to determine the new state of the selection tree. Considering the tree as a tournament, these three keys are the losers in the matches played by 061. This suggests that what we really ought to store in the internal nodes of the tree is the *loser* of each match, instead of the winner; then the information required for updating the tree is readily available.

Figure 63 shows the same tree as Fig. 62, but with the losers represented instead of the winners. An extra node number 0 has been appended at the top of the tree, to indicate the champion of the tournament. Note that each key except the champion is a loser exactly once (cf. Section 5.3.3), so each key appears once in an external node and once in an internal node.

In practice, the external nodes at the bottom of Fig. 63 will represent fairly long records stored in computer memory, and the internal nodes will represent pointers to those records. Note that P-way merging calls for exactly P external nodes and P internal nodes, each in adjacent groups, so that several efficient methods of storage allocation suggest themselves. It is not difficult to see how to use a "loser-oriented" tree for replacement selection; we shall discuss the algorithm in some detail later in this section.

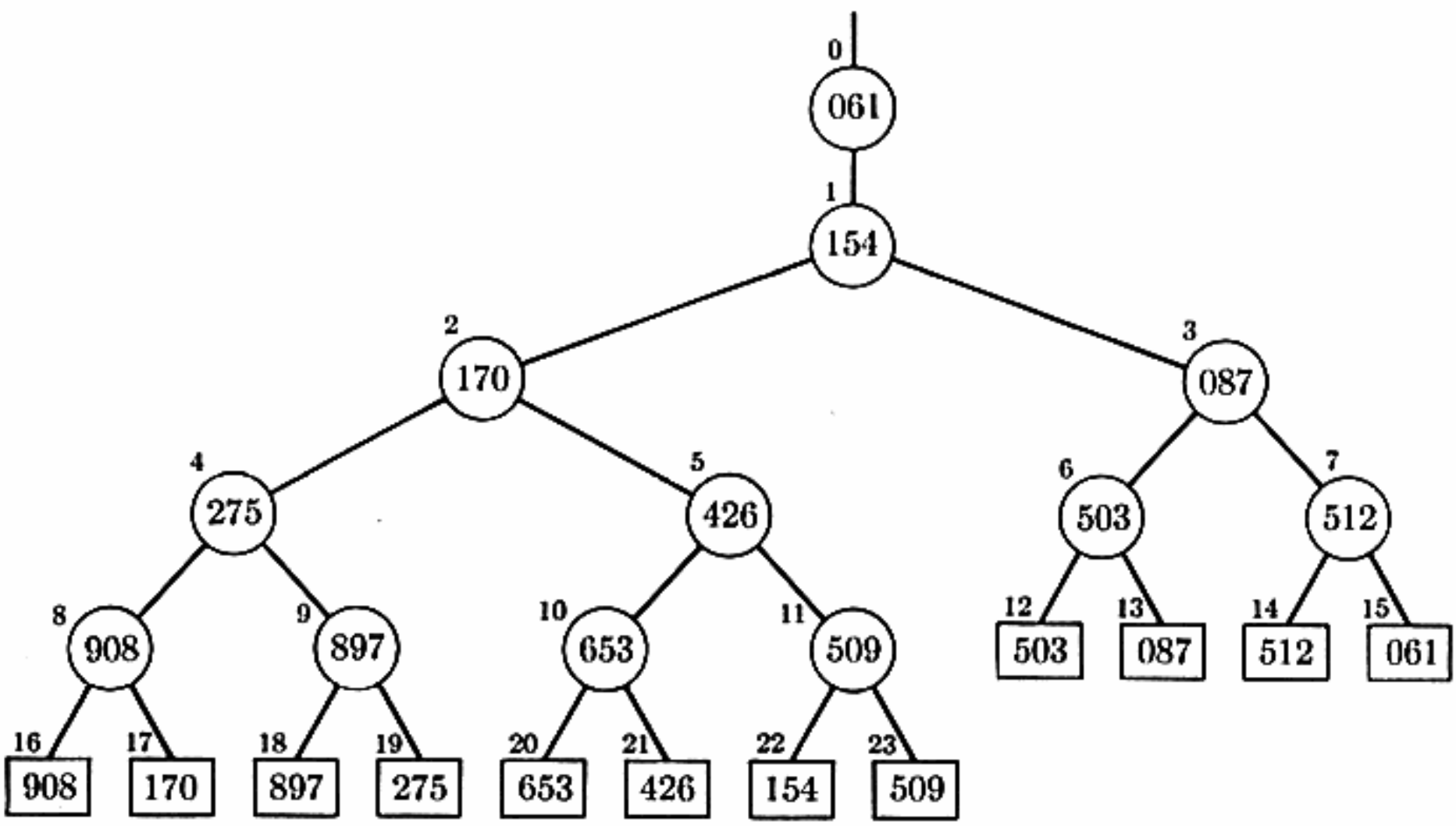


Fig. 63. The same tournament as Fig. 62, but showing the losers instead of the winners; the champion appears at the very top.

Initial runs by replacement selection. The technique of replacement selection can be used also in the first phase of external sorting, if we essentially do a P-way merge of the input data with itself! In this case we take P to be fairly large, so that the internal memory is essentially filled. When a record is output, it is replaced by the next record from the input. If the new record has a smaller key than the one just output, we cannot include it in the current run; but otherwise we can enter it into the selection tree in the usual way and it will form part of the run currently being produced. Thus the runs can contain more than P records each, even though we never have more than P in the selection tree at any time. Table 1 illustrates this process for $P = 4$; parenthesized numbers are waiting for inclusion in the following run.

Table 1

EXAMPLE OF FOUR-WAY REPLACEMENT SELECTION

Memory contents				Output
503	087	512	061	061
503	087	512	908	087
503	170	512	908	170
503	897	512	908	503
(275)	897	512	908	512
(275)	897	653	908	653
(275)	897	(426)	908	897
(275)	(154)	(426)	908	908
(275)	(154)	(426)	(509)	(end run)
275	154	426	509	154
275	612	426	509	275
		etc.		

This important method of forming initial runs was first described by Harold H. Seward [Master's Thesis, Digital Computer Laboratory Report R-232 (Mass. Inst. of Technology, 1954), 29–30], who gave reason to believe that the runs would contain more than $1.5P$ records when applied to random data. A. I. Dumey had also suggested the idea about 1950 in connection with a special sorting device planned by Engineering Research Associates, but he did not publish it. The name "replacement selecting" was coined by E. H. Friend [*JACM* **3** (1956), 154], who remarked that "the expected length of the sequences produced eludes formulation but experiment suggests that $2P$ is a reasonable expectation."

A clever way to show that $2P$ is indeed the expected run length was discovered by E. F. Moore, who compared the situation to a snowplow on a circular track [*U.S. Patent 2983904* (1961), cols. 3–4]. Consider the situation shown in Fig. 64; flakes of snow are falling uniformly on a circular road, and a lone snowplow is continually clearing the snow. Once the snow has been plowed

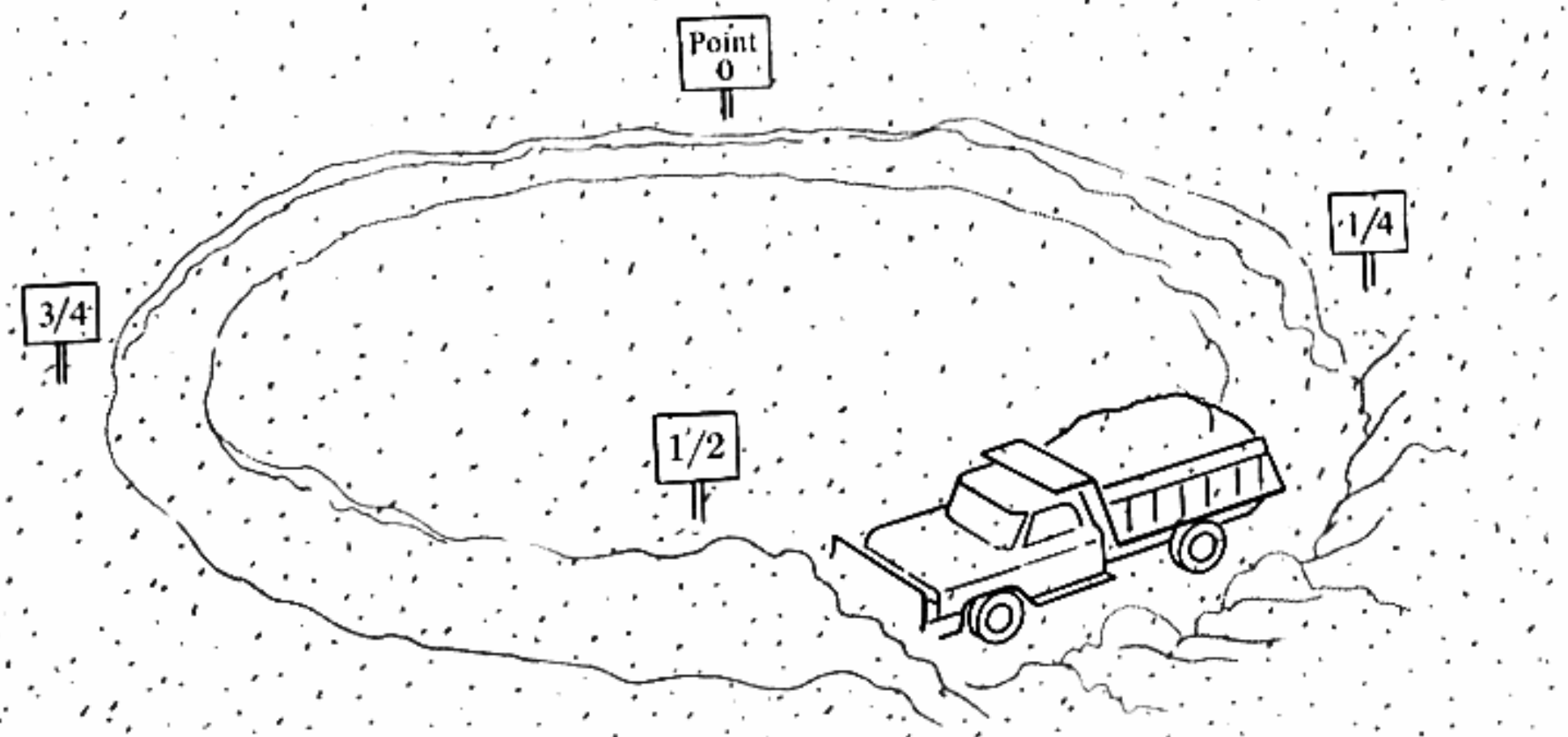


Fig. 64. The perpetual plow on its ceaseless cycle.

off the road, it disappears from the system. Points on the road may be designated by real numbers x, $0 \leq x < 1$; a flake of snow falling at position x represents an input record whose key is x, and the snowplow represents the output of replacement selection. The ground speed of the snowplow is inversely proportional to the height of snow it encounters, and the situation is perfectly balanced so that the total amount of snow on the road at all times is exactly P. A new run is formed in the output whenever the plow passes point 0.

After this system has been in operation for awhile, it is intuitively clear that it will approach a stable situation in which the snowplow runs at constant speed (because of the circular symmetry of the track). This means that the snow is at constant height when it meets the plow, and the height drops off linearly in front of the plow as shown in Fig. 65. It follows that the volume of snow removed in one revolution (namely the run length) is twice the amount present at any one time (namely P).

In many commercial applications the input data is *not* completely random, it already has a certain amount of existing order; therefore the runs produced

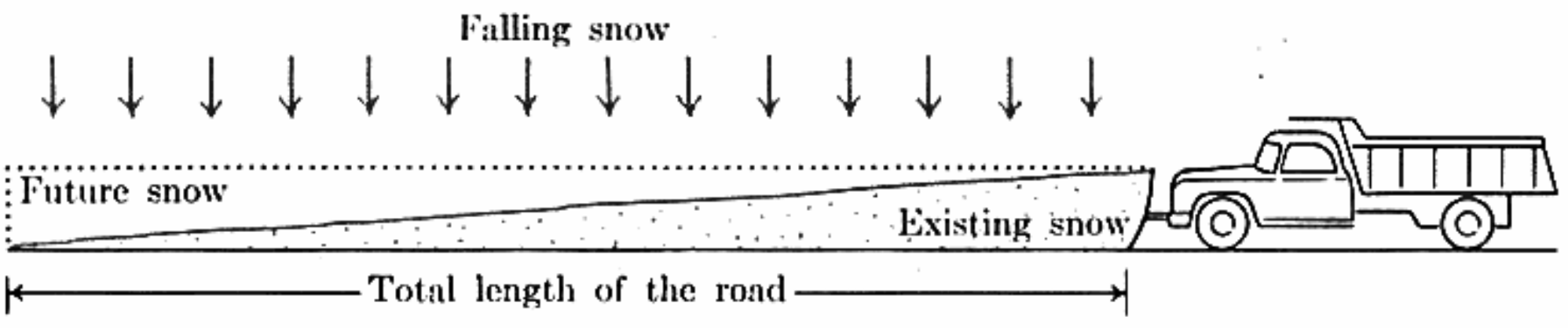


Fig. 65. Cross-section, showing the varying height of snow in front of the plow when the system is in its "steady state."

by replacement selection will tend to contain even more than $2P$ records. We shall see that the time required for external merge sorting is largely governed by the number of runs produced by the initial distribution phase, so that replacement selection becomes especially desirable; other types of internal sorting would produce about twice as many initial runs because of the limitations on memory size.

Let us now consider the process of creating initial runs by replacement selection in detail. The following algorithm is due to John R. Walters, James Painter, and Martin Zalk, who used it in a merge-sort program for the Philco 2000 in 1958. It incorporates a rather nice way to initialize the selection tree and to distinguish records belonging to different runs, as well as to flush out the last run, with comparatively simple and uniform logic. (The proper handling of the last run produced by replacement selection turns out to be a bit tricky, and it has tended to be a stumbling block for programmers.) The principal idea is to consider each key as a pair (S, K), where K is the original key and S is the run number to which this record belongs. When such extended keys are lexicographically ordered, with S as major key and K as minor key, we obtain the output sequence produced by replacement selection.

The algorithm below uses a data structure containing P nodes to represent the selection tree; the jth node $X[j]$ is assumed to contain c words beginning in $\texttt{LOC}(X[j]) = L_0 + cj$, for $0 \le j < P$, and it represents both internal node number j and external node number $P + j$ in Fig. 63. There are several named fields in each node:

KEY = the key stored in this external node;
RECORD = the record stored in this external node (including KEY as a subfield);
LOSER = pointer to the "loser" stored in this internal node;
RN = run number of the record pointed to by LOSER;
FE = pointer to internal node above this external node in the tree;
FI = pointer to internal node above this internal node in the tree.

For example, when $P = 12$, internal node number 5 and external node number 17 of Fig. 63 would both be represented in $X[5]$, by the fields KEY = 170, LOSER = $L_0 + 9c$ (the address of external node number 21), FE = $L_0 + 8c$, FI = $L_0 + 2c$.

The FE and FI fields have constant values, so they need not appear explicitly in memory; however, the initial phase of external sorting sometimes has trouble keeping up with the I/O devices, and it is often worth while to store these redundant values with the data instead of recomputing them each time.

Algorithm R (*Replacement selection.*) This algorithm reads records sequentially from an input file and writes them sequentially onto an output file, producing RMAX runs whose length is P or more (except for the final run). There are $P \ge 2$ nodes, $X[0], \ldots, X[P - 1]$, having fields as described above.

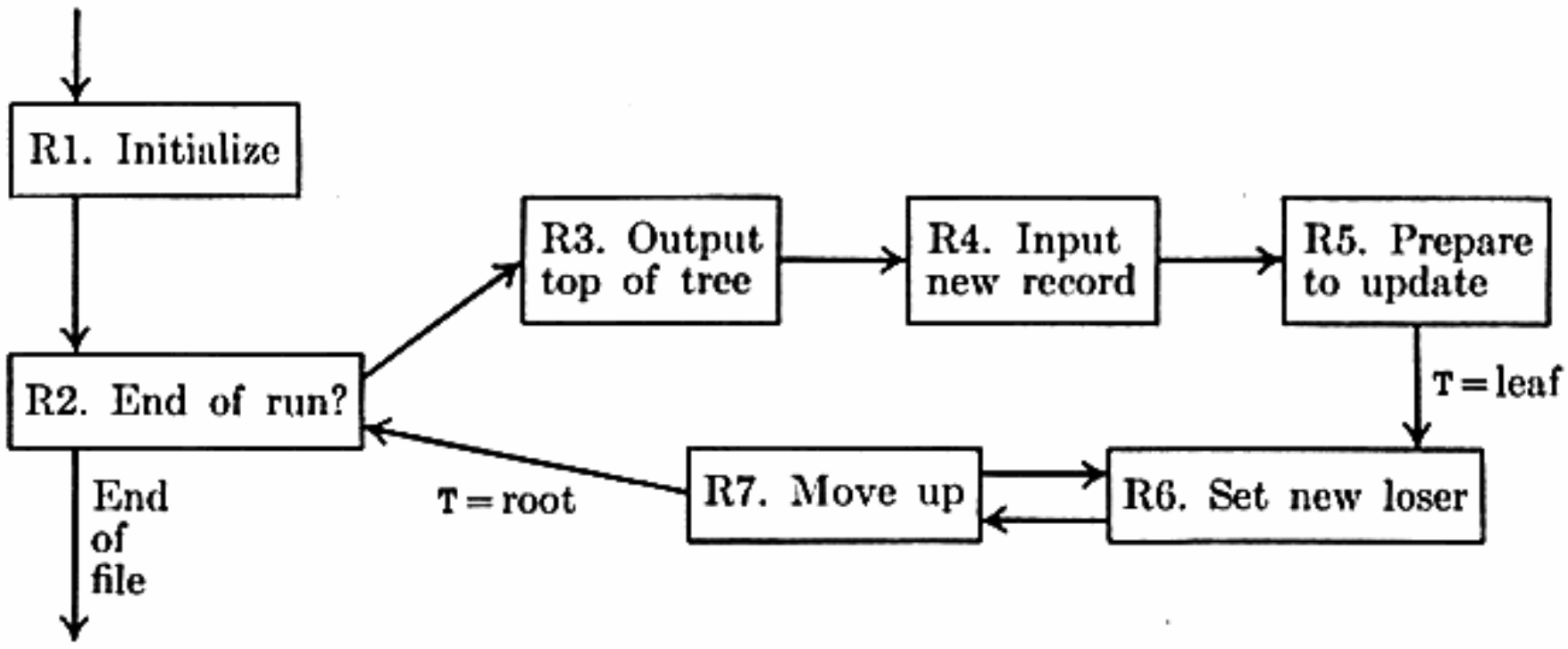


Fig. 66. Making initial runs by replacement selection.

R1. [Initialize.] Set $\texttt{RMAX} \leftarrow 0$, $\texttt{RC} \leftarrow 0$, $\texttt{LASTKEY} \leftarrow \infty$, $\texttt{Q} \leftarrow \texttt{LOC}(X[0])$, and $\texttt{RQ} \leftarrow 0$. (RC is the number of the current run and LASTKEY is the key of the last record output. The initial setting of LASTKEY should be larger than any possible key; cf. exercise 8.) For $0 \le j < P$, set the initial contents of $X[j]$ as follows, when $\texttt{J} = \texttt{LOC}(X[j])$:

$$\texttt{LOSER(J)} \leftarrow \texttt{J}; \qquad \texttt{RN(J)} \leftarrow 0;$$
$$\texttt{FE(J)} \leftarrow \texttt{LOC}(X[\lfloor (P+j)/2 \rfloor]); \qquad \texttt{FI(J)} \leftarrow \texttt{LOC}(X[\lfloor j/2 \rfloor]).$$

(The settings of LOSER(J) and RN(J) are artificial ways to get the tree initialized by considering a fictitious run number 0 which is never output. This is tricky; see exercise 10.)

R2. [End of run?] If $\texttt{RQ} = \texttt{RC}$, go on to step R3. (Otherwise $\texttt{RQ} = \texttt{RC} + 1$ and we have just completed run number RC; any special actions required by a merging pattern for subsequent passes of the sort would be done at this point.) If $\texttt{RQ} > \texttt{RMAX}$, stop; otherwise set $\texttt{RC} \leftarrow \texttt{RQ}$.

R3. [Output top of tree.] (Now Q points to the "champion," and RQ is its run number.) If $\texttt{RQ} \neq 0$, output RECORD(Q) and set $\texttt{LASTKEY} \leftarrow \texttt{KEY(Q)}$.

R4. [Input new record.] If the input file is exhausted, set $\texttt{RQ} \leftarrow \texttt{RMAX+1}$ and go on to step R5. Otherwise set RECORD(Q) to the next record from the input file. If $\texttt{KEY(Q)} < \texttt{LASTKEY}$ (so that this new record does not belong to the current run), set $\texttt{RQ} \leftarrow \texttt{RQ} + 1$ and then if $\texttt{RQ} > \texttt{RMAX}$ set $\texttt{RMAX} \leftarrow \texttt{RQ}$.

R5. [Prepare to update.] (Now Q points to a new record, whose run number is RQ.) Set $\texttt{T} \leftarrow \texttt{FE(Q)}$. (T is a pointer variable which will move up the tree.)

R6. [Set new loser.] If $\texttt{RN(T)} < \texttt{RQ}$ or if $\texttt{RN(T)} = \texttt{RQ}$ and $\texttt{KEY(LOSER(T))} < \texttt{KEY(Q)}$, then interchange $\texttt{LOSER(T)} \leftrightarrow \texttt{Q}$, $\texttt{RN(T)} \leftrightarrow \texttt{RQ}$. (Variables Q and RQ keep track of the current winner and its run number.)

R7. [Move up.] If $\texttt{T} = \texttt{LOC}(X[1])$ then go back to R2, otherwise set $\texttt{T} \leftarrow \texttt{FI(T)}$ and return to R6. ∎

Algorithm R speaks of input and output of records one at a time, while in practice it is best to read and write relatively large blocks of records. Therefore some input and output buffers are actually present in memory, behind the scenes, effectively lowering the size of P. We shall illustrate this in Section 5.4.6.

E. H. Friend [*JACM* **3** (1956), 154] suggested extending replacement selection so that whenever an input key is less than LASTKEY (so that it doesn't fit in the current run), but greater than or equal to the last key actually written onto tape (so that it actually could fit into the current run after all), it should be inserted into the output buffer. Furthermore, some computers have "scatter read" and "gather write" facilities for doing I/O to and from nonconsecutive locations, and this leads to some techniques which overlap buffers with the selection tree.

***Delayed reconstitution of runs.** A very interesting way to improve on replacement selection has been suggested by R. J. Dinsmore [*CACM* **8** (1965), 48] using a concept which we shall call *degrees of freedom*. As we have seen, each block of records on tape within a run is in nondecreasing order, so that its first element is the lowest and its last element is the highest. In the ordinary process of replacement selection, the lowest element of each block within a run is never less than the highest element of the preceding block in that run; this is "1 degree of freedom." Dinsmore suggests relaxing this condition to "m degrees of freedom," where the lowest element of each block may be less than the highest element of the preceding block so long as it is *not less than the highest elements in m different preceding blocks of the same run*. Records within individual blocks are ordered, as before, but adjacent blocks need not be in order.

For example, suppose that there are just two records per block; the following sequence of blocks is a run with three degrees of freedom:

$$\mid 08\ \ 50 \mid 06\ \ 90 \mid 17\ \ 27 \mid 42\ \ 67 \mid 51\ \ 89 \mid \qquad (1)$$

A subsequent block which is to be part of the same run must begin with an element not less than the third largest element of {50, 90, 27, 67, 89}, namely 67. The sequence (1) would not be a run if there were only two degrees of freedom, since 17 is less than both 50 and 90.

A run with m degrees of freedom can be "reconstituted" while it is being read during the next phase of sorting, so that for all practical purposes it is a run in the ordinary sense. We start by reading the first m blocks into m buffers, and doing an m-way merge on them; when one buffer is exhausted, we replace it with the $(m + 1)$st block, and so on. In this way we can recover the run as a single sequence, for the first word of every newly-read block must be greater than or equal to the last word of the just-exhausted block (lest it be less than the highest elements in m different blocks that precede it). This method of reconstituting the run is essentially like an m-way merge using a single tape unit for all the input blocks! The reconstitution procedure acts as a coroutine which is called upon to deliver one record of the run at a time. We could be reconstituting different runs from different tape units with different degrees of

freedom, and merging the resulting runs, all at the same time, in essentially the same way as the four-way merge illustrated at the beginning of this section may be thought of as several two-way merges going on at once.

This ingenious idea has not yet been thoroughly analyzed. Preliminary indications are that when P is large compared to the block size the run length is about $2.1P$ when $m = 2$, $2.3P$ when $m = 4$, $2.5P$ when $m = 8$. This increase may not be enough to justify the added complication; on the other hand, it may be advantageous when there is room for a rather large number of buffers during the second phase of sorting.

***Natural selection.** Another way to increase the run lengths produced by replacement selection has been explored by W. D. Frazer and C. K. Wong. Their idea is to proceed as in Algorithm R, but when `KEY(Q)` $<$ `LASTKEY` in step R4, the new `RECORD(Q)` is not left in the tree, it is output into an external *reservoir* instead, and another new record is read in. This process continues until the reservoir is filled with a certain number of records, P'; at this time the remainder of the current run is output from the tree, and the reservoir items are used as input for the next run.

This method tends to produce longer runs than replacement selection because it avoids "dead" records from the next string, instead of letting them fill up the tree; but it requires extra time for input and output to and from the reservoir. When $P' > P$ it is possible that some records will be placed into the reservoir twice, but when $P' \le P$ this will never happen.

Frazer and Wong made extensive empirical tests of their method, noticing that when P is reasonably large (say $P \ge 32$) and $P' = P$ the average run length for random data is given by eP, where $e \approx 2.718$ is the base of natural logarithms. This phenomenon, and the fact that the method is an evolutionary improvement over simple replacement selection, naturally led them to call their method *natural selection*.

The "natural" law for run lengths can be proved by considering the snowplow of Fig. 64 again, and applying elementary calculus. Let L be the length of the track, and let $x(t)$ be the position of the snowplow at time t, for $0 \le t \le T$. The reservoir is assumed to be full at time T, when the snow stops temporarily while the plow returns to its starting position (clearing the P units of snow remaining in its path). The situation is the same as before except that the "balance condition" is different; instead of P units of snow on the road at all times, we have P units of snow in front of the plow, and the reservoir (behind the plow) gets up to P units. The snowplow advances by dx during a time interval dt if $h(x, t)\,dx$ records are output, where $h(x, t)$ is the height of the snow at time t and position $x = x(t)$, measured in suitable units; hence $h(x, t) = h(x, 0) + Kt$ for all x. Since the number of records in memory stays constant, $h(x, t)\,dx$ is also the number of records that are input *ahead* of the plow, namely $K\,dt(L - x)$, where K is the rate of snowfall (see Fig. 67). Thus

$$\frac{dx}{dt} = \frac{K(L - x)}{h(x, t)}. \tag{2}$$

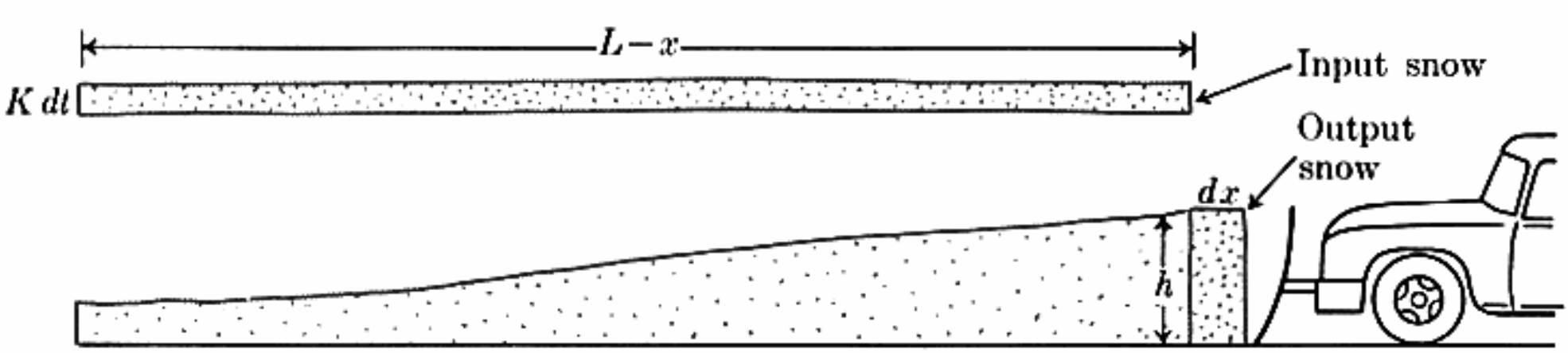


Fig. 67. Equal amounts of snow are input and output; the plow moves dx in time dt.

Fortunately, it turns out that $h(x, t)$ is constant, equal to KT, whenever $x = x(t)$ and $0 \le t \le T$, since the snow falls steadily at position $x(t)$ for $T - t$ units of time after the plow passes that point, plus t units of time before it comes back. In other words, the plow sees all snow at the same height on his journey, assuming that a steady state has been reached where each journey is the same. Hence the total amount of snow cleared (the run length) is KTL; and the amount of snow in memory is the amount cleared after time T, namely $KT(L - x(T))$. The solution to (2) such that $x(0) = 0$ is

$$x(t) = L(1 - e^{-t/T}); \tag{3}$$

hence $P = KTLe^{-1} =$ (run length)$/e$; and this is what we set out to prove.

Exercises 21 through 23 show that this analysis can be extended to the case of general P'; for example, when $P' = 2P$ the average run length turns out to be $e^{\theta}(e - \theta)P$, where $\theta = \frac{1}{2}(e - \sqrt{e^2 - 4})$, a result that probably wouldn't have been guessed offhand! Table 2 shows the dependence of run length on reservoir size; the usefulness of natural selection in a given computer environment can be estimated by referring to this table.

Table 2

RUN LENGTHS BY NATURAL SELECTION

Reservoir size	Run length	Parameter	Reservoir size	Run length	Parameter
1.00000P	2.71828P	1.00000	0.38629P	2.00000P	0.69315
2.00000P	3.53487P	1.43867	1.30432P	3.00000P	1.15881
3.00000P	4.16220P	1.74773	2.72294P	4.00000P	1.66862
4.00000P	4.69445P	2.01212	4.63853P	5.00000P	2.16714
5.00000P	5.16369P	2.24938	21.72222P	10.00000P	4.66667
10.00000P	7.00877P	3.17122	5.29143P	5.29143P	2.31329

The "parameter" $k + \theta$ is defined in exercise 22.

***Analysis of replacement selection.** Let us now return to the case of replacement selection without an auxiliary reservoir. The snowplow analogy gives us a fairly good indication of the average length of runs obtained by replacement

selection "in the limit," but it is possible to get much more precise information about Algorithm R by applying the facts about runs in permutations which we have studied in Section 5.1.3. For this purpose it is convenient to assume that the input file is an arbitrarily long sequence of independent random real numbers between 0 and 1.

Let

$$g_P(z_1, z_2, \ldots, z_k) = \sum_{l_1, l_2, \ldots, l_k \geq 0} a_P(l_1, l_2, \ldots, l_k) z_1^{l_1} z_2^{l_2} \cdots z_k^{l_k}$$

be the generating function for run lengths produced by P-way replacement selection on such a file, where $a_P(l_1, l_2, \ldots, l_k)$ is the probability that the first run has length l_1, the second has length l_2, ..., the kth has length l_k. The following "independence theorem" is basic, since it reduces the analysis to the case $P = 1$:

Theorem K. $g_P(z_1, z_2, \ldots, z_k) = g_1(z_1, z_2, \ldots, z_k)^P$.

Proof. Let the input keys be $X_1, X_2, X_3, \ldots$. Algorithm R partitions them into P subsequences, according to which external node position they land in the tree; the subsequence containing X_n is determined by the values of $X_1, \ldots, X_{n-1}$. Each of these subsequences is therefore an independent sequence of independent random numbers between 0 and 1. Furthermore, the output of replacement selection is precisely what would be obtained by doing a P-way merge on these subsequences; an element belongs to the jth run of a subsequence if and only if it belongs to the jth run produced by replacement selection (since in step R4, `LASTKEY` and `KEY(Q)` belong to the same subsequence).

In other words, we might just as well assume that Algorithm R is being applied to P independent random input files, and that step R4 reads the next record from the file corresponding to external node `Q`; in this sense, the algorithm is equivalent to a P-way merge, with "step-downs" marking the ends of the runs.

Thus the output has runs of lengths $(l_1, \ldots, l_k)$ if and only if the subsequences have runs of respective lengths $(l_{11}, \ldots, l_{1k}), \ldots, (l_{P1}, \ldots, l_{Pk})$, where the l_{ij} are some nonnegative integers satisfying $\sum_{1 \leq i \leq P} l_{ij} = l_j$ for $1 \leq j \leq k$. It follows that

$$a_P(l_1, \ldots, l_k) = \sum_{\substack{l_{11} + \cdots + l_{P1} = l_1 \\ \vdots \\ l_{1k} + \cdots + l_{Pk} = l_k}} a_1(l_{11}, \ldots, l_{1k}) \cdots a_1(l_{P1}, \ldots, l_{Pk}),$$

and this is equivalent to the desired result. ∎

We have discussed the average length L_k of the kth run, when $P = 1$, in Section 5.1.3, where the values are tabulated in Table 5.1.3–2. Theorem K implies that the average length of the kth run for general P is P times as long as the average when $P = 1$, namely $L_k P$; and the variance is also P times as

large, so the standard deviation of the run length is proportional to $\sqrt{P}$. These results were first derived by B. J. Gassner about 1958.

Thus the first run produced by Algorithm R will be about $(e - 1)P \approx 1.718P$ records long, for random data; the second run will be about $(e^2 - 2e)P \approx 1.952P$ records long; the third, about $1.996P$; and subsequent runs will be very close to $2P$ records long until we get to the last two runs (see exercise 14). The standard deviation of most of these run lengths is approximately $(4e - 10)P \approx 0.934\sqrt{P}$ [*CACM* **6** (1963), 685–687]. Furthermore, exercise 5.1.3–10 shows that the *total* length of the first k runs will be fairly close to $(2k - \frac{1}{3})P$, with a standard deviation of $((\frac{2}{3}k + \frac{2}{9})P)^{1/2}$. The generating functions $g_1(z, z, \ldots, z)$ and $g_1(1, \ldots, 1, z)$ are derived in exercises 5.1.3–9 and 11.

The above analysis has assumed that the input file is infinitely long, but the proof of Theorem K shows that the same probability $a_p(l_1, \ldots, l_k)$ would be obtained in any random input sequence containing at least $l_1 + \cdots + l_k + P$ elements. So the above results are applicable for, say, files of size $N > (2k+1)P$, in view of the small standard deviation.

We will be seeing some applications in which the merging pattern wants some of the runs to be ascending and some to be descending. Since the residue accumulated in memory at the end of an ascending run tends to contain numbers somewhat smaller on the average than random data, a change in the direction of ordering decreases the average length of the runs. Consider, for example, a snowplow which must make a U-turn every time it reaches an end of a straight road; it will go very speedily over the area just plowed. The run lengths when directions are reversed vary between $1.5P$ and $2P$ for random data (see exercise 24).

EXERCISES

1. [*10*] What is Step 4, in the example of four-way merging at the beginning of this section?

2. [*12*] What changes would be made to the tree of Fig. 63 if the key 061 were replaced by 612?

3. [*16*] (E. F. Moore.) What output is produced by four-way replacement selection when it is applied to successive words of the following sentence:

```
fourscore and seven years ago our fathers brought forth
on this continent a new nation conceived in liberty and
dedicated to the proposition that all men are created equal.
```

(Use ordinary alphabetic order, treating each word as one key.)

4. [*16*] Apply four-way *natural* selection to the sentence in exercise 3, using a reservoir of capacity 4.

5. [*00*] True or false: Replacement selection using a tree works only when P is a power of 2 or the sum of two powers of 2.

6. [*15*] Algorithm R specifies that P must be ≥ 2; what comparatively small changes to the algorithm would make it valid for all $P \geq 1$?

7. [*17*] What does Algorithm R do when there is no input at all?

8. [*20*] Algorithm R makes use of an artificial key "∞" which must be larger than any possible key. Show that the algorithm might fail if an actual key were equal to ∞, and explain how to modify the algorithm in case the implementation of a true ∞ is inconvenient.

▸ **9.** [*23*] How would you modify Algorithm R so that it causes certain specified runs (depending on RC) to be output in ascending order, and others in descending order?

10. [*26*] The initial setting of the LOSER pointers in step R1 usually doesn't correspond to any actual tournament, since external node $P + j$ may not lie in the subtree below internal node j. Explain why Algorithm R works anyway. [*Hint:* Would the algorithm work if {LOSER(LOC(X[0])), . . . , LOSER(LOC(X[P — 1]))} were set to an *arbitrary* permutation of {LOC(X[0]), . . . , LOC(X[P — 1])} in step R1?]

11. [*M25*] True or false: The probability that KEY(Q) < LASTKEY in step R4 is approximately $\frac{1}{2}$, assuming random input.

12. [*M46*] Carry out a detailed analysis of the number of times each portion of Algorithm R is executed; for example, how often are interchanges made in step R6?

13. [*18*] Why is the second run produced by replacement selection usually longer than the first run?

▸ **14.** [*HM25*] Use the snowplow analogy to estimate the average length of the *last two runs* produced by replacement selection on a long sequence of input data.

15. [*20*] True or false: The final run produced by replacement selection never contains more than P records. Discuss your answer.

16. [*M26*] Find a "simple" necessary and sufficient condition that a file $R_1\ R_2 \ldots R_N$ will be completely sorted in one pass by P-way replacement selection. What is the probability that this happens, as a function of P and N, when the input is a random permutation of $\{1, 2, \ldots, N\}$?

17. [*20*] What is output by Algorithm R when the input keys are in decreasing order, $K_1 \geq K_2 \geq \cdots \geq K_N$?

▸ **18.** [*22*] What happens if Algorithm R is applied *again* to an output file that was produced by Algorithm R?

19. [*HM22*] Use the snowplow analogy to prove that the first run produced by replacement selection is approximately $(e - 1)P$ records long.

20. [*HM24*] Approximately how long is the first run produced by natural selection, when $P = P'$?

▸ **21.** [*HM23*] Determine the approximate length of runs produced by natural selection when $P' < P$.

22. [*HM40*] The purpose of this exercise is to determine the average run length obtained in natural selection, when $P' > P$. Let $\kappa = k + \theta$ be a real number ≥ 1,

where $k = \lfloor \kappa \rfloor$ and $\theta = \kappa \bmod 1$, and consider the function $F(\kappa) = F_k(\theta)$, where $F_k(\theta)$ is the polynomial defined by the generating function

$$\sum_{k \geq 0} F_k(\theta) z^k = e^{-\theta z}/(1 - ze^{1-z}).$$

Thus, $F_0(\theta) = 1$, $F_1(\theta) = e - \theta$, $F_2(\theta) = e^2 - e - e\theta + \frac{1}{2}\theta^2$, etc.

Suppose that a snowplow starts out at time 0 to simulate the process of natural selection, and suppose that after T units of time exactly P snowflakes have fallen behind it. At this point a second snowplow begins on the same journey, occupying the same position at time $t + T$ as the first snowplow did at time t. Finally, at time κT, exactly P' snowflakes have fallen behind the first snowplow; he instantaneously plows the rest of the road and disappears.

Using this model to represent the process of natural selection, show that a run length equal to $e^\theta F(\kappa) P$ is obtained when

$$P'/P = k + 1 + e^\theta \left(\kappa F(\kappa) - \sum_{0 \leq j \leq \kappa} F(\kappa - j) \right).$$

23. [*HM35*] The preceding exercise analyzes natural selection when the records from the reservoir are always read in the same order that they were written, first-in-first-out. Find the approximate run length which would be obtained if the reservoir contents from the preceding run were read in completely *random* order, as if the records in the reservoir had been thoroughly shuffled between runs.

24. [*HM39*] The purpose of this exercise is to analyze the effect caused by haphazardly changing the direction of runs in replacement selection.

a) Let $g_P(z_1, z_2, \ldots, z_k)$ be a generating function defined as in Theorem K, but with each of the k runs specified as to whether they are to be ascending or descending. For example, we might say that all odd-numbered runs are ascending, all even-numbered runs are descending. Show that Theorem K is valid for each of the 2^k generating functions of this type.

b) As a consequence of (a), we may assume that $P = 1$. We may also assume that the input is a uniformly distributed sequence of independent random numbers between 0 and 1. Let

$$a(x, y) = \begin{cases} e^{1-x} - e^{y-x}, & \text{if } x \leq y; \\ e^{1-x}, & \text{if } x > y. \end{cases}$$

Given that $f(x)\,dx$ is the probability that a certain ascending run begins with x, prove that $(\int_0^1 a(x, y) f(x)\,dx)\,dy$ is the probability that the following run beings with y. [*Hint:* Consider, for each $n \geq 0$, the probability that $x \leq X_1 \leq \cdots \leq X_n > y$, when x and y are given.]

c) Consider runs that change direction with probability p; in other words, the direction of each run after the first is randomly chosen to be the same as that of the previous run, $q = (1 - p)$ of the time, but it is to be in the opposite direction p of the time. (Thus when $p = 0$, all runs have the same direction; when $p = 1$,

the runs alternate in direction; and when $p = \frac{1}{2}$, the runs are independently random.) Let

$$f_1(x) = 1, \qquad f_{n+1}(y) = p\int_0^1 a(x, y)f_n(1 - x)\,dx + q\int_0^1 a(x, y)f_n(x)\,dx.$$

Show that the probability that the nth run begins with x is $f_n(x)\,dx$ when the $(n - 1)$st run is ascending, $f_n(1 - x)\,dx$ when the $(n - 1)$st run is descending.

d) Find a solution f to the "steady state" equations

$$f(y) = p\int_0^1 a(x, y)f(1 - x)\,dx + q\int_0^1 a(x, y)f(x)\,dx, \qquad \int_0^1 f(x)\,dx = 1.$$

[*Hint:* Show that $f''(x)$ is independent of x.]

e) Show that the sequence $f_n(x)$ in part (c) converges rather rapidly to the function $f(x)$ in part (d). [*Hint:* See, for example, the proof of Theorem 4.5.3L.]

f) Show that the average length of an ascending run starting with x is e^{1-x}.

g) Finally, put the above results together to prove the following theorem: *If the directions of consecutive runs are independently reversed with probability p in replacement selection, the average run length approaches* $(6/(3 + p))P$.

(The case $p = 1$ of this theorem was first derived by Knuth [*CACM* **6** (1963), 685–688]; the case $p = \frac{1}{2}$ was first proved by A. G. Konheim in 1970.)

25. [*HM40*] Consider the following procedure:

N1. Read a record into a one-word "reservoir." Then read another record, R, and let K be its key.

N2. Output the reservoir, set LASTKEY to its key, and set the reservoir empty.

N3. If K < LASTKEY then output R and set LASTKEY ← K and go to N5.

N4. If the reservoir is nonempty, return to N2; otherwise enter R into the reservoir.

N5. Read in a new record, R, and let K be its key. Go to N3. ▮

This is essentially equivalent to natural selection with $P = 1$ and with $P' = 1$ or 2 (depending on whether you choose to empty the reservoir at the moment it fills or at the moment it is about to overfill), except that it produces *descending* runs, and it never stops. The latter anomalies are convenient and harmless assumptions for the purposes of this problem.

Proceeding as in exercise 24, let $f_n(x, y)\,dy\,dx$ be the probability that (x, y) are the respective values of (LASTKEY, K) just after the nth time step N2 is performed. Prove that there is a function $g_n(x)$ of one variable such that $f_n(x, y) = g_n(x)$ when $x < y$, and $f_n(x, y) = g_n(x) - e^{-y}(g_n(x) - g_n(y))$ when $x > y$. This function $g_n(x)$ is defined by the relations $g_1(x) = 1$,

$$g_{n+1}(x) = \int_0^x e^u g_n(u)\,du + \int_0^x dv(v + 1)\int_v^1 du((e^v - 1)g_n(u) + g_n(v))$$
$$+ x\int_x^1 dv\int_v^1 du((e^v - 1)g_n(u) + g_n(v)).$$

Show further that the expected length of the nth run is

$$\int_0^1 dx \int_0^x dy(g_n(x)(e^y - 1) + g_n(y))(2 - \tfrac{1}{2}y^2) + \int_0^1 dx(1 - x)g_n(x)e^x.$$

[*Note:* The "steady state" solution to these equations appears to be very complicated; it has been obtained numerically by J. McKenna, showing that the run lengths approach a limiting value of 2.61307209. Theorem K does not apply to natural selection, so the case $P = 1$ does not carry over to other P.

26. [*M33*] Considering the algorithm in exercise 25 as a definition of natural selection when $P' = 1$, find the expected length of the *first* run when $P' = r$, for any $r \geq 0$, as follows.

a) Show that the first run has length n with probability

$$(n + r)\begin{bmatrix} n + r \\ n \end{bmatrix} / (n + r + 1)!.$$

b) Define "second order Stirling numbers" $\left[\!\left[{n \atop m} \right]\!\right]$ by the rules

$$\left[\!\left[{0 \atop m} \right]\!\right] = \delta_{m0}, \qquad \left[\!\left[{n \atop m} \right]\!\right] = (n + m - 1)\left(\left[\!\left[{n - 1 \atop m} \right]\!\right] + \left[\!\left[{n - 1 \atop m - 1} \right]\!\right]\right) \quad \text{for} \quad n > 0.$$

Prove that

$$\begin{bmatrix} n + r \\ n \end{bmatrix} = \sum_{0 \leq k \leq r} \binom{n + r}{k + r} \left[\!\left[{r \atop k} \right]\!\right].$$

c) Prove that the average length of the first run is therefore $c_r e - r - 1$, where

$$c_r = \sum_{0 \leq k \leq r} \left[\!\left[{r \atop k} \right]\!\right] (r + k + 1)/(r + k)!.$$

27. [*25*] The text considers only the case that all records to be sorted have a fixed size. How can replacement selection be done reasonably well on *variable-length* records?

5.4.2. The Polyphase Merge

Now that we have seen how initial runs can be built up, we shall consider various patterns which can be used to distribute them onto tapes and to merge them together until only a single run remains.

Let us begin by assuming that there are three tape units, T1, T2, and T3, available; the technique of "balanced merging," described near the beginning of Section 5.4, can be used with $P = 2$ and $T = 3$, when it takes the following form:

B1. Distribute initial runs alternately on tapes T1 and T2.

B2. Merge runs from T1 and T2 onto T3; then stop if T3 contains only one run.

B3. Copy the runs of T3 alternately onto T1 and T2, then return to B2. ▮

If the initial distribution pass produces S runs, the first merge pass will produce $\lceil S/2 \rceil$ runs on T3, the second will produce $\lceil S/4 \rceil$, etc. Thus if, say, $17 \le S \le 32$, we will have 1 distribution pass, 5 merge passes, and 4 copy passes; in general, if $S > 1$, the number of passes over all the data is $2\lceil \log_2 S \rceil$.

The copying passes in this procedure are undesirable, since they do not reduce the number of runs. Half of the copying can be avoided if we use a *two-phase* procedure:

A1. Distribute initial runs alternately on tapes T1 and T2.

A2. Merge runs from T1 and T2 onto T3; then stop if T3 contains only one run.

A3. Copy *half* of the runs from T3 onto T1.

A4. Merge runs from T1 and T3 onto T2; then stop if T2 contains only one run.

A5. Copy *half* of the runs from T2 onto T1. Return to A2. ▮

The number of passes over the data has been reduced to $\frac{3}{2}\lceil \log_2 S \rceil + \frac{1}{2}$, since steps A3 and A5 do only "half a pass"; about 25 percent of the time has therefore been saved.

The copying can actually be eliminated *entirely*, if we start with F_n runs on T1 and F_{n-1} runs on T2, where F_n and F_{n-1} are consecutive Fibonacci numbers. Consider, for example, the case $n = 7$, $S = F_n + F_{n-1} = 13 + 8 = 21$:

	Contents of T1	Contents of T2	Contents of T3	Remarks
Phase 1.	1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1	1, 1, 1, 1, 1, 1, 1, 1		Initial distribution
Phase 2.	1, 1, 1, 1, 1	—	2, 2, 2, 2, 2, 2, 2, 2	Merge 8 runs to T3
Phase 3.	—	3, 3, 3, 3, 3	2, 2, 2	Merge 5 runs to T2
Phase 4.	5, 5, 5	3, 3	—	Merge 3 runs to T1
Phase 5.	5	—	8, 8	Merge 2 runs to T3
Phase 6.	—	13	8	Merge 1 run to T2
Phase 7.	21	—	—	Merge 1 run to T1

Here, for example, "2, 2, 2, 2, 2, 2, 2, 2" denotes eight runs of relative length 2, considering each initial run to be of relative length 1. Fibonacci numbers are omnipresent in this chart!

Only phases 1 and 7 are complete passes over the data; phase 2 processes only 16/21 of the initial runs, phase 3 only 15/21, etc., and so the total number of "passes" comes to $(21 + 16 + 15 + 15 + 16 + 13 + 21)/21 = 5\frac{4}{7}$ if we assume that the initial runs have approximately equal length. By comparison, the two-phase procedure above would have required 8 passes to sort these 21 initial runs. We shall see that in general this "Fibonacci" pattern requires approximately $1.04 \log_2 S + 0.99$ passes, making it competitive with a *four*-tape balanced merge although it requires only three tapes.

The same idea can be generalized to T tapes, for any $T \ge 3$, using $(T - 1)$-way merging. We shall see, for example, that the four-tape case requires only about $.703 \log_2 S + 0.96$ passes over the data. The generalized pattern involves generalized Fibonacci numbers. Consider the following six-tape example:

	T1	T2	T3	T4	T5	T6	Initial runs processed
Phase 1.	1^{31}	1^{30}	1^{28}	1^{24}	1^{16}	—	$31 + 30 + 28 + 24 + 16 = 129$
Phase 2.	1^{15}	1^{14}	1^{12}	1^{8}	—	5^{16}	$16 \times 5 = 80$
Phase 3.	1^{7}	1^{6}	1^{4}	—	9^{8}	5^{8}	$8 \times 9 = 72$
Phase 4.	1^{3}	1^{2}	—	17^{4}	9^{4}	5^{4}	$4 \times 17 = 68$
Phase 5.	1^{1}	—	33^{2}	17^{2}	9^{2}	5^{2}	$2 \times 33 = 66$
Phase 6.	—	65^{1}	33^{1}	17^{1}	9^{1}	5^{1}	$1 \times 65 = 65$
Phase 7.	129^{1}	—	—	—	—	—	$1 \times 129 = 129$

Here 1^{31} stands for 31 runs of relative length 1, etc.; five-way merges have been used throughout. This general pattern was developed by R. L. Gilstad [*Proc. AFIPS Eastern Jt. Computer Conf.* **18** (1960), 143–148], who called it the *polyphase merge*. The three-tape case had been discovered earlier by B. K. Betz [unpublished memorandum, Minneapolis-Honeywell Regulator Co. (1956)].

In order to make polyphase merging work as in the above examples, we need to have a "perfect Fibonacci distribution" of runs on the tapes after each phase. By reading the above table from bottom to top, we can see that the first seven perfect Fibonacci distributions when $T = 6$ are $\{1, 0, 0, 0, 0\}$, $\{1, 1, 1, 1, 1\}$, $\{2, 2, 2, 2, 1\}$, $\{4, 4, 4, 3, 2\}$, $\{8, 8, 7, 6, 4\}$, $\{16, 15, 14, 12, 8\}$, and $\{31, 30, 28, 24, 16\}$. The big questions now facing us are

1. What is the rule underlying these perfect Fibonacci distributions?

2. What do we do if S does not correspond to a perfect Fibonacci distribution?

3. How should we design the initial distribution pass so that it produces the desired configuration on the tapes?

4. How many "passes" over the data will a T-tape polyphase merge require, as a function of S (the number of initial runs)?

We shall discuss these four questions in turn, first giving "easy answers" and then making a more intensive analysis.

The perfect Fibonacci distributions can be obtained by running the pattern backwards, cyclically rotating the tape contents. For example, when $T = 6$ we have the following distributions of runs:

Level	T1	T2	T3	T4	T5	Total	Final output will be on
0	1	0	0	0	0	1	T1
1	1	1	1	1	1	5	T6
2	2	2	2	2	1	9	T5
3	4	4	4	3	2	17	T4
4	8	8	7	6	4	33	T3
5	16	15	14	12	8	65	T2
6	31	30	28	24	16	129	T1
7	61	59	55	47	31	253	T6
8	120	116	108	92	61	497	T5

$$\begin{array}{cccccccc}
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
n & a_n & b_n & c_n & d_n & e_n & t_n & \mathrm{T}(k) \\
n+1 & a_n+b_n & a_n+c_n & a_n+d_n & a_n+e_n & a_n & t_n+4a_n & \mathrm{T}(k-1) \\
\cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots
\end{array} \tag{1}$$

(Tape T6 will always be empty after the initial distribution.)

The rule for going from level n to level $n+1$ shows that the condition

$$a_n \geq b_n \geq c_n \geq d_n \geq e_n \tag{2}$$

will hold in every level. In fact, it is easy to see from (1) that

$$\begin{aligned}
e_n &= a_{n-1}, \\
d_n &= a_{n-1} + e_{n-1} = a_{n-1} + a_{n-2}, \\
c_n &= a_{n-1} + d_{n-1} = a_{n-1} + a_{n-2} + a_{n-3}, \\
b_n &= a_{n-1} + c_{n-1} = a_{n-1} + a_{n-2} + a_{n-3} + a_{n-4}, \\
a_n &= a_{n-1} + b_{n-1} = a_{n-1} + a_{n-2} + a_{n-3} + a_{n-4} + a_{n-5},
\end{aligned} \tag{3}$$

where $a_0 = 1$ and where we let $a_n = 0$ for $n = -1, -2, -3, -4$.

The *pth order Fibonacci numbers* $F_n^{(p)}$ are defined by the rules

$$\begin{gathered}
F_n^{(p)} = F_{n-1}^{(p)} + F_{n-2}^{(p)} + \cdots + F_{n-p}^{(p)}, \qquad \text{for} \qquad n \geq p; \\
F_n^{(p)} = 0 \qquad \text{for} \qquad 0 \leq n \leq p-2; \qquad F_{p-1}^{(p)} = 1.
\end{gathered} \tag{4}$$

In other words, we start with $p-1$ 0's, then 1, and then each number is the sum of the preceding p values. When $p = 2$, this is the usual Fibonacci sequence; for larger values of p the sequence was apparently first studied by V. Schlegel in *El Progreso Matemático* **4** (1894), 173–174. Schlegel derived the generating function

$$\sum_{n \geq 0} F_n^{(p)} z^n = \frac{z^{p-1}}{1 - z - z^2 - \cdots - z^p} = \frac{z^{p-1} - z^p}{1 - 2z + z^{p+1}}. \tag{5}$$

Equation (3) shows that the number of runs on T1 during a six-tape polyphase merge is a fifth-order Fibonacci number: $a_n = F_{n+4}^{(5)}$.

In general, if we set $P = T - 1$, the polyphase merge distributions for T tapes will correspond to Pth order Fibonacci numbers in the same way. The kth tape gets

$$F_{n+P-2}^{(P)} + F_{n+P-3}^{(P)} + \cdots + F_{n+k-2}^{(P)}$$

initial runs in the perfect nth level distribution, for $1 \leq k \leq P$, and the total number of initial runs on all tapes is therefore

$$t_n = PF_{n+P-2}^{(P)} + (P-1)F_{n+P-3}^{(P)} + \cdots + F_{n-1}^{(P)}. \tag{6}$$

This settles the issue of "perfect Fibonacci distributions." But what should we do if S is not exactly equal to t_n, for any n? And how do we get the runs onto the tapes in the first place?

When S isn't perfect (and so few values are), we can do just as we did in balanced P-way merging, adding artificial "dummy runs" so that we can pretend S is perfect after all. There are several ways to add the dummy runs, and we aren't ready yet to analyze the "best" way of doing this. We shall discuss first a method of distribution and dummy-run assignment which isn't strictly optimal, although it has the virtue of simplicity and appears to be better than all other equally simple methods.

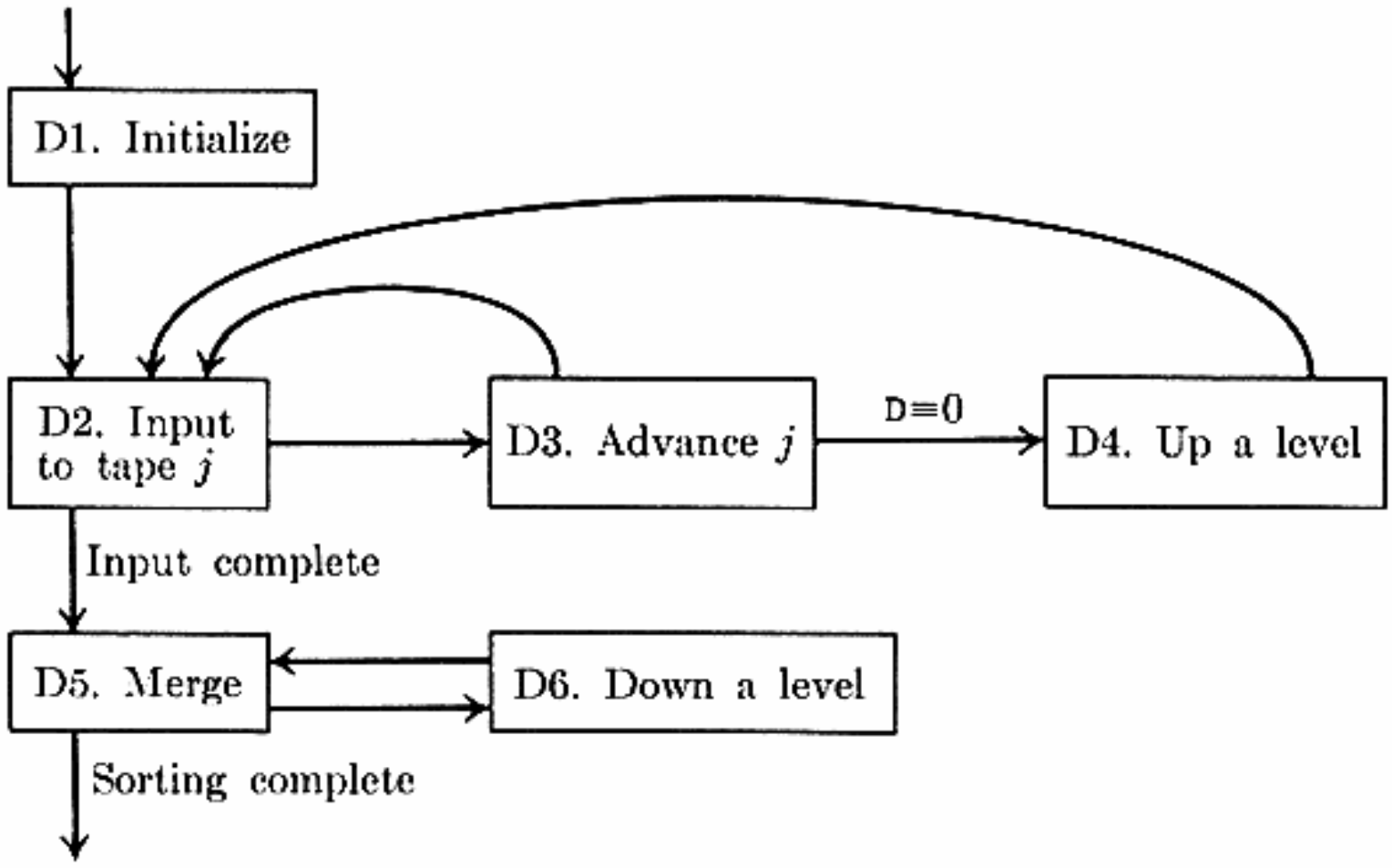


Fig. 68. Polyphase merge sorting.

Algorithm D (*Polyphase merge sorting with "horizontal" distribution*). This algorithm takes initial runs and disperses them to tapes, one run at a time, until the supply of initial runs is exhausted. Then it specifies how the tapes are to be merged, assuming that there are $T = P + 1 \geq 3$ available tape units, using P-way merging. Tape T may be used to hold the input, since it does not receive any initial runs. The following tables are maintained:

A[j], $1 \leq j \leq T$: The perfect Fibonacci distribution we are striving for.

D[j], $1 \leq j \leq T$: Number of dummy runs assumed to be present at the beginning of logical tape unit number j.

TAPE[j], $1 \leq j \leq T$: Number of the physical tape unit corresponding to logical tape unit number j.

(It is convenient to deal with "logical tape unit numbers" whose assignment to physical tape units varies as the algorithm proceeds.)

D1. [Initialize.] Set A[j] $\leftarrow$ D[j] $\leftarrow 1$ and TAPE[j] $\leftarrow j$, for $1 \leq j < T$. Set A[T] $\leftarrow$ D[T] $\leftarrow 0$ and TAPE[T] $\leftarrow T$. Then set $l \leftarrow 1$; $j \leftarrow 1$.

D2. [Input to tape j.] Write one run on tape number j, and decrease D[j] by 1. Then if the input is exhausted, rewind all the tapes and go to step D5.

D3. [Advance j.] If D[j] < D[$j+1$], increase j by 1 and return to D2. Otherwise if D[j] = 0, go on to D4. Otherwise set $j \leftarrow 1$ and return to D2.

D4. [Up a level.] Set $l \leftarrow l+1$, $a \leftarrow$ A[1], and then for $j = 1, 2, \ldots, P$ (in this order) set D[j] $\leftarrow a +$ A[$j+1$] $-$ A[j] and A[j] $\leftarrow a +$ A[$j+1$]. (See (1) and note that A[$P+1$] is always zero. At this point we will have D[1] $\geq$ D[2] $\geq \cdots \geq$ D[T].)
Now set $j \leftarrow 1$ and return to D2.

D5. [Merge.] If $l = 0$, sorting is complete and the output is on TAPE[1]. Otherwise, merge runs from TAPE[1], ..., TAPE[P] onto TAPE[T] until TAPE[P] is empty and D[P] = 0. The merging process should operate as follows, for each run merged: If D[j] > 0 for all j, $1 \leq j \leq P$, then increase D[T] by 1 and decrease each D[j] by 1 for $1 \leq j \leq P$; otherwise merge one run from each TAPE[j] such that D[j] = 0, and decrease D[j] by 1 for each other j. (Thus the dummy runs are imagined to be at the *beginning* of the tape instead of at the ending.)

D6. [Down a level.] Set $l \leftarrow l-1$. Rewind TAPE[P] and TAPE[T]. (Actually the rewinding of TAPE[P] could have been initiated during step D5, just after its last block was input.) Then set (TAPE[1], TAPE[2], ..., TAPE[T]) $\leftarrow$ (TAPE[T], TAPE[1], ..., TAPE[$T-1$]), (D[1], D[2], ..., D[T]) $\leftarrow$ (D[T], D[1], ..., D[$T-1$]), and return to step D5. ∎

The distribution rule which is stated so succinctly in step D3 of this algorithm is intended to equalize the number of dummies on each tape as well as possible. Figure 69 illustrates the order of distribution when we go from level 4

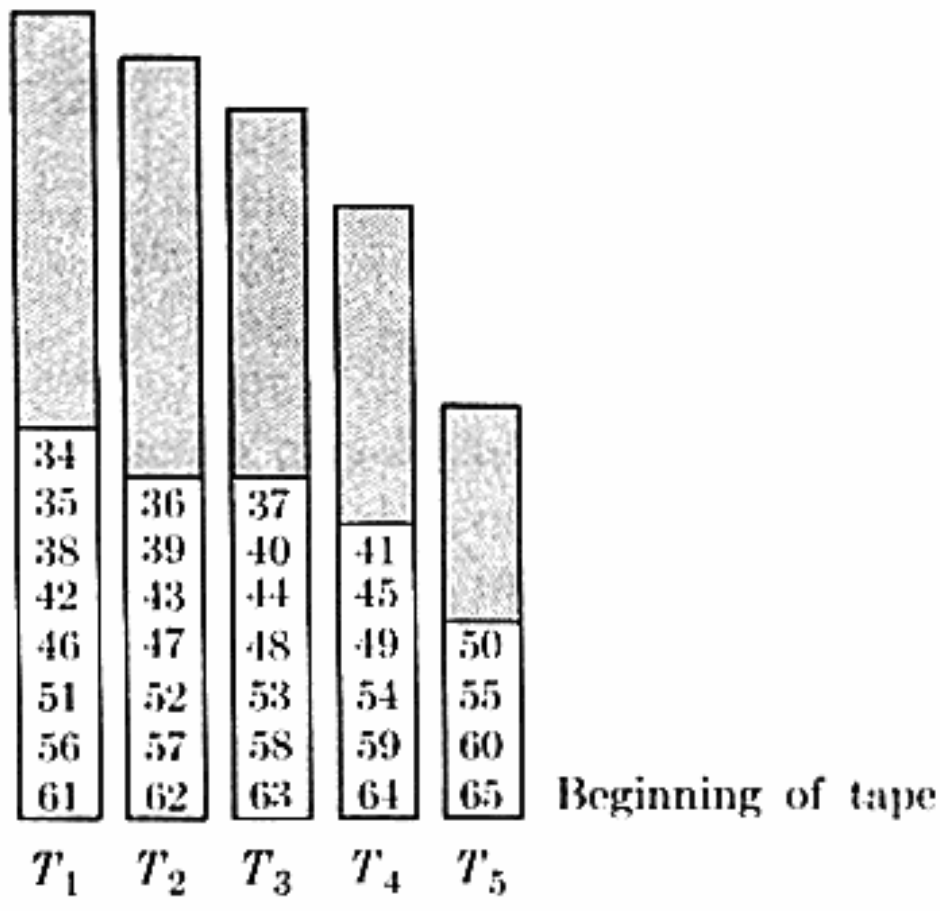


Fig. 69. The order in which runs 34 through 65 are distributed to tapes, when advancing from level 4 to level 5. (See the table of perfect distributions at the bottom of page 268.) Shaded areas represent the first 33 runs which were distributed when level 4 was reached.

(33 runs) to level 5 (65 runs) in a six-tape sort; if there were only, say, 53 initial runs, all runs numbered 54 and higher would be treated as dummies. (The runs are actually being written at the end of the tape, but it is best to imagine them being written at the beginning, since the dummies are assumed to be at the beginning.)

We have now discussed the first three questions listed above, and it remains to consider the number of "passes" over the data. Comparing our six-tape example to the table (1), we see that the total number of initial runs processed when $S = t_6$ was $a_5t_1 + a_4t_2 + a_3t_3 + a_2t_4 + a_1t_5 + a_0t_6$, excluding the initial distribution pass. Exercise 4 derives the generating functions

$$a(z) = \sum_{n\geq 0} a_n z^n = \frac{1}{1 - z - z^2 - z^3 - z^4 - z^5},$$
$$t(z) = \sum_{n\geq 1} t_n z^n = \frac{5z + 4z^2 + 3z^3 + 2z^4 + z^5}{1 - z - z^2 - z^3 - z^4 - z^5}. \tag{7}$$

It follows that, in general, the number of initial runs processed when $S = t_n$ is exactly the coefficient of z^n in $a(z) \cdot t(z)$, plus t_n (for the initial distribution pass). This makes it possible to calculate the asymptotic behavior of polyphase merging, as shown in exercises 5–7, and we obtain the results displayed in Table 1.

Table 1

APPROXIMATE BEHAVIOR OF POLYPHASE MERGE SORTING

Tapes	Phases	Passes	Pass/phase, percent	Growth ratio
3	$2.078 \ln S + 0.672$	$1.504 \ln S + 0.992$	72	1.6180340
4	$1.641 \ln S + 0.364$	$1.015 \ln S + 0.965$	62	1.8392868
5	$1.524 \ln S + 0.078$	$0.863 \ln S + 0.921$	57	1.9275620
6	$1.479 \ln S - 0.185$	$0.795 \ln S + 0.864$	54	1.9659482
7	$1.460 \ln S - 0.424$	$0.762 \ln S + 0.797$	52	1.9835828
8	$1.451 \ln S - 0.642$	$0.744 \ln S + 0.723$	51	1.9919642
9	$1.447 \ln S - 0.838$	$0.734 \ln S + 0.646$	51	1.9960312
10	$1.445 \ln S - 1.017$	$0.728 \ln S + 0.568$	50	1.9980295
20	$1.443 \ln S - 2.170$	$0.721 \ln S - 0.030$	50	1.9999981

In Table 1, the "growth ratio" is $\lim_{n\to\infty} t_{n+1}/t_n$, the approximate factor by which the number of runs increases at each level. "Passes" denotes the average number of times each record is processed, namely $(1/S)$ times the total number

of initial runs processed during the distribution and merge phases. The stated number of passes and phases is correct in each case up to $O(S^{-\epsilon})$, for some $\epsilon > 0$, for perfect distributions as $S \to \infty$.

Figure 70 shows the average number of times each record is merged, as a function of S, when Algorithm D is used to handle the case of nonperfect numbers. Note that with three tapes there are "peaks" of relative inefficiency occurring just after the perfect distributions, but this phenomenon largely disappears when there are four or more tapes. The use of eight or more tapes gives comparatively little improvement over six or seven tapes.

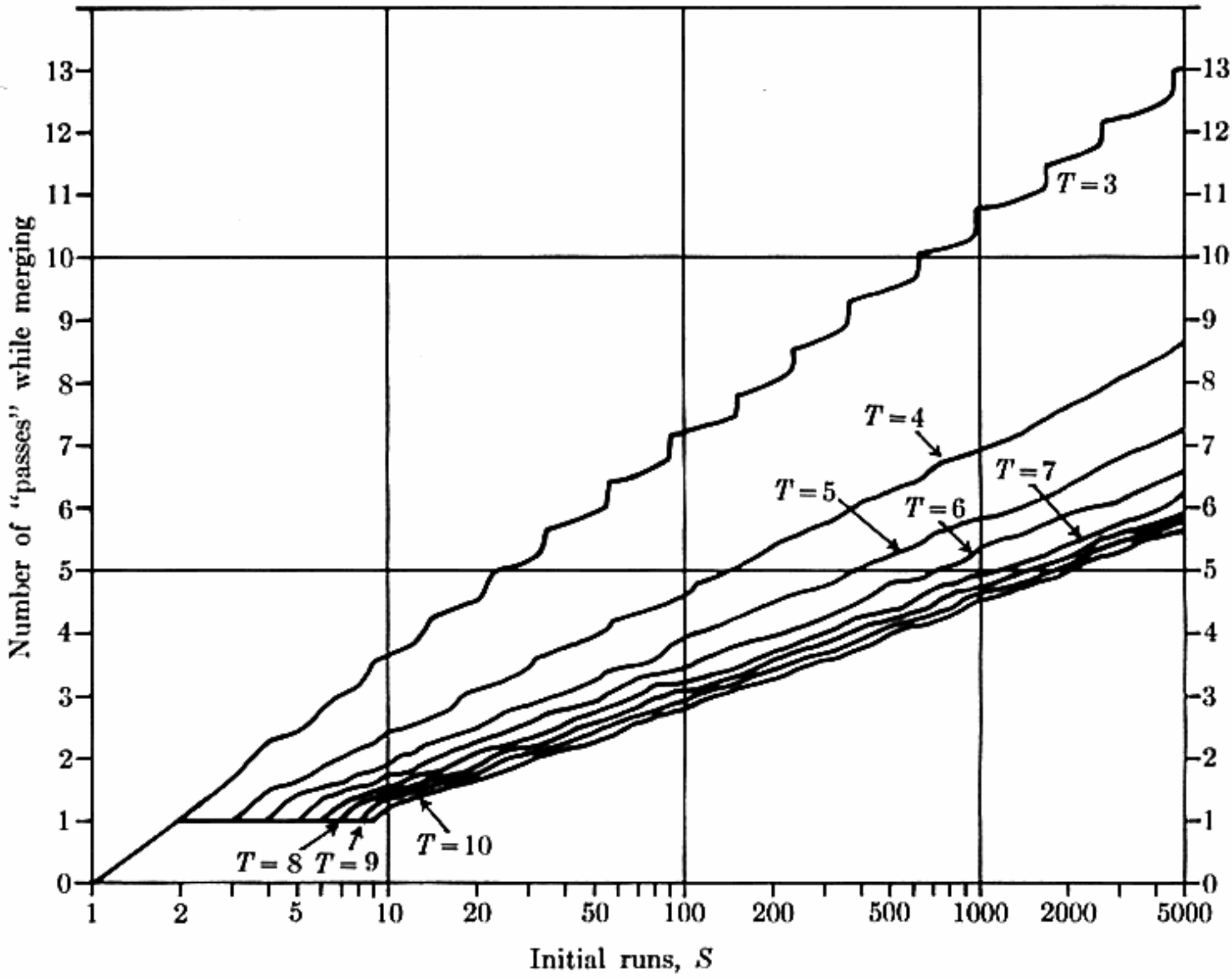


Fig. 70. Efficiency of polyphase merge using Algorithm D.

***A closer look.** In a balanced merge requiring k passes, every record is processed exactly k times during the course of the sort. But the polyphase procedure does not have this lack of bias; some records may get processed many more times than others, and we can gain speed if we arrange to put dummy runs into the oft-processed positions.

Let us therefore study the polyphase distribution more closely; instead of merely looking at the number of runs on each tape, as in (1), let us associate

with each run its *merge number*, the number of times it will be processed during the complete polyphase sort. We get the following table in place of (1):

Level	T1	T2	T3	T4	T5
0	0	—	—	—	—
1	1	1	1	1	1
2	21	21	21	21	2
3	3221	3221	3221	322	32
4	43323221	43323221	4332322	433232	4332
5	544343243323221	544343324332322	54434332433232	544343324332	54434332
. . . . .					
n	A_n	B_n	C_n	D_n	E_n
$n+1$	$(A_n+1)B_n$	$(A_n+1)C_n$	$(A_n+1)D_n$	$(A_n+1)E_n$	A_n+1
. . . . .					

(8)

Here A_n is a string of a_n values representing the merge numbers for each run on T1, if we begin with the level n distribution; B_n is the corresponding string for T2; etc. The notation "$(A_n + 1)B_n$" means, "A_n with all values increased by 1, followed by B_n."

Figure 71(a) shows A_5, B_5, C_5, D_5, E_5 tipped on end, showing how the merge numbers for each run appear on tape; note, for example, that the run at the beginning of each tape will be processed five times, while the run at the end of T1 will be processed only once. This discriminatory practice of the polyphase merge makes it much better to put a dummy run at the beginning of the tape than at the end. Figure 71(b) shows an optimum order in which to distribute runs for a five-level polyphase merge, placing each new run into a position with the smallest available merge number. Note that Algorithm D (Fig. 69) is not quite as good, since it fills some "4" positions before all of the "3" positions are used up.

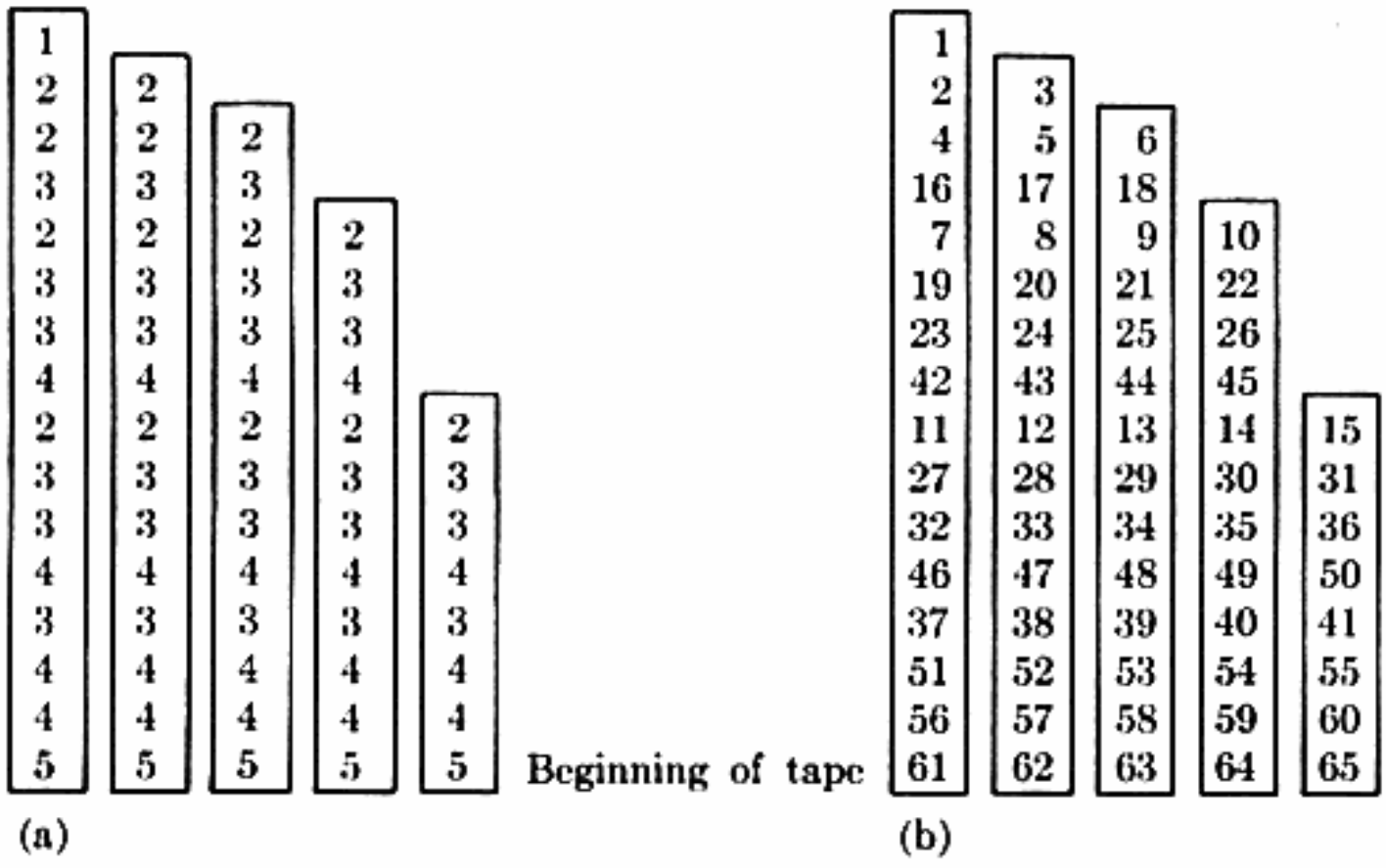


Fig. 71. Analysis of the fifth-level polyphase distribution for six tapes: (a) merge numbers, (b) optimum distribution order.

The recurrence relations (8) show that each of B_n, C_n, D_n, E_n are initial substrings of A_n. In fact, we can use (8) to derive the formulas

$$\begin{aligned} E_n &= (A_{n-1}) + 1, \\ D_n &= (A_{n-1}A_{n-2}) + 1, \\ C_n &= (A_{n-1}A_{n-2}A_{n-3}) + 1, \\ B_n &= (A_{n-1}A_{n-2}A_{n-3}A_{n-4}) + 1, \\ A_n &= (A_{n-1}A_{n-2}A_{n-3}A_{n-4}A_{n-5}) + 1, \end{aligned} \tag{9}$$

generalizing Eqs. (3), which treated only the lengths of these strings. Furthermore, the rule defining the A's implies that essentially the same structure is present at the beginning of every level; we have

$$A_n = n - Q_n, \tag{10}$$

where Q_n is a string of a_n values defined by the law

$$Q_n = Q_{n-1}(Q_{n-2}+1)(Q_{n-3}+2)(Q_{n-4}+3)(Q_{n-5}+4), \quad \text{for} \quad n \geq 1;$$
$$Q_0 = \text{'0'}; \qquad Q_n = (\text{empty}) \qquad \text{for} \qquad n < 0. \tag{11}$$

Since Q_n begins with Q_{n-1}, we can consider the *infinite* string Q_∞, whose first a_n elements are equal to Q_n; this string Q_∞ essentially characterizes all the merge numbers in the polyphase distribution. In the six-tape case,

$$Q_\infty = 01121223122323341223233423343441223233423343445233434453445451223 2\cdots. \tag{12}$$

Exercise 11 contains an interesting interpretation of this string.

Given that A_n is the string $m_1 m_2 \ldots m_{a_n}$, let $A_n(x) = x^{m_1} + x^{m_2} + \cdots + x^{m_{a_n}}$ be the corresponding generating function which counts the number of times each merge number appears; and define $B_n(x)$, $C_n(x)$, $D_n(x)$, $E_n(x)$ similarly. For example, $A_4(x) = x^4 + x^3 + x^3 + x^2 + x^3 + x^2 + x^2 + x = x^4 + 3x^3 + 3x^2 + x$. Relations (9) tell us that

$$\begin{aligned} E_n(x) &= x\big(A_{n-1}(x)\big), \\ D_n(x) &= x\big(A_{n-1}(x) + A_{n-2}(x)\big), \\ C_n(x) &= x\big(A_{n-1}(x) + A_{n-2}(x) + A_{n-3}(x)\big), \\ B_n(x) &= x\big(A_{n-1}(x) + A_{n-2}(x) + A_{n-3}(x) + A_{n-4}(x)\big), \\ A_n(x) &= x\big(A_{n-1}(x) + A_{n-2}(x) + A_{n-3}(x) + A_{n-4}(x) + A_{n-5}(x)\big), \end{aligned} \tag{13}$$

for $n \geq 1$, where $A_0(x) = 1$ and $A_n(x) = 0$ for $n = -1, -2, -3, -4$. Consequently,

$$\begin{aligned} \sum_{n\geq 0} A_n(x)z^n &= \frac{1}{1 - x(z + z^2 + z^3 + z^4 + z^5)} \\ &= \sum_{k\geq 0} x^k(z + z^2 + z^3 + z^4 + z^5)^k. \end{aligned} \tag{14}$$

Considering the runs on all tapes, we let

$$T_n(x) = A_n(x) + B_n(x) + C_n(x) + D_n(x) + E_n(x), \qquad n \geq 1; \tag{15}$$

from (13) we immediately have

$$T_n(x) = 5A_{n-1}(x) + 4A_{n-2}(x) + 3A_{n-3}(x) + 2A_{n-4}(x) + A_{n-5}(x),$$

hence

$$\sum_{n\geq 1} T_n(x)z^n = \frac{x(5z + 4z^2 + 3z^3 + 2z^4 + z^5)}{1 - x(z + z^2 + z^3 + z^4 + z^5)}, \tag{16}$$

The form of (16) shows that it is easy to compute the coefficients of $T_n(x)$:

	z	z^2	z^3	z^4	z^5	z^6	z^7	z^8	z^9	z^{10}	z^{11}	z^{12}	z^{13}	z^{14}	
x	5	4	3	2	1	0	0	0	0	0	0	0	0	0	
x^2	0	5	9	12	14	15	10	6	3	1	0	0	0	0	
x^3	0	0	5	14	26	40	55	60	57	48	35	20	10	4	(17)
x^4	0	0	0	5	19	45	85	140	195	238	260	255	220	170	
x^5	0	0	0	0	5	24	69	154	294	484	703	918	1088	1168	

The columns of this tableau give $T_n(x)$; for example, $T_4(x) = 2x + 12x^2 + 14x^3 + 5x^4$. After the first row, each entry in the tableau is the sum of the five entries just above and to the left in the previous row.

The number of runs in a "perfect" nth level distribution is $T_n(1)$, and the total amount of processing as these runs are merged is the derivative, $T_n'(1)$. Now

$$\sum_{n\geq 1} T_n'(x)z^n = \frac{5z + 4z^2 + 3z^3 + 2z^4 + z^5}{(1 - x(z + z^2 + z^3 + z^4 + z^5))^2}; \tag{18}$$

setting $x = 1$ in (16) and (18) gives a result in agreement with our earlier demonstration that the merge processing for a perfect nth level distribution is the coefficient of z^n in $a(z)t(z)$ [cf. (7)].

We can use the functions $T_n(x)$ to determine the work involved when dummy runs are added in an optimum way. Let $\Sigma_n(m)$ be the sum of the smallest m merge numbers in an nth level distribution. These values are readily calculated by looking at the columns of (17), and we find that $\Sigma_n(m) =$

$m =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$n = 1$	1	2	3	4	5	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
$n = 2$	1	2	3	4	6	8	10	12	14	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
$n = 3$	1	2	3	5	7	9	11	13	15	17	19	21	24	27	30	33	36	∞	∞	∞	∞
$n = 4$	1	2	4	6	8	10	12	14	16	18	20	22	24	26	29	32	35	38	41	44	47
$n = 5$	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	32	35	38	41	44	47
$n = 6$	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	33	36	39	42	45	48
$n = 7$	2	4	6	8	10	12	14	16	18	20	23	26	29	32	35	38	41	44	47	50	53

(19)

For example, if we wish to sort 17 runs using a level-3 distribution, the total amount of processing is $\Sigma_3(17) = 36$; but if we use a level-4 or level-5 distribution and position the dummy runs optimally, the total amount of processing during the merge phases is only $\Sigma_4(17) = \Sigma_5(17) = 35$. It is better to use level 4, even though 17 corresponds to a "perfect" level-3 distribution! Indeed, as S gets large it turns out that the optimum number of levels is many more than that used in Algorithm D.

Exercise 14 shows that there is a nondecreasing sequence of numbers M_n such that level n is "optimum" for $M_n \le S < M_{n+1}$, but not for $S \ge M_{n+1}$. In the six-tape case the table of $\Sigma_n(m)$ we have just calculated shows that

$$M_0 = 0, \qquad M_1 = 2, \qquad M_2 = 6, \qquad M_3 = 10, \qquad M_4 = 14.$$

The above discussion treats only the case of six tapes, but it is clear that the same ideas apply to polyphase merging with T tapes for any $T \ge 3$; we simply replace 5 by $P = T - 1$ in all appropriate places. Table 2 shows the sequences M_n obtained for various values of T. Table 3 and Fig. 72 indicate the total number of initial runs that are processed after making an optimum

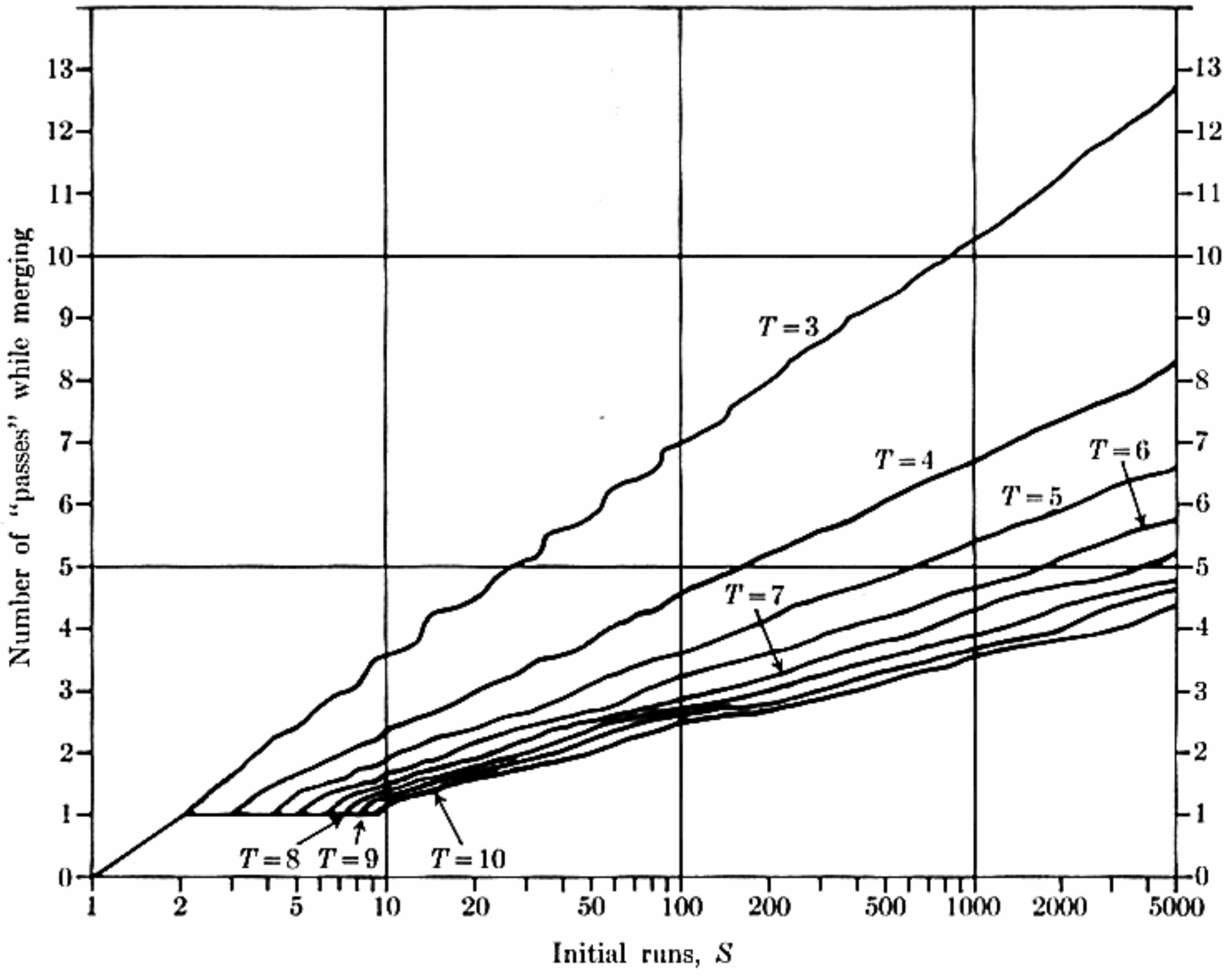


Fig. 72. Efficiency of polyphase merge with optimum initial distribution (cf. Fig. 70).

distribution of dummy runs. (Formulas which appear at the bottom of Table 3 should be taken with a grain of salt, since they are "least-squares fits" over the range $1 \leq S \leq 5000$ ($1 \leq S \leq 10000$ for $T = 3$); this leads to somewhat erratic behavior because the given range of S values is not equally favorable for all T. As $S \to \infty$, the number of initial runs processed after an optimum polyphase distribution is asymptotically $S \log_P S$, but convergence to this asymptotic limit is extremely slow.)

Table 2

NUMBER OF RUNS FOR WHICH A GIVEN LEVEL IS OPTIMUM

Level	$T = 3$	$T = 4$	$T = 5$	$T = 6$	$T = 7$	$T = 8$	$T = 9$	$T = 10$	
1	2	2	2	2	2	2	2	2	M_1
2	3	4	5	6	7	8	9	10	M_2
3	4	6	8	10	12	14	16	18	M_3
4	6	10	14	14	17	20	23	26	M_4
5	9	18	23	29	20	24	28	32	M_5
6	14	32	35	43	53	27	32	37	M_6
7	22	55	76	61	73	88	35	41	M_7
8	35	96	109	154	98	115	136	44	M_8
9	56	173	244	216	283	148	171	199	M_9
10	90	280	359	269	386	168	213	243	M_{10}
11	145	535	456	779	481	640	240	295	M_{11}
12	234	820	1197	1034	555	792	1002	330	M_{12}
13	378	1635	1563	1249	1996	922	1228	1499	M_{13}
14	611	2401	4034	3910	2486	1017	1432	1818	M_{14}
15	988	4959	5379	4970	2901	4397	1598	2116	M_{15}
16	1598	7029	6456	5841	10578	5251	1713	2374	M_{16}
17	2574	14953	18561	19409	13097	5979	8683	2576	M_{17}
18	3955	20583	22876	23918	15336	6499	10069	2709	M_{18}
19	6528	44899	64189	27557	17029	30164	11259	15787	M_{19}

Table 3

INITIAL RUNS PROCESSED DURING OPTIMUM POLYPHASE MERGE

S	$T = 3$	$T = 4$	$T = 5$	$T = 6$	$T = 7$	$T = 8$	$T = 9$	$T = 10$
10	36	24	19	17	15	14	13	12
20	90	60	49	44	38	36	34	33
50	294	194	158	135	128	121	113	104
100	702	454	362	325	285	271	263	254
1000	10371	6680	5430	4672	4347	3872	3739	3632
5000	63578	41286	32905	28620	26426	23880	23114	22073
S	$(1.51$	0.951	0.761	0.656	0.589	0.548	0.539	$0.488) \cdot S \ln S$
	$+ (- .11$	$+ .14$	$+ .16$	$+ .19$	$+ .21$	$+ .20$	$+ .02$	$+ .18) \cdot S$

Table 4 shows how the distribution method of Algorithm D compares with the results of optimum distribution in Table 3. It is clear that Algorithm D is not very close to the optimum when S and T become large; but it is not clear how to do much better than Algorithm D without considerable complication in such cases, especially if we do not know S in advance. Fortunately, we rarely have to worry about large S (see Section 5.4.6), so Algorithm D is not too bad in practice; in fact, it's pretty good.

Table 4

INITIAL RUNS PROCESSED DURING STANDARD POLYPHASE MERGE

S	$T = 3$	$T = 4$	$T = 5$	$T = 6$	$T = 7$	$T = 8$	$T = 9$	$T = 10$
10	36	24	19	17	15	14	13	12
20	90	62	49	44	41	37	34	33
50	294	194	167	143	134	131	120	114
100	714	459	393	339	319	312	292	277
1000	10730	6920	5774	5370	4913	4716	4597	4552
5000	64740	43210	36497	32781	31442	29533	28817	28080

Polyphase sorting was first analyzed mathematically by W. C. Carter [*Proc. IFIP Congress* (1962), 62–66]. Many of the above results about optimal dummy run placement are due originally to B. Sackman and T. Singer ["A vector model for merge sort analysis," unpublished paper presented at the ACM Sort Symposium (November, 1962), 21 pp.]. Sackman later suggested the horizontal method of distribution used in Algorithm D. Donald Shell [*CACM* **14** (1971), 713–719; **15** (1972), 28] developed the theory independently, noted relation (10), and made a detailed study of several different distribution algorithms. Further instructive developments and refinements have been made by Derek A. Zave [*JACM* (to appear)]. The generating function (16) was first investigated by W. Burge [*Proc. IFIP Congress* (1971), to appear].

But what about rewind time? So far we have taken "initial runs processed" as the sole measure of efficiency for comparing tape merge strategies. But after each of phases 2 through 6, in the examples at the beginning of this section, it is necessary for the computer to wait for two tapes to rewind; both the previous output tape and the new current output tape must be repositioned at the beginning, before the next phase can proceed. This can cause a significant delay, since the previous output tape generally contains a significant percentage of the records being sorted (see the "Pass/Phase" column in Table 1). It is a shame to have the computer twiddling its thumbs during all these rewind operations, since useful work could be done with the other tapes if we used a different merging pattern.

A simple modification of the polyphase procedure will overcome this problem, although it requires at least five tapes [see Y. Cesari, thesis, U. of Paris (1968), 25–27, where the idea is credited to J. Caron]. Each phase in Caron's scheme merges runs from $T - 3$ tapes onto another tape, while the remaining two tapes are rewinding.

For example, consider the case of six tapes and 49 initial runs. In the following tableau, R denotes rewinding during the phase, and T5 is assumed to contain the original input:

Phase	T1	T2	T3	T4	T5	T6	Write time	Rewind time
1	1^{11}	1^{17}	1^{13}	1^{8}	—	(R)	49	17
2	(R)	1^{9}	1^{5}	—	R	3^{8}	$8 \times 3 = 24$	$49 - 17 = 32$
3	1^{6}	1^{4}	—	R	3^{5}	R	$5 \times 3 = 15$	max (8, 24)
4	1^{2}	—	R	5^{4}	R	3^{4}	$4 \times 5 = 20$	max (13, 15)
5	—	R	7^{2}	R	3^{3}	3^{2}	$2 \times 7 = 14$	max (17, 20)
6	R	11^{2}	R	5^{2}	3^{1}	—	$2 \times 11 = 22$	max (11, 14)
7	15^{1}	R	7^{1}	5^{1}	—	R	$1 \times 15 = 15$	max (22, 24)
8	R	11^{1}	7^{0}	—	R	23^{1}	$1 \times 23 = 23$	max (15, 15)
9	15^{1}	11^{1}	—	R	33^{0}	R	$0 \times 33 = 0$	max (20, 23)
10	(15^{0})	—	R	49^{1}	(R)	(23^{0})	$1 \times 49 = 49$	14

Here all the rewind time is essentially overlapped, except in phase 9 (a "dummy phase" which prepares for the final merge), and after the initial distribution phase (when all tapes are rewound). If t is the time to merge the number of records in one initial run, and if r is the time to rewind over one initial run, this process takes about $182t + 40r$ plus the time for initial distribution and final rewind. The corresponding figures for standard polyphase using Algorithm D are $140t + 104r$, which is slightly worse when $r = \frac{3}{4}t$, slightly better when $r = \frac{1}{2}t$.

Everything we have said about standard polyphase can be adapted to Caron's polyphase; for example, the sequence a_n now satisfies the recurrence

$$a_n = a_{n-2} + a_{n-3} + a_{n-4} \tag{20}$$

instead of (3). The reader will find it instructive to analyze this method in the same way we analyzed standard polyphase, since it will enhance an understanding of both methods. (See, for example, exercises 19 and 20.)

Table 5 gives facts about Polyphase Caron which are analogous to the facts about Polyphase Ordinaire in Table 1. Note that Caron's method actually becomes *superior* to polyphase on eight or more tapes, in the number of runs processed as well as in the rewind time, even though it does $(T - 3)$-way merging instead of $(T - 1)$-way merging!

Table 5

APPROXIMATE BEHAVIOR OF CARON'S POLYPHASE MERGE SORT

Tapes	Phases	Passes	Pass/phase, percent	Growth ratio
5	$3.556 \ln S + 0.158$	$1.463 \ln S + 1.016$	41	1.3247180
6	$2.616 \ln S - 0.166$	$0.951 \ln S + 1.014$	36	1.4655712
7	$2.337 \ln S - 0.472$	$0.781 \ln S + 1.001$	33	1.5341577
8	$2.216 \ln S - 0.762$	$0.699 \ln S + 0.980$	32	1.5701473
9	$2.156 \ln S - 1.034$	$0.654 \ln S + 0.954$	30	1.5900054
10	$2.124 \ln S - 1.290$	$0.626 \ln S + 0.922$	29	1.6013473
20	$2.078 \ln S - 3.093$	$0.575 \ln S + 0.524$	28	1.6179086

This may seem paradoxical until we realize that *a high order of merge does not necessarily imply an efficient sort.* As an extreme example, consider placing 1 run on T1 and n runs on T2, T3, T4, T5; if we alternately do five-way merging to T6 and T1 until T2, T3, T4, T5 are empty, the processing time is $(2n^2 + 3n)$ initial run lengths, essentially proportional to S^2 instead of $S \log S$, although five-way merging was done throughout.

Tape splitting. Efficient overlapping of rewind time is a problem which arises in many applications, not just sorting, and there is a general approach which can often be used. Consider an iterative process which uses two tapes in the following way:

	T1	T2
Phase 1	Output 1	—
	Rewind	—
Phase 2	Input 1	Output 2
	Rewind	Rewind
Phase 3	Output 3	Input 2
	Rewind	Rewind
Phase 4	Input 3	Output 4
	Rewind	Rewind

and so on, where "Output k" means write the kth output file and "Input k" means read it. The rewind time can be avoided when three tapes are used, as suggested by C. Weisert [*CACM* **5** (1962), 102]:

	T1	T2	T3
Phase 1	Output 1.1	—	—
	Output 1.2	—	—
	Rewind	Output 1.3	—
Phase 2	Input 1.1	Output 2.1	—
	Input 1.2	Rewind	Output 2.2
	Rewind	Input 1.3	Output 2.3
Phase 3	Output 3.1	Input 2.1	Rewind
	Output 3.2	Rewind	Input 2.2
	Rewind	Output 3.3	Input 2.3
Phase 4	Input 3.1	Output 4.1	Rewind
	Input 3.2	Rewind	Output 4.2
	Rewind	Input 3.3	Output 4.3

and so on. Here "Output $k.j$" means, write the jth third of the kth output file, and "Input $k.j$" means, read it. Virtually all of the rewind time will be eliminated if rewinding is at least twice as fast as the read/write speed. Such a procedure, in which the output of each phase is divided between tapes, is called "tape-splitting."

R. L. McAllester [*CACM* 7 (1964), 158–159] has shown that tape-splitting leads to an efficient way of overlapping the rewind time in a polyphase merge. His method can be used with four or more tapes, and it does $(T - 2)$-way merging.

Assuming once again that we have six tapes, let us try to design a merge pattern that operates as follows, splitting the output on each level, where "I", "O", and "R," respectively, denote input, output, and rewinding:

Level	T1	T2	T3	T4	T5	T6	Number of runs output	
7	I	I	I	I	R	O	u_7	
	I	I	I	I	O	R	v_7	
6	I	I	I	R	O	I	u_6	
	I	I	I	O	R	I	v_6	
5	I	I	R	O	I	I	u_5	
	I	I	O	R	I	I	v_5	
4	I	R	O	I	I	I	u_4	
	I	O	R	I	I	I	v_4	
3	R	O	I	I	I	I	u_3	
	O	R	I	I	I	I	v_3	
2	O	I	I	I	I	R	u_2	
	R	I	I	I	I	O	v_2	
1	I	I	I	I	R	O	u_1	
	I	I	I	I	O	R	v_1	
0	I	I	I	R	O	I	u_0	
	I	I	I	O	R	I	v_0	(21)

In order to end with one run on T4 and all other tapes empty, we need to have

$$\begin{aligned}
v_0 &= 1,\\
u_0 + v_1 &= 0,\\
u_1 + v_2 &= u_0 + v_0,\\
u_2 + v_3 &= u_1 + v_1 + u_0 + v_0,\\
u_3 + v_4 &= u_2 + v_2 + u_1 + v_1 + u_0 + v_0,\\
u_4 + v_5 &= u_3 + v_3 + u_2 + v_2 + u_1 + v_1 + u_0 + v_0,\\
u_5 + v_6 &= u_4 + v_4 + u_3 + v_3 + u_2 + v_2 + u_1 + v_1,
\end{aligned}$$

etc.; in general, the requirement is that

$$u_n + v_{n+1} = u_{n-1} + v_{n-1} + u_{n-2} + v_{n-2} + u_{n-3} + v_{n-3} + u_{n-4} + v_{n-4} \qquad (22)$$

for all $n \geq 0$, if we regard $u_j = v_j = 0$ for all $j < 0$.

There is no unique solution to these equations; indeed, if we let all the u's be zero, we get the usual polyphase merge with one tape wasted! But if we choose $u_n \approx v_{n+1}$, the rewind time will be satisfactorily overlapped.

McAllester suggested taking $u_n = v_{n-1} + v_{n-2} + v_{n-3} + v_{n-4}$, $v_{n+1} = u_{n-1} + u_{n-2} + u_{n-3} + u_{n-4}$, so that the sequence

$$\langle x_0, x_1, x_2, x_3, x_4, x_5, \ldots \rangle = \langle v_0, u_0, v_1, u_1, v_2, u_2, \ldots \rangle$$

satisfies the uniform recurrence $x_n = x_{n-3} + x_{n-5} + x_{n-7} + x_{n-9}$. However, it turns out to be better to let

$$\begin{aligned}
v_{n+1} &= u_{n-1} + v_{n-1} + u_{n-2} + v_{n-2},\\
u_n &= u_{n-3} + v_{n-3} + u_{n-4} + v_{n-4};
\end{aligned} \qquad (23)$$

this sequence is not only slightly better in its merging time, it also has the great virtue that the corresponding merging time can be analyzed mathematically. [McAllester's choice is extremely difficult to analyze because runs of different lengths may occur during a single phase; we shall see that this does not happen with (23).]

We can deduce the number of runs on each tape on each level, by working backwards in the pattern (21), and we obtain the following sorting scheme:

Level	T1	T2	T3	T4	T5	T6	Write time	Rewind time
	1^{23}	1^{21}	1^{17}	1^{10}	—	1^{11}	82	23
7	1^{19}	1^{17}	1^{13}	1^{6}	R	$1^{11}4^{4}$	$4 \times 4 = 16$	82 – 23
	1^{13}	1^{11}	1^{7}	—	4^{6}	R	$6 \times 4 = 24$	25
6	1^{10}	1^{8}	1^{4}	R	4^{9}	$1^{8}4^{4}$	$3 \times 4 = 12$	10
	1^{6}	1^{4}	—	4^{4}	R	$1^{4}4^{4}$	$4 \times 4 = 16$	36
5	1^{5}	1^{3}	R	$4^{4}7^{1}$	4^{8}	$1^{3}4^{4}$	$1 \times 7 = 7$	17
	1^{2}	—	7^{3}	R	4^{5}	4^{4}	$3 \times 7 = 21$	23
4	1^{1}	R	$7^{3}13^{1}$	$4^{3}7^{1}$	4^{4}	4^{3}	$1 \times 13 = 13$	21
	—	13^{1}	R	$4^{2}7^{1}$	4^{3}	4^{2}	$1 \times 13 = 13$	34
3	R	$13^{1}19^{1}$	$7^{2}13^{1}$	$4^{1}7^{1}$	4^{2}	4^{1}	$1 \times 19 = 19$	23
	19^{1}	R	$7^{1}13^{1}$	7^{1}	4^{1}	—	$1 \times 19 = 19$	32
2	$19^{1}31^{0}$	$13^{1}19^{1}$	$7^{1}13^{1}$	7^{1}	4^{1}	R	$0 \times 31 = 0$	25
	R	19^{1}	13^{1}	7^{0}	—	31^{1}	$1 \times 31 = 31$	19
1	$19^{1}31^{0}$	19^{1}	13^{1}	7^{0}	R	$31^{1}52^{0}$	$0 \times 52 = 0$	
	$19^{1}31^{0}$	19^{1}	13^{1}	—	52^{0}	R	$0 \times 52 = 0$	max (36, 31, 23)
0	$19^{1}31^{0}$	19^{1}	13^{1}	R	$52^{0}82^{0}$	$31^{1}52^{0}$	$0 \times 82 = 0$	
	(31^{0})	(19^{0})	—	82^{1}	(R)	(52^{0})	$1 \times 82 = 82$	0

Unoverlapped rewinding occurs when the input tape T5 is being rewound (82 units), during the first half of the level 2 phase (25 units), and during the final "dummy merge" phases in levels 1 and 0 (36 units). So we may estimate the time as $273t + 143r$; the corresponding amount for Algorithm D, $268t + 208r$, is almost always inferior.

It is not difficult to see (cf. exercise 23) that the run lengths output during each phase are successively

$$4, 4, 7, 13, 19, 31, 52, 82, 133, \ldots \tag{24}$$

a sequence $\langle t_1, t_2, t_3, \ldots \rangle$ satisfying the law

$$t_n = t_{n-2} + 2t_{n-3} + t_{n-4} \tag{25}$$

if we regard $t_n = 1$ for $n \leq 0$. We can also analyze the optimum placement of dummy runs, by looking at strings of merge numbers as we did for standard polyphase [cf. (8)]:

Level	T1	T2	T3	T4	T6	Final output on
1	1	1	1	1	—	T5
2	1	1	1	—	1	T4
3	21	21	2	2	1	T3
4	2221	222	222	22	2	T2
5	23222	23222	2322	23	222	T1
6	33332322	33332322	333323	3333	2322	T6
. . . . .						
n	A_n	B_n	C_n	D_n	E_n	T(k)
$n+1$	$(A_n''E_n + 1)B_n$	$(A_n''E_n + 1)C_n$	$(A_n''E_n + 1)D_n$	$A_n''E_n + 1$	A_n'	T($k-1$)
. . . . .						

(26)

where $A_n = A'_n A''_n$, and A''_n consists of the last u_n merge numbers of A_n. The above rule for going from level n to level $n+1$ is valid for *any* scheme satisfying (22). When we define the u's and v's by (23), the strings $A_n, \ldots, E_n$ can be expressed in the following rather simple way [cf. (9)]:

$$\begin{aligned} A_n &= (W_{n-1}W_{n-2}W_{n-3}W_{n-4}) + 1, \\ B_n &= (W_{n-1}W_{n-2}W_{n-3}) + 1, \\ C_n &= (W_{n-1}W_{n-2}) + 1, \\ D_n &= (W_{n-1}) + 1, \\ E_n &= (W_{n-2}W_{n-3}) + 1, \end{aligned} \tag{27}$$

where

$$\begin{aligned} &W_n = (W_{n-3}W_{n-4}W_{n-2}W_{n-3}) + 1 \quad \text{for} \quad n > 0, \\ &W_0 = \text{'0'}, \quad \text{and} \quad W_n = (\text{empty}) \quad \text{for} \quad n < 0. \end{aligned} \tag{28}$$

From these relations it is easy to make a detailed analysis of the six-tape case.

In general, when there are $T \geq 5$ tapes, we let $P = T - 2$, and we define the sequences $\langle u_n \rangle$, $\langle v_n \rangle$ by the rules

$$\begin{aligned} v_{n+1} &= u_{n-1} + v_{n-1} + \cdots + u_{n-r} + v_{n-r}, \\ u_n &= u_{n-r-1} + v_{n-r-1} + \cdots + u_{n-P} + v_{n-P}, \quad \text{for} \quad n \geq 0, \end{aligned} \tag{29}$$

where $r = \lfloor P/2 \rfloor$; $v_0 = 1$, and $u_n = v_n = 0$ for $n < 0$. If $w_n = u_n + v_n$, we therefore have

$$w_n = w_{n-2} + \cdots + w_{n-r} + 2w_{n-r-1} + w_{n-r-2} + \cdots + w_{n-P}, \quad n > 0; \tag{30}$$

$w_0 = 1$; and $w_n = 0$ for $n < 0$. The initial distribution on tapes for level $n+1$ places $w_n + w_{n-1} + \cdots + w_{n-P+k}$ runs on tape k, for $1 \leq k \leq P$, and $w_{n-1} + \cdots + w_{n-r}$ on tape T; tape $T-1$ is used for input. Then u_n runs are merged to tape T while $T-1$ is being rewound; v_n are merged to $T-1$ while T is rewinding; u_{n-1} to $T-1$ while $T-2$ is rewinding; etc.

Table 6 shows the approximate behavior of this procedure when S is not too small. The "Pass/Phase" column indicates approximately how much of the entire file is being rewound during each half of a phase, and approximately how much of the file is being written during each full phase. *The tape-splitting method is superior to standard polyphase on six or more tapes*, and probably also on five, at least for large S.

When $T = 4$ the above procedure would become essentially equivalent to balanced two-way merging, *without* overlapping the rewind time, since w_{2n+1} would be 0 for all n. So the entries in Table 6 for $T = 4$ have been obtained by making a slight modification, letting $v_2 = 0$, $u_1 = 1$, $v_1 = 0$, $u_0 = 0$, $v_0 = 1$, and $v_{n+1} = u_{n-1} + v_{n-1}$, $u_n = u_{n-2} + v_{n-2}$ for $n \geq 2$. This leads to a very interesting sorting scheme (see exercises 25 and 26).

Table 6

APPROXIMATE BEHAVIOR OF POLYPHASE MERGE WITH TAPE SPLITTING

Tapes	Phases	Passes	Pass/phase, percent	Growth ratio
4	$2.885 \ln S + 0.000$	$1.443 \ln S + 1.000$	50	1.4142136
5	$2.078 \ln S + 0.232$	$0.929 \ln S + 1.022$	45	1.6180340
6	$2.078 \ln S - 0.170$	$0.752 \ln S + 1.024$	36	1.6180340
7	$1.958 \ln S - 0.408$	$0.670 \ln S + 1.007$	34	1.6663019
8	$2.008 \ln S - 0.762$	$0.624 \ln S + 0.994$	31	1.6454116
9	$1.972 \ln S - 0.987$	$0.595 \ln S + 0.967$	30	1.6604077
10	$2.013 \ln S - 1.300$	$0.580 \ln S + 0.941$	29	1.6433803
20	$2.069 \ln S - 3.164$	$0.566 \ln S + 0.536$	27	1.6214947

EXERCISES

1. [*16*] Figure 69 shows the order in which runs 34 through 65 are distributed to five tapes with Algorithm D; in what order are runs 1 through 33 distributed?

▸ **2.** [*21*] True or false: After two merge phases in Algorithm D, i.e., on the second time we reach step D6, all dummy runs have disappeared.

▸ **3.** [*22*] Prove that the condition $\texttt{D}[1] \geq \texttt{D}[2] \geq \cdots \geq \texttt{D}[T]$ is always satisfied at the conclusion of step D4. Explain why this condition is important, in the sense that the mechanism of steps D2 and D3 would not work properly otherwise.

4. [*M20*] Derive the generating functions (7).

5. [*HM26*] (E. P. Miles, Jr., 1960.) For all $p \geq 2$, prove that the polynomial $f_p(z) = z^p - z^{p-1} - \cdots - z - 1$ has p distinct roots, of which exactly one has magnitude greater than unity. [*Hint:* Consider the polynomial $z^{p+1} - 2z^p + 1$.]

6. [*HM24*] The purpose of this exercise is to consider how Tables 1, 5, and 6 can be prepared. Assume that we have a merging pattern whose properties are characterized by polynomials $p(z)$ and $q(z)$ in the following way: (1) The number of initial runs present in a "perfect distribution" requiring n merging phases is the coefficient of z^n in $p(z)/q(z)$. (2) The number of initial runs processed during these n merging phases is the coefficient of z^n in $p(z)/q(z)^2$. (3) There is a "dominant root" α of $q(z^{-1})$ such that $q(\alpha^{-1}) = 0$, $q'(\alpha^{-1}) \neq 0$, $p(\alpha^{-1}) \neq 0$, and $q(\beta^{-1}) = 0$ implies that $\beta = \alpha$ or $|\beta| < |\alpha|$.

Prove that there is a number $\epsilon > 0$ such that, if S is the number of runs in a perfect distribution requiring n merging phases, and if ρS initial runs are processed during those phases, we have $n = a \ln S + b + O(S^{-\epsilon})$, $\rho = c \ln S + d + O(S^{-\epsilon})$, where

$$a = (\ln \alpha)^{-1}, \qquad b = -a \ln \left(\frac{p(\alpha^{-1})}{-q'(\alpha^{-1})} \right) - 1, \qquad c = a \frac{\alpha}{-q'(\alpha^{-1})},$$

$$d = \frac{(b+1)\alpha - p'(\alpha^{-1})/p(\alpha^{-1}) + q''(\alpha^{-1})/q'(\alpha^{-1})}{-q'(\alpha^{-1})}.$$

7. [*HM22*] Let α_p be the dominant root of the polynomial $f_p(z)$ in exercise 5. What is the asymptotic behavior of α_p as $p \longrightarrow \infty$?

8. [*M20*] (E. Netto, 1901.) Let $N_m^{(p)}$ be the number of ways to express m as an ordered sum of the integers $\{1, 2, \ldots, p\}$. For example, when $p = 3$ and $m = 5$, there are 13 ways, namely $1+1+1+1+1 = 1+1+1+2 = 1+1+2+1 = 1+1+3 = 1+2+1+1 = 1+2+2 = 1+3+1 = 2+1+1+1 = 2+1+2 = 2+2+1 = 2+3 = 3+1+1 = 3+2$. Show that $N_m^{(p)}$ is a generalized Fibonacci number.

9. [*M20*] Let $K_m^{(p)}$ be the number of sequences of m 0's and 1's such that there are no p consecutive 1's. For example, when $p = 3$ and $m = 5$ there are 24 ways: 00000, 00001, 00010, 00011, 00100, 00101, 00110, 01000, 01001, ..., 11011. Show that $K_m^{(p)}$ is a generalized Fibonacci number.

10. [*M27*] (*Generalized Fibonacci number system.*) Prove that every nonnegative integer n has a unique representation as a sum of distinct pth order Fibonacci numbers $F_j^{(p)}$, for $j \geq p$, subject to the condition that no p consecutive Fibonacci numbers are used.

11. [*M24*] Prove that the nth element of the string Q_∞ in (12) is equal to the number of distinct Fibonacci numbers in the fifth-order Fibonacci representation of $n - 1$. (Cf. exercise 10.)

►**12.** [*M20*] Find a connection between powers of the matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

and the perfect Fibonacci distributions in (1).

►**13.** [*22*] Prove the following rather odd property of perfect Fibonacci distributions: When the final output will be on tape number T, the number of runs on each other tape is *odd;* when the final output will be on some tape other than T, the number of runs will be *odd* on that tape, and it will be *even* on the others. [Cf. (1).]

14. [*M35*] Let $T_n(x) = \sum_{k\geq 0} T_{nk}x^k$, where $T_n(x)$ is the polynomial defined in (16). (a) Show that for each k there is a number $n(k)$ such that $T_{1k} \leq T_{2k} \leq \cdots \leq T_{n(k)k} > T_{n(k)+1,k} \geq \cdots$. (b) Given that $T_{n'k'} < T_{nk'}$ and $n' < n$, prove that $T_{n'k} \leq T_{nk}$ for all $k \geq k'$. (c) Prove that there is a nondecreasing sequence $\langle M_n \rangle$ such that $\Sigma_n(S) = \min_{j\geq 1} \Sigma_j(S)$ when $M_n \leq S < M_{n+1}$, but $\Sigma_n(S) > \min_{j\geq 1} \Sigma_j(S)$ when $S \geq M_{n+1}$. [Cf. (19).]

15. [*M43*] Prove or disprove: $\Sigma_{n-1}(m) < \Sigma_n(m)$ implies that $\Sigma_n(m) \leq \Sigma_{n+1}(m) \leq \Sigma_{n+2}(m) \leq \cdots$. [Such a result would greatly simplify the calculation of Table 2.]

16. [*M43*] Determine the asymptotic behavior of the polyphase merge with optimum distribution of dummy runs.

17. [*32*] Prove or disprove: There is a way to disperse runs for an optimum polyphase distribution in such a way that the distribution for $S+1$ initial runs is formed by adding one run (on an appropriate tape) to the distribution for S initial runs.

18. [*30*] Does the optimum polyphase distribution produce the best possible merging pattern, in the sense that the total number of initial runs processed is minimized, if we insist that the initial runs be placed on at most $T-1$ of the tapes? (Ignore rewind time.)

19. [*21*] Make a table analogous to (1), for Caron's polyphase sort on six tapes.

20. [*M24*] What generating functions for Caron's polyphase sort on six tapes correspond to (7) and to (16)? What relations, analogous to (9) and (27), define the strings of merge numbers?

21. [*11*] What should appear on level 7 in (26)?

22. [*M21*] Each term of the sequence (24) is approximately equal to the sum of the previous two. Does this phenomenon hold for the remaining numbers of the sequence? Formulate and prove a theorem about $t_n - t_{n-1} - t_{n-2}$.

▸**23.** [*29*] What changes would be made to (25), (27), and (28), if (23) were changed to $v_{n+1} = u_{n-1} + v_{n-1} + u_{n-2}$, $u_n = v_{n-2} + u_{n-3} + v_{n-3} + u_{n-4} + v_{n-4}$?

24. [*HM41*] Compute the asymptotic behavior of the tape-splitting polyphase procedure, when v_{n+1} is defined to be the sum of the first q terms of $u_{n-1} + v_{n-1} + \cdots + u_{n-P} + v_{n-P}$, for various $P = T-2$ and for $0 \le q \le 2P$. (The text treats only the case $q = 2\lfloor P/2 \rfloor$; cf. exercise 23.)

25. [*19*] Show how the tape-splitting polyphase merge on four tapes, mentioned at the end of this section, would sort 32 initial runs. (Give a phase-by-phase analysis like the 82-run six-tape example in the text.)

26. [*M21*] Analyze the behavior of the tape-splitting polyphase merge on four tapes, when $S = 2^n$ and when $S = 2^n + 2^{n-1}$. (See exercise 25.)

27. [*28*] Once the initial runs have been distributed to tapes in a perfect distribution, the polyphase strategy is simply to "merge until empty." We merge runs from all nonempty input tapes until one of them becomes empty; then we use that tape as the next output tape, and let the previous output tape serve as an input.

Does this merge-until-empty strategy always sort, no matter how the initial runs are distributed, as long as we distribute them onto at least two tapes? (One tape will, of course, be left empty so that it can be the first output tape.)

28. [*M26*] The previous exercise defines a rather large family of merging patterns. Show that polyphase is the *best* of these, in the following sense: If there are six tapes, and if we consider the class of all initial distributions (a, b, c, d, e) such that the merge-until-empty strategy requires n or less phases to sort, then $a + b + c + d + e \le t_n$, where t_n is the corresponding value for polyphase sorting (1).

29. [*M47*] Exercise 28 shows that the polyphase distribution is optimal among all "merge until empty" patterns in the minimum-phase sense. But is it optimal also in the minimum-pass sense?

Let a be relatively prime to b, and assume that $a + b$ is the Fibonacci number F_n. Prove or disprove the following conjecture due to R. M. Karp: The number of initial runs processed during the merge-until-empty pattern starting with distribution (a, b)

is greater than or equal to $((n-5)F_{n+1}+(2n+2)F_n)/5$. (The latter figure is achieved when $a = F_{n-1}$, $b = F_{n-2}$.)

30. [*42*] Prepare a table analogous to Table 2, for the tape-splitting polyphase merge.

5.4.3. The Cascade Merge

Another basic pattern, called the "cascade merge," was actually discovered before polyphase [B. K. Betz and W. C. Carter, *ACM Nat'l Conference* **14** (1959), Paper 14]. This approach is illustrated for six tapes and 190 initial runs in the following table, using the notation developed in Section 5.4.2:

	T1	T2	T3	T4	T5	T6	Initial runs processed
Pass 1.	1^{55}	1^{50}	1^{41}	1^{29}	1^{15}	—	190
Pass 2.	—	$1^{5}{}_{*}$	2^{9}	3^{12}	4^{14}	5^{15}	190
Pass 3.	15^{5}	14^{4}	12^{3}	9^{2}	$5^{1}{}_{*}$	—	190
Pass 4.	—	$15^{1}{}_{*}$	29^{1}	41^{1}	50^{1}	55^{1}	190
Pass 5.	190^{1}	—	—	—	—	—	190

A cascade merge, like polyphase, starts out with a "perfect distribution" of runs on tapes, although the rule for perfect distributions is somewhat different from those in Section 5.4.2. Each line in the table represents a complete pass over *all* the data. Pass 2, for example, is obtained by doing a five-way merge from T1, T2, T3, T4, T5 to T6, until T5 is empty (this puts 15 runs of relative length 5 on T6), then a four-way merge from T1, T2, T3, T4 to T5, then a three-way merge to T4, a two-way merge to T3, and finally a one-way merge (a copying operation) from T1 to T2. Pass 3 is obtained in the same way, first doing a five-way merge until one tape becomes empty, then a four-way merge, and so on. (Perhaps the present section of this book should be numbered 5.4.3.2.1 instead of 5.4.3!)

It is clear that the copying operations are unnecessary, and they could be omitted. Actually, however, in the six-tape case this copying takes only a small percentage of the total time. The items marked with an asterisk in the above table are those which were simply copied; only 25 of the 950 runs processed are of this type. Most of the time is devoted to five-way and four-way merging.

At first it may seem that the cascade pattern is a rather poor choice, by comparison with polyphase, since standard polyphase uses $(T - 1)$-way merging throughout while the cascade uses $(T - 1)$-way, $(T - 2)$-way, $(T - 3)$-way, etc. But in fact it is asymptotically *better* than polyphase, on six or more tapes! As we have observed in Section 5.4.2, a high order of merge is not a guarantee of efficiency. Table 1 shows the performance characteristics of cascade merge, by analogy with the similar tables in Section 5.4.2.

Table 1

APPROXIMATE BEHAVIOR OF CASCADE MERGE SORTING

Tapes	Passes (with copying)	Passes (without copying)	Growth Ratio
3	$2.078 \ln S + 0.672$	$1.504 \ln S + 0.992$	1.6180340
4	$1.235 \ln S + 0.754$	$1.102 \ln S + 0.820$	2.2469796
5	$0.946 \ln S + 0.796$	$0.897 \ln S + 0.800$	2.8793852
6	$0.796 \ln S + 0.821$	$0.773 \ln S + 0.808$	3.5133371
7	$0.703 \ln S + 0.839$	$0.691 \ln S + 0.822$	4.1481149
8	$0.639 \ln S + 0.852$	$0.632 \ln S + 0.834$	4.7833861
9	$0.592 \ln S + 0.861$	$0.587 \ln S + 0.845$	5.4189757
10	$0.555 \ln S + 0.869$	$0.552 \ln S + 0.854$	6.0547828
20	$0.397 \ln S + 0.905$	$0.397 \ln S + 0.901$	12.4174426

It is not hard to derive the "perfect distributions" for a cascade merge. With six tapes, they are

Level	T1	T2	T3	T4	T5
0	1	0	0	0	0
1	1	1	1	1	1
2	5	4	3	2	1
3	15	14	12	9	5
4	55	50	41	29	15
5	190	175	146	105	55
. . . . .					
n	a_n	b_n	c_n	d_n	e_n
$n+1$	$a_n+b_n+c_n+d_n+e_n$	$a_n+b_n+c_n+d_n$	$a_n+b_n+c_n$	a_n+b_n	a_n

(1)

It is interesting to note that the relative magnitudes of these numbers appear also in the diagonals of a regular $(2T - 1)$-sided polygon. For example, the five diagonals in the undecagon of Fig. 73 have relative lengths very nearly equal to 190, 175, 146, 105, and 55! We shall prove this remarkable fact later in this section, and we shall also see that the relative amount of time spent in $(T - 1)$-way merging, $(T - 2)$-way merging, . . . , 1-way merging is approximately proportional to the *squares* of the lengths of these diagonals.

***Initial distribution of runs.** When the actual number of initial runs isn't perfect, we can insert dummy runs as usual. A superficial analysis of this situation indicates that the method of dummy run assignment is immaterial, since cascade merging operates by complete passes; if we have 190 initial runs, each record is processed five times as in the above example, but if there are 191 we must apparently go up a level so that every record is processed six times.

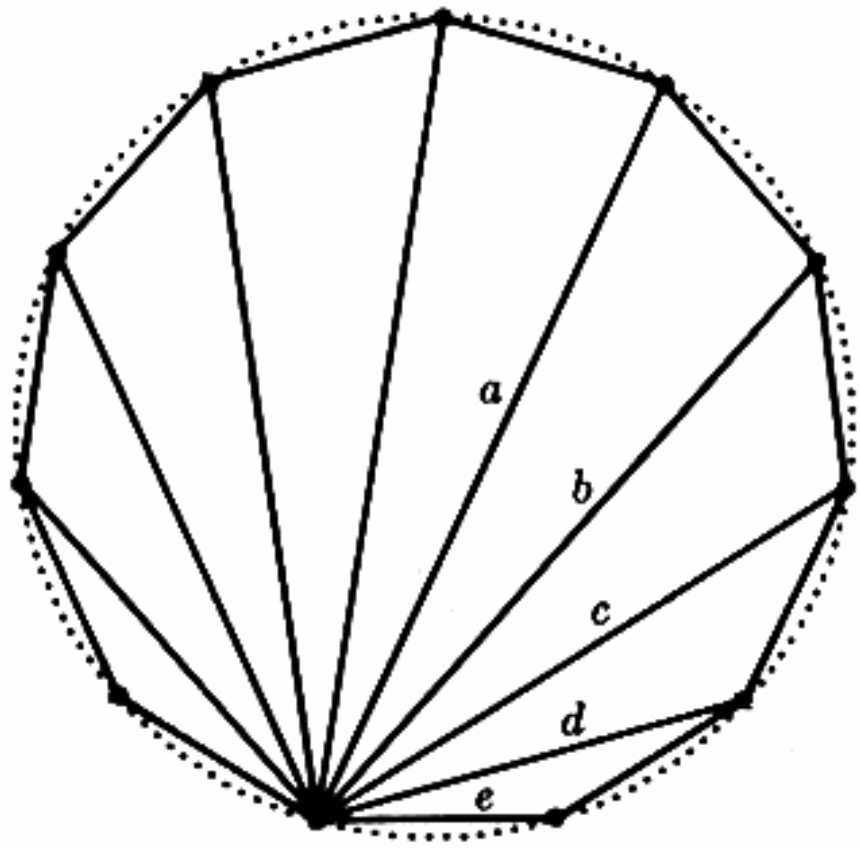


Fig. 73. Geometrical interpretation of cascade numbers.

Fortunately this abrupt change is not actually necessary; David E. Ferguson has found a way to distribute initial runs so that many of the operations during the first merge pass reduce to copying the contents of a tape. When such copying relations are bypassed (by simply changing "logical" tape unit numbers relative to the "physical" numbers as in Algorithm 5.4.2D), we obtain a relatively smooth transition from level to level, as shown in Fig. 74.

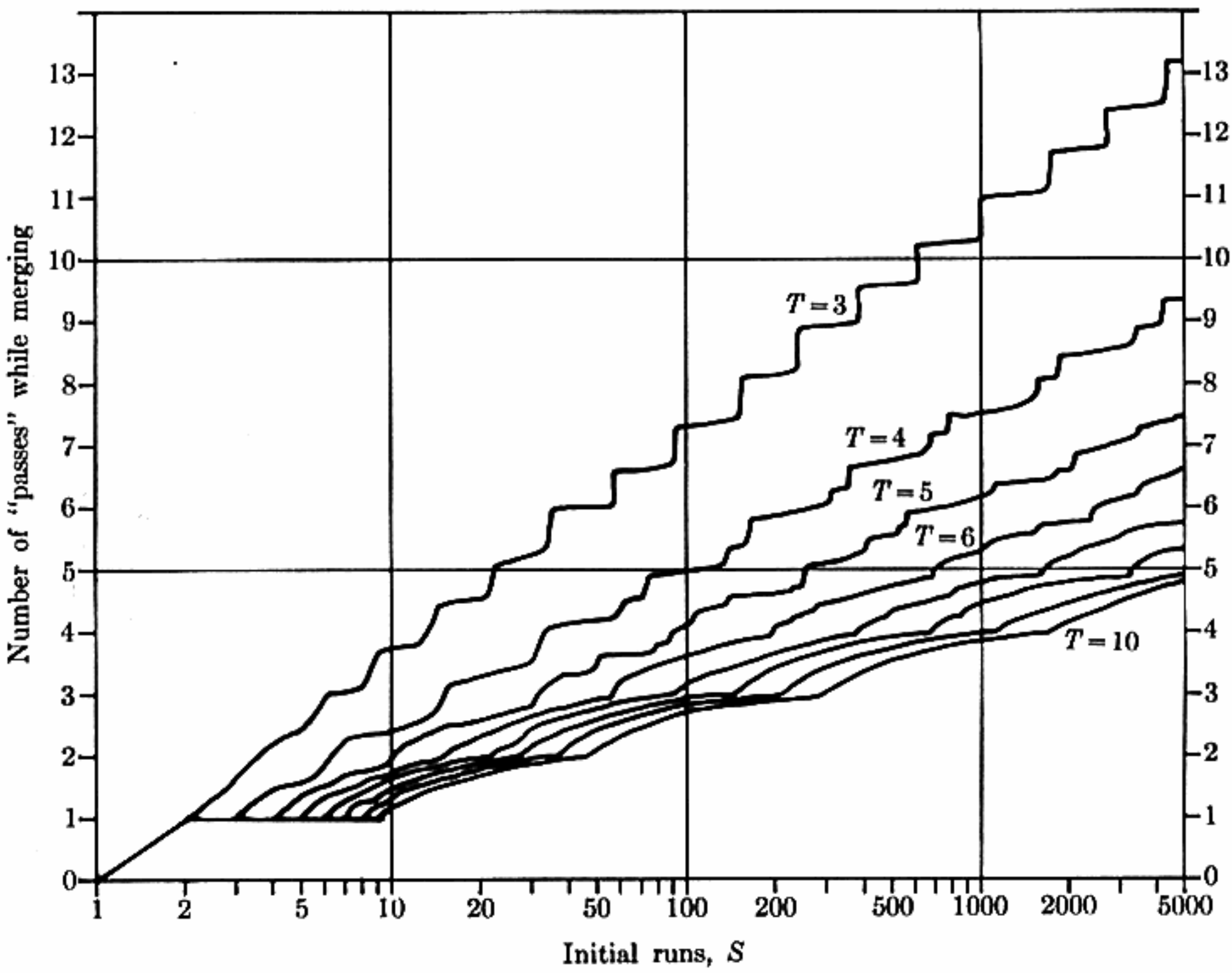


Fig. 74. Efficiency of cascade merge with the distribution of Algorithm D.

Suppose that (a, b, c, d, e) is a perfect distribution, where $a \geq b \geq c \geq d \geq e$. By redefining the correspondence between logical and physical tape units, we can imagine that the distribution is actually (e, d, c, b, a), with a runs on T5, b on T4, etc. The next perfect distribution is $(a + b + c + d + e, a + b + c + d, a + b + c, a + b, a)$; and if the input is exhausted before we reach this next level, let us assume that the tapes contain, respectively, $(D_1, D_2, D_3, D_4, D_5)$ dummy runs, where

$$D_1 \leq a + b + c + d, \qquad D_2 \leq a + b + c, \qquad D_3 \leq a + b,$$
$$D_4 \leq a, \qquad D_5 = 0; \qquad D_1 \geq D_2 \geq D_3 \geq D_4 \geq D_5. \tag{2}$$

We are free to imagine that the dummy runs appear in any convenient place on the tapes. The first merge pass is supposed to produce a runs by five-way merging, then b by four-way merging, etc., and our goal is to arrange the dummies so as to replace merging by copying. It is convenient to do the first merge pass as follows:

1. If $D_4 = a$, subtract a from each of D_1, D_2, D_3, D_4 and pretend that T5 is the result of the merge. If $D_4 < a$, merge a runs from tapes T1 through T5, using the minimum possible number of dummies on tapes T1 through T5 so that the new values of D_1, D_2, D_3, D_4 will satisfy

$$D_1 \leq b + c + d, \qquad D_2 \leq b + c, \qquad D_3 \leq b,$$
$$D_4 = 0; \qquad D_1 \geq D_2 \geq D_3 \geq D_4. \tag{3}$$

Thus, if D_2 was originally $\leq b + c$, we use no dummies from it at this step, while if $b + c < D_2 \leq a + b + c$ we use exactly $D_2 - b - c$ of them.

2. (This step is similar to step 1, but "shifted.") If $D_3 = b$, subtract b from each of D_1, D_2, D_3 and pretend that T4 is the result of the merge. If $D_3 < b$, merge b runs from tapes T1 through T4, reducing the number of dummies if necessary in order to make

$$D_1 \leq b + c, \qquad D_2 \leq b, \qquad D_3 = 0; \qquad D_1 \geq D_2 \geq D_3.$$

3. And so on.

Ferguson's method of distributing runs to tapes can be illustrated by considering the process of going from level 3 to level 4 in (1). Assume that "logical" tapes (T1, . . . , T5) contain respectively (5, 9, 12, 14, 15) runs, and that we want eventually to bring this up to (55, 50, 41, 29, 15). The procedure may be summarized as shown at the top of the facing page.

	Add to T1	Add to T2	Add to T3	Add to T4	Add to T5	"Amount saved"
Step (1, 1)	9	0	0	0	0	$15 + 14 + 12 + 5$
Step (2, 2)	3	12	0	0	0	$15 + 14 + 9 + 5$
Step (2, 1)	9	0	0	0	0	$15 + 14 + 5$
Step (3, 3)	2	2	14	0	0	$15 + 12 + 5$
Step (3, 2)	3	12	0	0	0	$15 + 9 + 5$
Step (3, 1)	9	0	0	0	0	$15 + 5$
Step (4, 4)	1	1	1	15	0	$14 + 5$
Step (4, 3)	2	2	14	0	0	$12 + 5$
Step (4, 2)	3	12	0	0	0	$9 + 5$
Step (4, 1)	9	0	0	0	0	5

We first put nine runs on T1, then (3, 12) on T1 and T2, etc. If the input becomes exhausted during, say, Step (3, 2), then the "amount saved" is $15 + 9 + 5$, meaning that the five-way merge of 15 runs, the two-way merge of 9 runs, and the one-way merge of 5 runs, are avoided by the dummy run assignment. In other words, $15 + 9 + 5$ of the runs present at level 3 are not processed during the first merge phase.

The following algorithm defines the process in detail.

Algorithm C (*Cascade merge sorting with special distribution*). This algorithm takes initial runs and disperses them to tapes, one run at a time, until the supply of initial runs is exhausted. Then it specifies how the tapes are to be merged, assuming that there are $T \geq 3$ available tape units, using at most $(T - 1)$-way merging and avoiding unnecessary one-way merging. Tape T may be used to hold the input, since it does not receive any initial runs. The following tables are maintained:

A[j], $1 \leq j \leq T$: The perfect cascade distribution we have most recently reached.

AA[j], $1 \leq j \leq T$: The perfect cascade distribution we are striving for.

D[j], $1 \leq j \leq T$: Number of dummy runs assumed to be present on logical tape unit number j.

M[j], $1 \leq j \leq T$: Maximum number of dummy runs desired on logical tape unit number j.

TAPE[j], $1 \leq j \leq T$: Number of the physical tape unit corresponding to logical tape unit number j.

C1. [Initialize.] Set A[k] $\leftarrow$ AA[k] $\leftarrow$ D[k] $\leftarrow 0$ for $2 \leq k \leq T$; and set A[1] $\leftarrow 0$, AA[1] $\leftarrow 1$, D[1] $\leftarrow 1$. Set TAPE[k] $\leftarrow k$ for $1 \leq k \leq T$. Finally set $i \leftarrow T - 2$, $j \leftarrow 1$, $k \leftarrow 1$, $l \leftarrow 0$, $m \leftarrow 1$, and go to step C5. (This maneuvering is one way to get everything started, by jumping right into the inner loop with appropriate settings of the control variables.)

C2. [Begin new level.] (We have just reached a perfect distribution, and since there is more input we must get ready for the next level.) Increase l by 1.

Set A[k] ← AA[k], for $1 \le k \le T$; then set AA[$T-k$] ← AA[$T-k+1$] + A[k], for $k = 1, 2, \ldots, T-1$ in this order. Set (TAPE[1], TAPE[2], ..., TAPE[$T-1$]) ← (TAPE[$T-1$], ..., TAPE[2], TAPE[1]), and set D[k] ← AA[$k+1$] for $1 \le k < T$. Finally set $i \leftarrow 1$.

C3. [Begin ith sublevel.] Set $j \leftarrow i$. (The variables i and j represent "Step (i, j)" in the example shown on page 293.)

C4. [Begin Step (i, j).] Set $k \leftarrow j$ and $m \leftarrow$ A[$T-j-1$]. If $m = 0$ and $i = j$, set $i \leftarrow T-2$ and return to C3; if $m = 0$ and $i \neq j$, return to C2. (Variable m represents the number of runs to be written onto TAPE[k]; $m = 0$ occurs only when $l = 1$.)

C5. [Input to TAPE[k].] Write one run on tape number TAPE[k], and decrease D[k] by 1. Then if the input is exhausted, rewind all the tapes and go to step C7.

C6. [Advance.] Decrease m by 1. If $m > 0$, return to C5. Otherwise decrease k by 1; if $k > 0$, set $m \leftarrow$ A[$T-j-1$] − A[$T-j$] and return to C5. Otherwise decrease j by 1; if $j > 0$, go to C4. Otherwise increase i by 1; if $i < T-1$, return to C3. Otherwise go to C2.

C7. [Prepare to merge.] (At this point the initial distribution is complete, and the A, AA, D, and TAPE tables describe the present states of the tapes.) Set M[k] ← AA[$k+1$] for $1 \le k < T$, and set FIRST ← 1. (Variable FIRST is nonzero only during the first merge pass.)

C8. [Cascade.] If $l = 0$, stop; sorting is complete and the output is on TAPE[1]. Otherwise, for $p = T-1, T-2, \ldots, 1$, in this order, do a p-way merge from TAPE[1], ..., TAPE[p] to TAPE[$p+1$] as follows:

If $p = 1$, simulate the one-way merge by simply rewinding TAPE[2], then interchanging TAPE[1] ↔ TAPE[2].
Otherwise if FIRST = 1 and D[$p-1$] = M[$p-1$], simulate the p-way merge by simply interchanging TAPE[p] ↔ TAPE[$p+1$], rewinding TAPE[p], and subtracting M[$p-1$] from each of D[1], ..., D[$p-1$], M[1], ..., M[$p-1$].
Otherwise, subtract M[$p-1$] from each of M[1], ..., M[$p-1$]. Then merge one run from each TAPE[j] such that $1 \le j \le p$ and D[j] ≤ M[j]; subtract one from each D[j] such that $1 \le j \le p$ and D[j] > M[j]; and put the output run on TAPE[$p+1$]. Continue doing this until TAPE[p] is empty. Then rewind TAPE[p] and TAPE[$p+1$].

C9. [Down a level.] Decrease l by 1, set FIRST ← 0, and set (TAPE[1], ..., TAPE[T]) ← (TAPE[T], ..., TAPE[1]). (At this point all D's and M's are zero and will remain so.) Return to C8. ∎

Steps C1–C6 of this algorithm do the distribution, and steps C7–C9 do the merging; the two parts are fairly independent of each other, and it would be possible to store M[k] and AA[$k+1$] in the same memory locations.

***Analysis of cascade merging.** The cascade merge is somewhat harder to analyze than polyphase, but the analysis is especially interesting because so

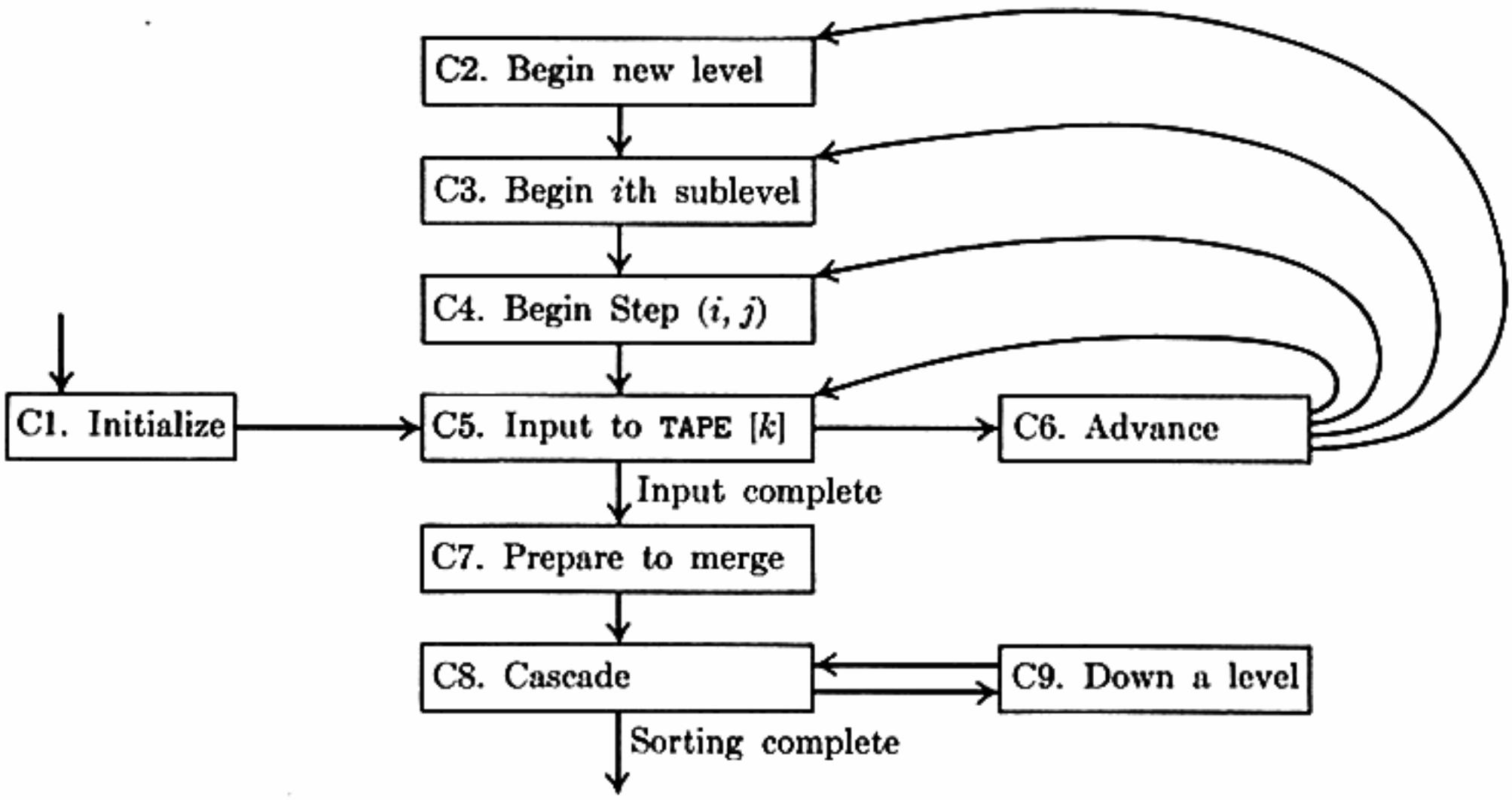


Fig. 75. The cascade merge, with special distribution.

many remarkable formulas are present. Readers who enjoy discrete mathematics are urged to study the cascade distribution for themselves, *before reading further*, since the numbers have so many extraordinary properties which are a pleasure to discover! We shall discuss here one of the many ways to approach the analysis, emphasizing the way in which the results might be discovered.

For convenience, let us consider the six-tape case, looking for formulas which generalize to all T. Relations (1) lead to the first basic pattern:

$$\begin{aligned}
a_n &= a_n && = \binom{0}{0} a_n, \\
b_n &= a_n - e_{n-1} = a_n - a_{n-2} && = \binom{1}{0} a_n - \binom{2}{2} a_{n-2}, \\
c_n &= b_n - d_{n-1} = b_n - a_{n-2} - b_{n-2} && = \binom{2}{0} a_n - \binom{3}{2} a_{n-2} + \binom{4}{4} a_{n-4}, \\
d_n &= c_n - e_{n-1} = c_n - a_{n-2} - b_{n-2} - c_{n-2} && = \binom{3}{0} a_n - \binom{4}{2} a_{n-2} + \binom{5}{4} a_{n-4} - \binom{6}{6} a_{n-6}, \\
e_n &= d_n - b_{n-1} = d_n - a_{n-2} - b_{n-2} - c_{n-2} - d_{n-2} && = \binom{4}{0} a_n - \binom{5}{2} a_{n-2} + \binom{6}{4} a_{n-4} - \binom{7}{6} a_{n-6} + \binom{8}{8} a_{n-8}.
\end{aligned} \tag{4}$$

Let $A(z) = \sum_{n \geq 0} a_n z^n, \ldots, E(z) = \sum_{n \geq 0} e_n z^n$, and define the polynomials

$$\begin{aligned}
q_m(z) &= \binom{m}{0} - \binom{m+1}{2} z^2 + \binom{m+2}{4} z^4 - \cdots \\
&= \sum_{k \geq 0} \binom{m+k}{2k} (-1)^k z^{2k} = \sum_{0 \leq k \leq m} \binom{2m-k}{k} (-1)^{m-k} z^{2m-2k}.
\end{aligned} \tag{5}$$

The result of (4) can be summarized by saying that $B(z) - q_1(z)A(z), \ldots, E(z) - q_4(z)A(z)$ reduce to finite sums corresponding to the boundary conditions, namely the values of $a_{-1}, a_{-2}, a_{-3}, \ldots$ which appear in (4) for small n

but not in $A(z)$. In order to supply appropriate boundary conditions, let us run the recurrence backwards to negative levels, through level -8:

n	a_n	b_n	c_n	d_n	e_n
0	1	0	0	0	0
−1	0	0	0	0	1
−2	1	−1	0	0	0
−3	0	0	0	−1	2
−4	2	−3	1	0	0
−5	0	0	1	−4	5
−6	5	−9	5	−1	0
−7	0	−1	6	−14	14
−8	14	−28	20	−7	1

(On seven tapes the table would be similar, with entries for odd n shifted right one column.) The sequence $a_0, a_{-2}, a_{-4}, \ldots = 1, 1, 2, 5, 14, \ldots$ is a dead giveaway for computer scientists, since it occurs in connection with so many recursive algorithms (e.g. exercise 2.2.1–4 and Eq. 2.3.4.4–13); we therefore conjecture that in the T-tape case

$$\begin{aligned} a_{-2n} &= \binom{2n}{n}\frac{1}{n+1}, \qquad \text{for} \qquad 0 \le n \le T-2; \\ a_{-2n-1} &= 0, \qquad\qquad\quad \text{for} \qquad 0 \le n \le T-3. \end{aligned} \tag{6}$$

To verify that this choice is correct, it suffices to show that (6) and (4) yield the correct results for levels 0 and 1. On level 1 this is obvious, and on level 0 we have to verify that

$$\binom{m}{0}a_0 - \binom{m+1}{2}a_{-2} + \binom{m+2}{4}a_{-4} - \binom{m+3}{6}a_{-6} + \cdots = \sum_{k\ge 0}\binom{m+k}{2k}\binom{2k}{k}\frac{(-1)^k}{k+1} = \delta_{m0} \tag{7}$$

for $0 \le m \le T-2$. Fortunately this sum can be evaluated by standard techniques (it is "Problem 2," one of the basic examples in the text of Section 1.2.6).

Now we can compute the coefficients of $B(z) - q_1(z)A(z)$, etc. For example, consider the coefficient of z^{2m} in $D(z) - q_3(z)A(z)$: it is

$$\begin{aligned} \sum_{k\ge 0}\binom{3+m+k}{2m+2k}(-1)^{m+k}a_{-2k} &= \sum_{k\ge 0}\binom{3+m+k}{2m+2k}\binom{2k}{k}\frac{(-1)^{m+k}}{k+1} \\ &= (-1)^m\left(\binom{2+m}{2m-1} - \binom{3+m}{2m}\right) \\ &= (-1)^{m+1}\binom{2+m}{2m}, \end{aligned}$$

by the result of "Problem 3" in Section 1.2.6. Therefore we have deduced that

$$\begin{aligned} A(z) &= q_0(z)A(z), \\ B(z) &= q_1(z)A(z) - q_0(z), \\ C(z) &= q_2(z)A(z) - q_1(z), \\ D(z) &= q_3(z)A(z) - q_2(z), \\ E(z) &= q_4(z)A(z) - q_3(z). \end{aligned} \tag{8}$$

Furthermore we have $e_{n+1} = a_n$; hence $zA(z) = E(z)$, and

$$A(z) = q_3(z)/(q_4(z) - z). \tag{9}$$

The generating functions have now been derived in terms of the q polynomials, and so we want to understand the q's better. Exercise 1.2.9–15 is useful in this regard, since it gives us a "closed form" which may be written

$$q_m(z) = \frac{((\sqrt{4 - z^2} + iz)/2)^{2m+1} + ((\sqrt{4 - z^2} - iz)/2)^{2m+1}}{\sqrt{4 - z^2}}. \tag{10}$$

Everything simplifies if we now set $z = 2 \sin \theta$:

$$\begin{aligned} q_m(2 \sin \theta) &= \frac{(\cos \theta + i \sin \theta)^{2m+1} + (\cos \theta - i \sin \theta)^{2m+1}}{2 \cos \theta} \\ &= \frac{\cos (2m + 1)\theta}{\cos \theta}. \end{aligned} \tag{11}$$

(This coincidence leads us to suspect that the polynomial $q_m(z)$ is well known in mathematics; and indeed, a glance at appropriate tables will show that $q_m(z)$ is essentially a Chebyshev polynomial of the second kind, namely $(-1)^m U_{2m}(z/2)$ in conventional notation.)

We can now determine the roots of the denominator in (9): $q_4(2 \sin \theta) = 2 \sin \theta$ reduces to

$$\cos 9\theta = 2 \sin \theta \cos \theta = \sin 2\theta.$$

We can obtain solutions to this relation whenever $\pm 9\theta = 2\theta + (2n - \frac{1}{2})\pi$; and all such θ yield roots of the denominator in (9) provided that $\cos \theta \neq 0$. (When $\cos \theta = 0$, $q_m(\pm 2) = \pm(2m + 1)$ is never equal to ± 2.) The following eight distinct roots are therefore obtained:

$$q_4(z) - z = 0 \qquad \text{when} \qquad z = 2 \sin \frac{-5}{14}\pi,\ 2 \sin \frac{-1}{14}\pi,\ 2 \sin \frac{3}{14}\pi;$$

$$2 \sin \frac{-7}{22}\pi,\ 2 \sin \frac{-3}{22}\pi,\ 2 \sin \frac{1}{22}\pi,\ 2 \sin \frac{5}{22}\pi,\ 2 \sin \frac{9}{22}\pi.$$

Since $q_4(z)$ is a polynomial of degree 8, this accounts for all the roots. The first three of these values make $q_3(z) = 0$, so $q_3(z)$ and $q_4(z) - z$ have a polynomial of degree three as a common factor. The other five roots govern the asymptotic behavior of the coefficients of $A(z)$, if we expand (9) in partial fractions.

Considering the general T-tape case, let $\theta_k = (4k + 1)\pi/(4T - 2)$. The generating function $A(z)$ for the T-tape cascade distribution numbers takes the form

$$\frac{4}{2T-1} \sum_{-T/2<k<\lfloor T/2 \rfloor} \frac{\cos^2 \theta_k}{1 - z/(2 \sin \theta_k)} \tag{12}$$

(see exercise 8); hence

$$a_n = \frac{4}{2T-1} \sum_{-T/2<k<\lfloor T/2 \rfloor} \cos^2 \theta_k \left(\frac{1}{2 \sin \theta_k}\right)^n. \tag{13}$$

Eqs. (8) now lead to the similar formulas

$$b_n = \frac{4}{2T-1} \sum_{-T/2<k<\lfloor T/2 \rfloor} \cos \theta_k \cos 3\theta_k \left(\frac{1}{2 \sin \theta_k}\right)^n,$$

$$c_n = \frac{4}{2T-1} \sum_{-T/2<k<\lfloor T/2 \rfloor} \cos \theta_k \cos 5\theta_k \left(\frac{1}{2 \sin \theta_k}\right)^n, \tag{14}$$

$$d_n = \frac{4}{2T-1} \sum_{-T/2<k<\lfloor T/2 \rfloor} \cos \theta_k \cos 7\theta_k \left(\frac{1}{2 \sin \theta_k}\right)^n,$$

and so on. Exercise 9 shows that these equations hold for all $n \geq 0$, not only for large n. In each sum the term for $k = 0$ dominates all the others, especially when n is reasonably large; therefore the "growth ratio" is

$$\frac{1}{2 \sin \theta_0} = \frac{2}{\pi} T - \frac{1}{\pi} + \frac{\pi}{48T} + O(T^{-2}). \tag{15}$$

Cascade sorting was first analyzed by W. C. Carter [*Proc. IFIP Congress* (1962), 62–66], who obtained numerical results for small T, and by David E. Ferguson [see *CACM* **7** (1964), 297], who discovered the first two terms in the asymptotic behavior (15) of the growth ratio. During the summer of 1964, R. W. Floyd discovered the explicit form $1/(2 \sin \theta_0)$ of the growth ratio, so that exact formulas could be used for all T. An intensive analysis of the cascade numbers was independently carried out by G. N. Raney [*Canadian J. Math.* **18** (1966), 332–349], who came across them in quite another way having nothing to do with sorting. Raney observed the "ratio of diagonals" principle of Fig. 73, and derived many other interesting properties of the numbers. Floyd and Raney used matrix manipulations in their proofs (cf. exercise 6).

Modifications of cascade sorting. If one more tape is added, it is possible to overlap nearly all of the rewind time during a cascade sort. For example, we can merge T1–T5 to T7, then T1–T4 to T6, then T1–T3 to T5 (which by now is rewound), then T1–T2 to T4, and the next pass can begin when the comparatively short data on T4 has been rewound. The efficiency of this process can be predicted from the above analysis of cascade. (See Section 5.4.6 for further information.)

A "compromise merge" scheme, which includes both polyphase and cascade as special cases, was suggested by D. E. Knuth in *CACM* **6** (1963), 585–587. Each phase consists of $(T-1)$-way, $(T-2)$-way, ..., P-way merges, where P is any fixed number between 1 and $T-1$. When $P = T-1$, this is polyphase, and when $P = 1$ it is "pure cascade"; when $P = 2$ it is cascade without copy phases. Analyses of this scheme have been made by C. E. Radke [*IBM Systems J.* **5** (1966), 226–247] and by W. H. Burge [*Proc. IFIP Congress* (1971), to appear]. Burge found the generating function $\sum T_n(x)z^n$ for each (P, T) compromise merge, generalizing Eq. 5.4.2–16; he showed that the best value of P, from the standpoint of least initial runs processed as a function of S as $S \to \infty$ (using a straightforward distribution scheme and ignoring rewind time), is respectively (2, 3, 3, 4, 4, 4, 3, 3, 4) for $T = (3, 4, 5, 6, 7, 8, 9, 10, 11)$. These values of P lean more towards cascade than polyphase as T increases; and it turns out that the compromise merge is never substantially better than cascade itself. On the other hand, with an optimum choice of levels and distribution of dummy runs, as described in Section 5.4.2, pure polyphase seems to be best of all the compromise merges; unfortunately the optimum distribution is comparatively difficult to implement.

Th. L. Johnsen [*BIT* **6** (1966), 129–143] has studied a combination of balanced and polyphase merging; a rewind-overlap variation of balanced merging has been proposed by M. A. Goetz [*Digital Computer User's Handbook*, ed. by M. Klerer and G. A. Korn (New York: McGraw-Hill, 1967), 1.311–1.312]; and many other hybrid schemes can be imagined.

EXERCISES

1. [*10*] Using Table 1, compare cascade merging with the "tape-splitting" version of polyphase described in Section 5.4.2. Which is better? (Ignore rewind time.)

▶ **2.** [*22*] Compare cascade sorting on three tapes, using Algorithm C, to polyphase sorting on three tapes, using Algorithm 5.4.2D. What similarities and differences are there?

3. [*20*] Prepare a table which shows what happens when 100 initial runs are sorted on six tapes using Algorithm C.

4. [*M20*] (G. N. Raney.) An "nth level cascade distribution" is a multiset defined as follows (in the case of six tapes): $\{1, 0, 0, 0, 0\}$ is a 0th level cascade distribution; and if $\{a, b, c, d, e\}$ is an nth level cascade distribution, $\{a + b + c + d + e, a + b +$

$c+d,\ a+b+c,\ a+b,\ a\}$ is an $(n+1)$st level cascade distribution. (Since a multiset is unordered, up to 5! different $(n+1)$st level distributions can be formed from a single nth level distribution.) (a) Prove that *any* multiset $\{a, b, c, d, e\}$ of relatively prime integers is an nth level cascade distribution, for some n. (b) Prove that the distribution defined for cascade sorting is *optimum*, in the sense that, if $\{a, b, c, d, e\}$ is any nth level distribution with $a \geq b \geq c \geq d \geq e$, we have $a \leq a_n$, $b \leq b_n$, $c \leq c_n$, $d \leq d_n$, $e \leq e_n$, where $\{a_n, b_n, c_n, d_n, e_n\}$ is the distribution defined in (1).

▶ **5.** [*20*] Prove that the cascade numbers defined in (1) satisfy the law

$$a_k a_{n-k} + b_k b_{n-k} + c_k c_{n-k} + d_k d_{n-k} + e_k e_{n-k} = a_n, \qquad \text{for} \qquad 0 \leq k \leq n.$$

[*Hint:* Interpret this relation by considering how many runs of various lengths are output during the kth pass of a complete cascade sort.]

6. [*M20*] Find a 5×5 matrix Q such that the first row of Q^n contains the six-tape cascade numbers $a_n\ b_n\ c_n\ d_n\ e_n$, for all $n \geq 0$.

7. [*M20*] Given that cascade merge is being applied to a perfect distribution of a_n initial runs, find a formula for the amount of processing saved when one-way merging is suppressed.

8. [*HM23*] Derive (12).

9. [*HM26*] Derive (14).

▶ **10.** [*M28*] Instead of using the pattern (4) to begin the study of the cascade numbers, start with the identities

$$\begin{aligned}
e_n &= a_{n-1} &&= \binom{1}{1} a_{n-1},\\
d_n &= 2a_{n-1} - e_{n-2} &&= \binom{2}{1} a_{n-1} - \binom{3}{3} a_{n-3},\\
c_n &= 3a_{n-1} - d_{n-2} - 2e_{n-2} &&= \binom{3}{1} a_{n-1} - \binom{4}{3} a_{n-3} + \binom{5}{5} a_{n-5},
\end{aligned}$$

etc. Letting

$$r_m(z) = \binom{m}{1} z - \binom{m+1}{3} z^3 + \binom{m+2}{5} z^5 - \cdots,$$

express $A(z)$, $B(z)$, etc. in terms of these r polynomials.

11. [*M38*] Let

$$f_m(z) = \sum_{0 \leq k \leq m} \binom{\lfloor (m+k)/2 \rfloor}{k} (-1)^{\lceil k/2 \rceil} z^k.$$

Prove that the generating function $A(z)$ for the T-tape cascade numbers is equal to $f_{T-3}(z)/f_{T-1}(z)$, where the numerator and denominator in this expression have no common factor.

12. [*M40*] Prove that Ferguson's distribution scheme is optimum, in the sense that no method of placing the dummy runs, satisfying (2), will cause less initial runs to be processed during the first pass, *provided* that the strategy of steps C7–C9 is used during this pass.

13. [*40*] The text suggests overlapping most of the rewind time, by adding an extra tape. Explore this idea. (For example, the text's scheme involves waiting for T4 to rewind; would it be better to omit T4 from the first merge phase of the next pass?)

5.4.4. Reading Tape Backwards

Many magnetic tape units have the ability to read tape in the opposite direction from which it was written. The merging patterns we have encountered so far always write information onto tape in the "forward" direction, then rewind the tape, read it forwards, and rewind again. (The tape files therefore behave as queues, operating in a first-in-first-out manner.) Backwards reading allows us to eliminate both of these rewind operations: We write the tape forwards and read it backwards. (In this case the files behave as stacks, since they are used in a last-in-first-out manner.)

The balanced, polyphase, and cascade merge patterns can all be adapted to backward reading. The main difference is that *merging reverses the order of the runs* when we read backward and write forward. If two runs are in ascending order on tape, we can merge them while reading backwards, but this produces descending order. The descending runs produced in this way will subsequently become ascending on the next pass; so the merging algorithms must be capable of dealing with runs in either order. A programmer confronted with read-backwards for the first time will often find himself standing on his head!

As an example of backwards reading, consider the process of merging 8 initial runs, using a *balanced* merge on four tapes. The operations can be summarized as follows:

	T1	T2	T3	T4	
Pass 1.	$A_1A_1A_1A_1$	$A_1A_1A_1A_1$	—	—	Initial distribution.
Pass 2.	—	—	D_2D_2	D_2D_2	Merge to T3 and T4.
Pass 3.	A_4	A_4	—	—	Merge to T1 and T2.
Pass 4.	—	—	D_8	—	Final merge to T3.

Here A_r stands for a run which appears on tape in ascending order, if the tape is read forwards as in our previous examples, having relative length r; D_r is the corresponding notation for a descending run of length r. During Pass 2 the ascending runs become descending: they appear to be descending in the input since we are reading backwards. They switch orientation again on Pass 3.

Note that the above process finishes with the result on tape T3, in *descending* order. If this is bad (depending on whether the output is to be read backwards, or to be dismounted and put away for future use), we could copy it to another tape, reversing the direction. A faster way would be to rewind T1 and T2 after Pass 3, producing A_8 during Pass 4. Still faster would be to start with eight *descending* runs during Pass 1, since this would interchange all the A's and D's. However, the balanced merge on 16 initial runs would

require the initial runs to be ascending; and since we usually don't know in advance how many initial runs will be formed, it is necessary to choose one consistent direction. Therefore the idea of rewinding after Pass 3 is probably best.

The *cascade* merge carries over in the same way. For example, consider sorting 14 initial runs on four tapes:

	T1	T2	T3	T4
Pass 1.	$A_1A_1A_1A_1A_1A_1$	$A_1A_1A_1A_1A_1$	$A_1A_1A_1$	—
Pass 2.	—	D_1	D_2D_2	$D_3D_3D_3$
Pass 3.	A_6	A_5	A_3	—
Pass 4.	—	—	—	D_{14}

Again, we could produce A_{14} instead of D_{14}, if we rewound T1, T2, T3 just before the final pass. Note that this is a "pure" cascade merge, in the sense that all of the one-way merges have explicitly been performed. If we had suppressed the copying operations, as in Algorithm 5.4.3D, we would have been confronted with the situation

$$A_1 \qquad — \qquad D_2D_2 \qquad D_3D_3D_3$$

after Pass 2, and it would have been impossible to continue with a three-way merge since we cannot merge runs that are in opposite directions! The operation of copying T1 to T2 could be avoided if we rewound T1 and proceeded to read it forward during the next merge phase (while reading T3 and T4 backwards). But it would then be necessary to rewind T1 again after merging, so this trick trades one copy for two rewinds.

Thus the distribution method of Algorithm 5.4.3C does not work as efficiently for read-backwards as for read-forwards; the amount of time required jumps rather sharply every time the number of initial runs passes a "perfect" cascade distribution number. Another dispersion technique can be used to give a smoother transition between perfect cascade distributions (see exercise 17).

Read-backward polyphase. At first glance (and even at second and third glance!), the polyphase merge scheme seems to be totally unfit for reading backwards. For example, suppose that we have 13 initial runs and three tapes:

	T1	T2	T3
Phase 1.	$A_1A_1A_1A_1A_1$	$A_1A_1A_1A_1A_1A_1A_1A_1$	—
Phase 2.	—	$A_1A_1A_1$	$D_2D_2D_2D_2D_2$.

Now we're stuck; we could rewind either T2 or T3 and then read it forwards, while reading the other tape backwards, but this would get things rather mixed up and we would have gained comparatively little by reading backwards.

An ingenious idea which saves the situation is to *alternate the direction of runs on each tape.* Then the merging can proceed in perfect synchronization:

Phase 1.	$A_1D_1A_1D_1A_1$	$D_1A_1D_1A_1D_1A_1D_1A_1$	—
Phase 2.	—	$D_1A_1D_1$	$D_2A_2D_2A_2D_2$
Phase 3.	$A_3D_3A_3$	—	D_2A_2
Phase 4.	A_3	D_5A_5	—
Phase 5.	—	D_5	D_8
Phase 6.	A_{13}	—	—

This principle was mentioned briefly by R. L. Gilstad in his original article on polyphase merging, and he described it more fully in *CACM* **6** (1963), 220–223.

The $ADA\cdots$ technique works properly for polyphase merging on *any* number of tapes; for we can show that the A's and D's will be properly synchronized at each phase, provided only that the initial distribution pass produces alternating A's and D's on each tape and that each tape ends with A (or each tape ends with D): Since the last run written on the output file during one phase is in the opposite direction from the last runs used from the input files, the next phase always finds its runs in the proper orientation. Furthermore we have seen in exercise 5.4.2–13 that most of the perfect Fibonacci distributions call for an *odd* number of runs on one tape (the eventual output tape), and an *even* number of runs on each other tape. If T1 is designated as the final output tape, we can therefore guarantee that all tapes end with an A run, if we start T1 with an A and let the remaining tapes start with a D. A distribution method analogous to Algorithm 5.4.2D can be used, modified so that the distributions on each level have T1 as the final output tape. (We skip levels $1, T+1, 2T+1, \ldots$, since they are the levels in which the initially empty tape is the final output tape.) For example, in the six-tape case, we can use the following distribution numbers in place of 5.4.2–(1):

Level	T1	T2	T3	T4	T5	Total	Final output will be on	
0	1	0	0	0	0	1	T1	
2	1	2	2	2	2	9	T1	
3	3	4	4	4	2	17	T1	
4	7	8	8	6	4	33	T1	(1)
5	15	16	14	12	8	65	T1	
6	31	30	28	24	16	129	T1	
8	61	120	116	108	92	497	T1	

Thus, T1 always get an odd number of runs, while T2 through T5 get the even numbers, in decreasing order for flexibility in dummy run assignment. Such a distribution has the advantage that the final output tape is known in advance, regardless of the number of initial runs that happen to be present. It turns

out (see exercise 3) that the output will always appear in *ascending* order on T1 when this scheme is used.

Another way to handle the distribution for read-backward polyphase has been suggested by D. T. Goodwin and J. L. Venn [*CACM* **7** (1964), 315]. We can distribute runs almost as in Algorithm 5.4.2D, beginning with a D run on each tape. When the input is exhausted, a dummy A run is imagined to be at the beginning of the unique "odd" tape, unless a distribution with all odd numbers has been reached. Other dummies are imagined at the end of the tapes, or grouped into pairs in the middle. The question of optimum placement of dummy runs is analyzed in exercise 5 below.

Optimum merge patterns. So far we have been discussing various patterns for merging on tape, without asking for "best possible" methods. It appears to be quite difficult to determine the optimal patterns, especially in the read-forward case where the interaction of rewind time with merge time is hard to handle. On the other hand, when merging is done by reading backwards and writing forwards, all rewinding is essentially eliminated, and it is possible to get a fairly good characterization of optimal merging patterns. Richard M. Karp has introduced some very interesting ways to attack this problem, and we shall conclude this section by discussing the theory he has developed.

In the first place we need a more satisfactory way to describe merging patterns, instead of the rather mysterious "tape contents" charts which have been used above. Karp has suggested two ways to do this, the *vector representation* and the *tree representation* of a merge pattern. Both forms of representation are useful in practice, so we shall describe them in turn.

The vector representation of a merge pattern consists of a sequence of "merge vectors" $y^{(m)} \ldots y^{(1)}y^{(0)}$, each of which has T components. The ith-last merge step is represented by $y^{(i)}$ in the following way:

$$y_j^{(i)} = \begin{cases} +1, & \text{if tape number } j \text{ is an input to the merge;} \\ 0, & \text{if tape number } j \text{ is not used in the merge;} \\ -1, & \text{if tape number } j \text{ gets the output of the merge.} \end{cases} \tag{2}$$

Thus, exactly one component of $y^{(i)}$ is -1, and the other components are 0's and 1's. The final vector $y^{(0)}$ is special; it is a unit vector, having 1 in position j if the final sorted output appears on unit j, and 0 elsewhere. These definitions imply that the vector sum

$$v^{(i)} = y^{(i)} + y^{(i-1)} + \cdots + y^{(0)} \tag{3}$$

represents the distribution of runs on tape just before the ith-last merge step, with $v_j^{(i)}$ runs on tape j. In particular, $v^{(m)}$ tells how many runs the initial distribution pass places on each tape.

The three merge patterns described in tabular form earlier in this section have the following vector representations:

Balanced ($T = 4$, $S = 8$)

$$
\begin{aligned}
v^{(7)} &= (\ 4,\ \ 4,\ \ 0,\ \ 0) \\
y^{(7)} &= (+1,\ +1,\ -1,\ \ 0) \\
y^{(6)} &= (+1,\ +1,\ \ 0,\ -1) \\
y^{(5)} &= (+1,\ +1,\ -1,\ \ 0) \\
y^{(4)} &= (+1,\ +1,\ \ 0,\ -1) \\
y^{(3)} &= (-1,\ \ 0,\ +1,\ +1) \\
y^{(2)} &= (\ 0,\ -1,\ +1,\ +1) \\
y^{(1)} &= (+1,\ +1,\ -1,\ \ 0) \\
y^{(0)} &= (\ 0,\ \ 0,\ \ 1,\ \ 0)
\end{aligned}
$$

Cascade ($T = 4$, $S = 14$)

$$
\begin{aligned}
v^{(10)} &= (\ 6,\ \ 5,\ \ 3,\ \ 0) \\
y^{(10)} &= (+1,\ +1,\ +1,\ -1) \\
y^{(9)} &= (+1,\ +1,\ +1,\ -1) \\
y^{(8)} &= (+1,\ +1,\ +1,\ -1) \\
y^{(7)} &= (+1,\ +1,\ -1,\ \ 0) \\
y^{(6)} &= (+1,\ +1,\ -1,\ \ 0) \\
y^{(5)} &= (+1,\ -1,\ \ 0,\ \ 0) \\
y^{(4)} &= (-1,\ +1,\ +1,\ +1) \\
y^{(3)} &= (\ 0,\ -1,\ +1,\ +1) \\
y^{(2)} &= (\ 0,\ \ 0,\ -1,\ +1) \\
y^{(1)} &= (+1,\ +1,\ +1,\ -1) \\
y^{(0)} &= (\ 0,\ \ 0,\ \ 0,\ \ 1)
\end{aligned}
$$

Polyphase ($T = 3$, $S = 13$)

$$
\begin{aligned}
v^{(12)} &= (\ 5,\ \ 8,\ \ 0) \\
y^{(12)} &= (+1,\ +1,\ -1) \\
y^{(11)} &= (+1,\ +1,\ -1) \\
y^{(10)} &= (+1,\ +1,\ -1) \\
y^{(9)} &= (+1,\ +1,\ -1) \\
y^{(8)} &= (+1,\ +1,\ -1) \\
y^{(7)} &= (-1,\ +1,\ +1) \\
y^{(6)} &= (-1,\ +1,\ +1) \\
y^{(5)} &= (-1,\ +1,\ +1) \\
y^{(4)} &= (+1,\ -1,\ +1) \\
y^{(3)} &= (+1,\ -1,\ +1) \\
y^{(2)} &= (+1,\ +1,\ -1) \\
y^{(1)} &= (-1,\ +1,\ +1) \\
y^{(0)} &= (\ 1,\ \ 0,\ \ 0)
\end{aligned}
$$

It may seem awkward to number these vectors backwards, with $y^{(m)}$ coming first and $y^{(0)}$ last, but this peculiar viewpoint turns out to be advantageous for developing the theory. One good way to search for an optimal method is to start with the sorted output and to imagine "unmerging" it to various tapes, then unmerging these, etc., considering the successive distributions $v^{(0)}$, $v^{(1)}$, $v^{(2)}, \ldots$ in the reverse order from which they actually occur during the sorting process. In fact that is essentially the approach we have taken already in our analysis of polyphase and cascade merging.

Every merge pattern obviously has a vector representation. Conversely, it is easy to see that the sequence of vectors $y^{(m)} \ldots y^{(1)}\, y^{(0)}$ corresponds to an actual merge pattern if and only if the following three conditions are satisfied:

i) $y^{(0)}$ is a unit vector.

ii) $y^{(i)}$ has exactly one component equal to -1, all other components equal to 0 or $+1$, for $m \geq i \geq 1$.

iii) All components of $y^{(i)} + \cdots + y^{(1)} + y^{(0)}$ are nonnegative, for $m \geq i \geq 1$.

The tree representation of a merge pattern gives another picture of the same information. We construct a tree with one external "leaf" node for each initial run, and one internal node for each run that is merged, in such a way that the descendants of each internal node are the runs from which it was fabricated. Each internal node is labeled with the step number on which the corresponding run was formed, numbering steps backwards as in the vector representation; furthermore, the line just above each node is labeled with the name of the tape on which that run appears. For example, the three merge patterns above have the tree representations depicted in Fig. 76, if we call the tapes A, B, C, D instead of T1, T2, T3, T4.

This representation displays many of the relevant properties of the merge pattern in convenient form; for example, if the run on level 0 of the tree (the

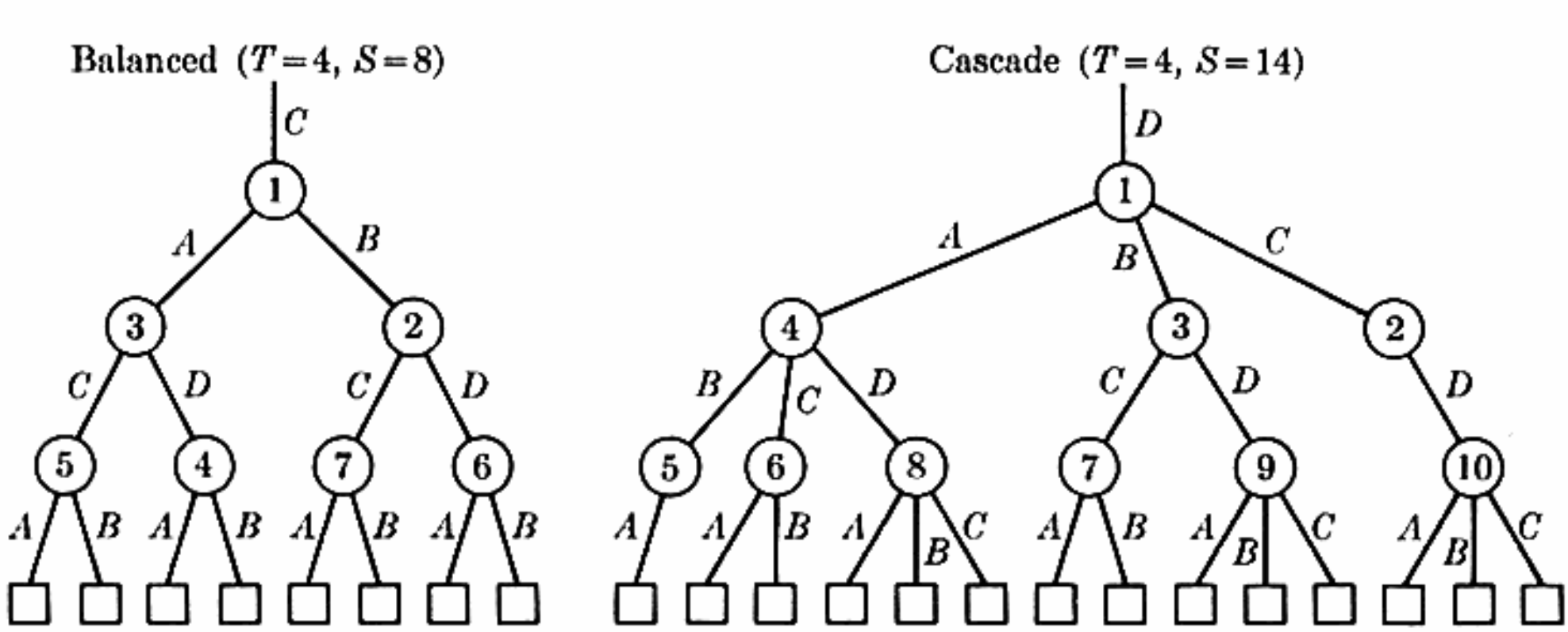


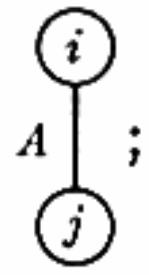
Fig. 76. Tree representations of three merge patterns.

root) is to be ascending, then the runs on level 1 must be descending, those on level 2 must be ascending, etc.; an initial run is ascending if and only if the corresponding external node is on an even-numbered level. Furthermore the total number of initial runs processed during the merging (not including the initial distribution) is exactly equal to the *external path length* of the tree, since each initial run on level k is processed exactly k times.

Every merge pattern has a tree representation, but not every tree defines a merge pattern. A tree whose internal nodes have been labeled with the numbers 1 through m, and whose lines have been labeled with tape names, represents a valid read-backwards merge pattern if and only if

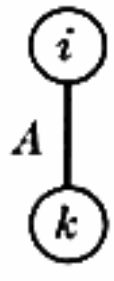
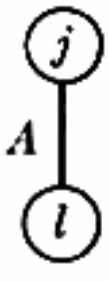
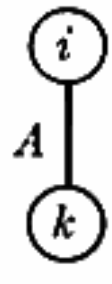
a) No two lines adjacent to the same internal node have the same tape name;

b) If $i > j$, and if A is a tape name, the tree does not contain the configuration



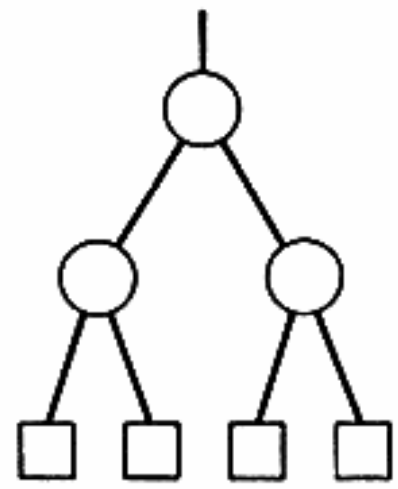
;

c) If $i < j < k < l$, and if A is a tape name, the tree does not contain

both  and  or both  and  . (4)

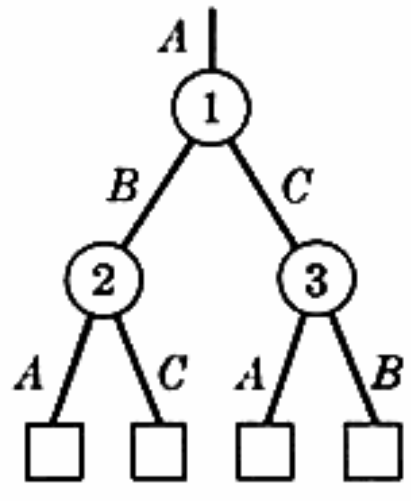
Condition (a) is self-evident, since the input and output tapes in a merge must be distinct; similarly, (b) is obvious. The "no crossover" condition (c) mirrors the last-in-first-out restriction which characterizes read-backwards operations on tape: A run formed at step k must be removed before a run formed previously on that same tape; hence the configurations in (4) are impossible. It is not difficult to verify that any labeled tree satisfying conditions (a), (b), (c) does indeed correspond to a read-backwards merge pattern.

If there are T tape units, condition (a) implies that the degree of each internal node is $T - 1$ or less. It is not always possible to attach suitable labels to all such trees; for example, when $T = 3$ there is no merge pattern whose tree has the shape



(5)

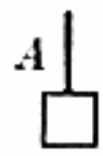
This shape would lead to an optimal merge pattern if we could attach step numbers and tape names in a suitable way, since it is the only way to achieve the minimum external path length in a tree having four external nodes. But there is essentially only one way to do the labeling according to conditions (a) and (b), because of the symmetries of the diagram, namely,



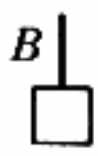
(6)

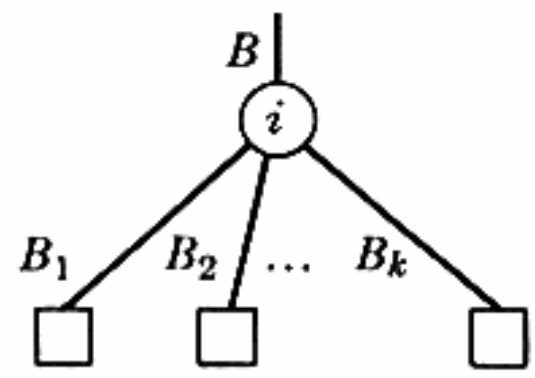
and this violates condition (c). A shape which *can* be labeled, according to the above conditions, using T or less tape names, is called a *T-lifo* tree.

Another way to characterize all labeled trees that can arise from merge patterns is to consider how all such trees can be "grown." Start with some tape name, say A, and with the seedling tree



Step number i in the tree's growth consists of choosing distinct tape names $B, B_1, B_2, \ldots, B_k$, and changing the *most recently formed* external node corresponding to B

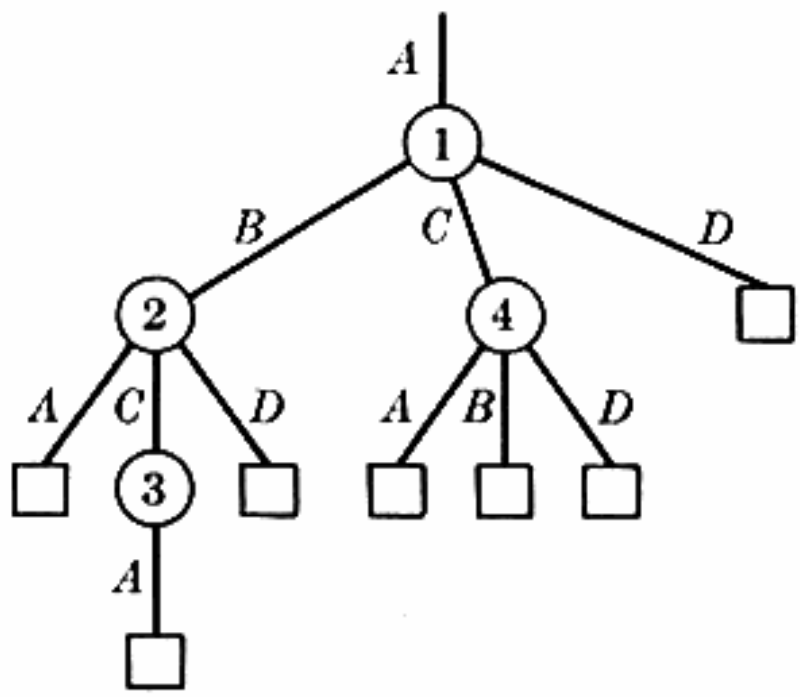
from 

to 

(7)

This "last formed, first grown on" rule explains exactly how the tree representation can be constructed directly from the vector representation.

The determination of strictly optimum T-tape merge patterns, i.e. of trees which have the exact minimum path length over all T-lifo trees with a given number of external nodes, seems to be quite difficult. For example, the following nonobvious pattern turns out to be an optimum way to merge seven initial runs on four tapes, reading backwards:



(8)

A one-way merge is actually necessary to achieve the optimum! (See exercise 8.) On the other hand, it is not so difficult to give constructions that are *asymptotically* optimal, for any fixed T.

Let $K_T(n)$ be the minimum external path length achievable in a T-lifo tree with n external nodes. From the theory developed in Section 2.3.4.5, it is not difficult to prove that

$$K_T(n) \geq nq - \lfloor ((T-1)^q - n)/(T-2) \rfloor, \qquad q = \lceil \log_{T-1} n \rceil, \tag{9}$$

since this is the minimum external path length of *any* tree with n external nodes and all nodes of degree $< T$. At the present time comparatively few values of $K_T(n)$ are known exactly. Here are some upper bounds which are probably exact:

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$K_3(n) \leq$	0	2	5	9	12	16	21	25	30	34	39	45	50	56	61
$K_4(n) \leq$	0	2	3	6	8	11	14	17	20	24	27	31	33	37	40

(10)

Karp has discovered that *any* tree whose internal nodes have degrees $< T$ is *almost* T-lifo, in the sense that it can be made T-lifo by changing some of the external nodes to one-way merges. In fact, the construction of a suitable labeling is fairly simple. Let A be a particular tape name, and proceed as follows:

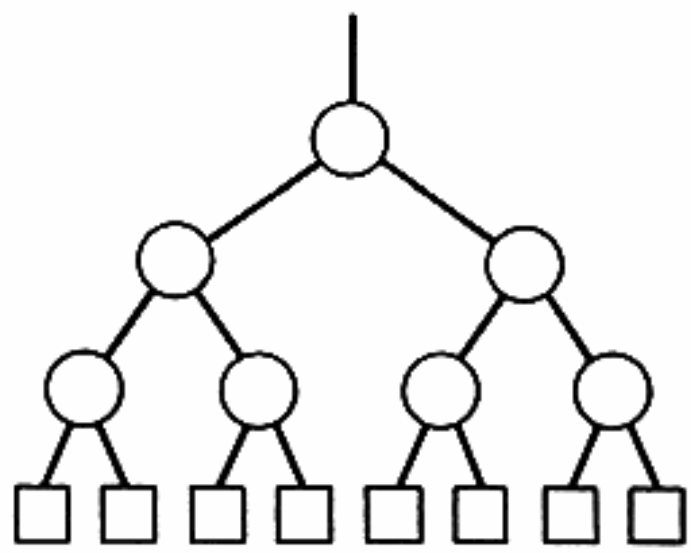
Step 1. Attach tape names to the lines of the tree diagram, in any manner consistent with condition (a) above, provided that the special name A is used only in the leftmost line of a branch.

Step 2. Replace each external node of the form

whenever $B \neq A$.

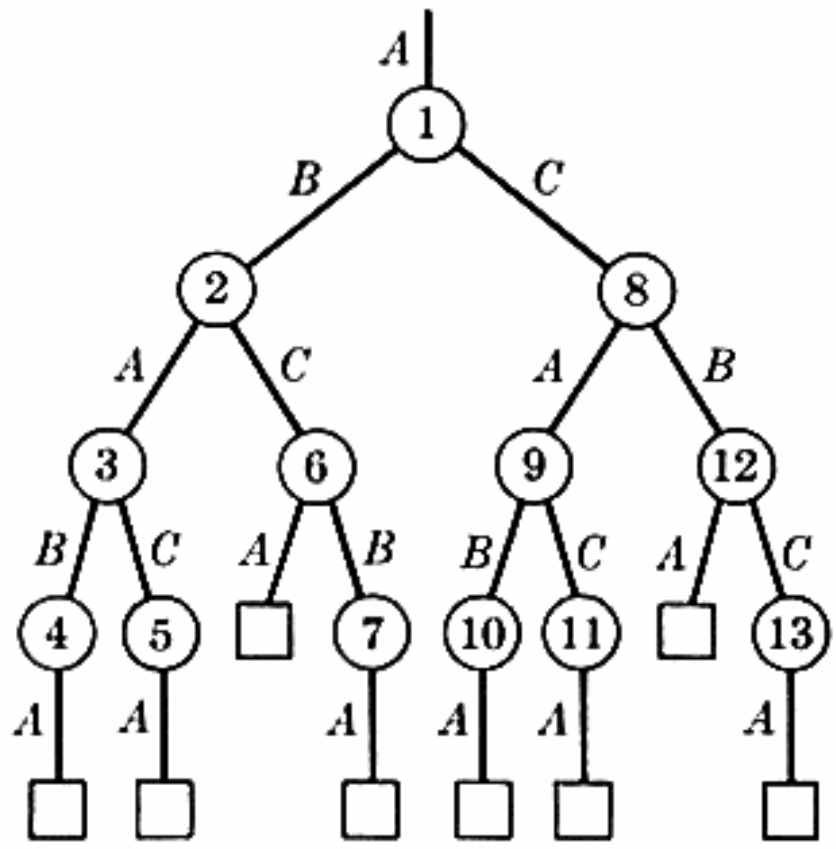
Step 3. Number the internal nodes of the tree in *preorder*. The result will be a labeling satisfying conditions (a), (b), (c).

For example, if we start with the tree



(11)

and three tapes, this procedure might assign labels as follows:



(12)

It is not difficult to verify that Karp's construction satisfies the "last formed, first grown on" discipline, because of the nature of preorder (see exercise 12).

Note that the result of this construction is a merge pattern for which all of the initial runs appear on tape A. This suggests the following distribution and sorting scheme which we may call the *preorder merge:*

P1. Distribute initial runs onto tape A until the input is exhausted. Let S be the total number of initial runs.

P2. Carry out the above construction, using a minimum-path-length $(T - 1)$-ary tree with S external nodes, obtaining a T-lifo tree whose external path length is within S of the lower bound in (9).

P3. Merge the runs according to this pattern. ▮

This scheme will produce its output on any desired tape. But *it has one serious flaw*—does the reader see what will go wrong? The problem is that the merge pattern requires some of the runs initially on tape A to be ascending, and some to be descending, depending on whether the corresponding external node appears on an odd or an even level. This problem can be resolved without knowing S in advance, by copying runs which should be descending onto an auxiliary tape or tapes, just before they are needed. Then the total amount of processing, in terms of initial run lengths, comes to

$$S \log_{T-1} S + O(S). \tag{13}$$

Thus the preorder merge is definitely better than polyphase or cascade, as $S \to \infty$; indeed, it is asymptotically *optimum*, since (9) shows that $S \log_{T-1} S + O(S)$ is the best we could ever hope to achieve on T tapes. On the other hand, for the comparatively small values of S which usually arise in practice, the preorder merge is rather inefficient; polyphase or cascade methods are simpler and faster, when S is reasonably small. Perhaps it will be possible to invent a simple distribution-and-merge scheme that is competitive with polyphase and cascade for small S, and which is asymptotically optimum for large S.

The second set of exercises below shows how Karp has formulated the question of *read-forward* merging in a similar way. The theory turns out to be

rather more complicated in this case, although some very interesting results have been discovered.

EXERCISES—First Set

1. [*17*] It is often convenient, during read-forward merging, to mark the end of each run on tape by including an artificial "sentinel" record whose key is $+\infty$. How should this practice be modified, when reading backwards?

2. [*20*] Will the columns of an array like (1) always be nondecreasing, or is there a chance that we will have to "subtract" runs from some tape as we go from one level to the next?

▶ **3.** [*20*] Prove that the polyphase distribution method described in connection with (1) will always produce an A run on tape T1 when sorting is complete, if T1 originally starts with $ADA\cdots$ and T2 through T5 start with $DAD\cdots$.

4. [*22*] Is it a good idea to do read-backward polyphase merging after distributing all runs in *ascending* order, imagining all the "D" positions to be initially filled with dummies?

▶ **5.** [*23*] What formulas for the strings of merge numbers replace (8), (9), (10), and (11) of Section 5.4.2, when read-backward polyphase merging is used? Show the merge numbers for the fifth level distribution on six tapes, by drawing a diagram like Fig. 71(a).

6. [*07*] What is the vector representation of the merge pattern whose tree representation is (8)?

7. [*16*] Draw the tree representation for the read-backwards merge pattern defined by the following sequence of vectors:

$$
\begin{aligned}
v^{(33)} &= (\ 20,\ \ 9,\ \ 5) & y^{(22)} &= (+1,-1,+1) & y^{(10)} &= (+1,+1,-1)\\
y^{(33)} &= (+1,-1,+1) & y^{(21)} &= (-1,+1,+1) & y^{(9)} &= (+1,-1,+1)\\
y^{(32)} &= (+1,+1,-1) & y^{(20)} &= (+1,+1,-1) & y^{(8)} &= (+1,+1,-1)\\
y^{(31)} &= (+1,+1,-1) & y^{(19)} &= (-1,+1,+1) & y^{(7)} &= (+1,+1,-1)\\
y^{(30)} &= (+1,+1,-1) & y^{(18)} &= (+1,+1,-1) & y^{(6)} &= (+1,+1,-1)\\
y^{(29)} &= (+1,-1,+1) & y^{(17)} &= (+1,+1,-1) & y^{(5)} &= (-1,+1,+1)\\
y^{(28)} &= (-1,+1,+1) & y^{(16)} &= (+1,+1,-1) & y^{(4)} &= (+1,-1,+1)\\
y^{(27)} &= (+1,-1,+1) & y^{(15)} &= (+1,+1,-1) & y^{(3)} &= (-1,+1,+1)\\
y^{(26)} &= (+1,-1,+1) & y^{(14)} &= (+1,-1,+1) & y^{(2)} &= (+1,-1,+1)\\
y^{(25)} &= (+1,+1,-1) & y^{(13)} &= (+1,-1,+1) & y^{(1)} &= (-1,+1,+1)\\
y^{(24)} &= (+1,-1,+1) & y^{(12)} &= (-1,+1,+1) & y^{(0)} &= (\ 1,\ \ 0,\ \ 0)\\
y^{(23)} &= (+1,-1,+1) & y^{(11)} &= (+1,+1,-1)
\end{aligned}
$$

8. [*23*] Prove that (8) is an optimum way to merge, reading backwards, when $S = 7$ and $T = 4$, and that all methods which avoid one-way merging are inferior.

9. [*M22*] Prove the lower bound (9).

10. [*41*] Prepare a table of the exact values of $K_T(n)$, using a computer.

▸**11.** [*20*] True or false: Any read-backwards merge pattern that uses nothing but $(T-1)$-way merging must always have the runs alternating $ADAD\cdots$ on each tape; it will not work if two adjacent runs appear in the same order.

12. [*22*] Prove that Karp's preorder construction always yields a labeled tree satisfying conditions (a), (b), and (c).

13. [*16*] Make (12) more efficient, by removing as many of the one-way merges as possible so that preorder still gives a valid labeling of the internal nodes.

14. [*40*] Devise an algorithm which carries out the preorder merge without explicitly representing the tree in steps P2 and P3, using only $O(\log S)$ words of memory to control the merging pattern.

15. [*M39*] Karp's preorder construction in the text yields trees with one-way merges at several terminal nodes. Prove that when $T = 3$ it is possible to contruct asymptotically optimal 3-lifo trees in which two-way merging is used throughout.

In other words, let $\hat{K}_T(n)$ be the minimum external path length over all T-lifo trees with n external nodes, such that every internal node has degree $T-1$. Prove that $\hat{K}_3(n) = n\log_2 n + O(n)$.

16. [*M46*] In the notation of exercise 15, is $\hat{K}_T(n) = n\log_{T-1} n + O(n)$ for *all* $T \geq 3$, when $n \equiv 1$ (modulo $T-1$)?

▸**17.** [*28*] (Richard D. Pratt.) To achieve ascending order in a read-backward cascade merge, we could insist on an *even* number of merging passes; this suggests a technique of initial distribution which is somewhat different from Algorithm 5.4.3C. (a) Change 5.4.3–1 so that it shows only the perfect distributions which require an even number of merging passes. (b) Design an initial distribution scheme which interpolates between these perfect distributions. (Thus, if the number of initial runs falls between perfect distributions, it is desirable to merge some, but not all, of the runs twice, in order to reach a perfect distribution.)

18. [*46*] Suppose that T tape units are available, for some $T \geq 3$, and that T1 contains N records while the remaining tapes are empty. Is it possible to reverse the order of the records on T1 in less than $O(N \log N)$ steps, *without* reading backwards? (The operation is, of course, trivial if backwards reading is allowed.) Cf. exercise 5.2.5–14 for a class of such algorithms that *do* require order $N \log N$ steps.

EXERCISES—Second Set

The following exercises develop the theory of tape merging on read-forward tapes; in this case each tape acts as a queue instead of as a stack. A merge pattern can be represented as a sequence of vectors $y^{(m)} \ldots y^{(1)}y^{(0)}$ exactly as in the text, but when we convert the vector representation to a tree representation we change "last formed, first grown on" to "*first* formed, first grown on." Thus the invalid configurations (4) would be changed to

both  and  or both  and  . (4′)

A tree which can be labeled so as to represent a read-forward merge on T tapes is called T-fifo, analogous to the term "T-lifo" in the read-backwards case.

When tapes can be read backwards, they make very good stacks. But unfortunately they don't make very good general purpose queues. If we randomly write and read, in a first-in-first-out manner, we waste a lot of time moving from one part of the tape to another. Even worse, we will soon run off the end of the tape! We run into the same problem as the queue overrunning memory in 2.2.2–(4) and (5), but the solution in 2.2.2–(6) and (7) doesn't apply to tapes since they aren't circular loops. Therefore we shall call a tree *strongly T-fifo* if it can be labeled so that the corresponding merge pattern makes each tape follow the special queue discipline "write, rewind, read all, rewind; write, rewind, read all, rewind; etc."

▶ **19.** [*22*] (R. M. Karp.) Find a binary tree which is not 3-fifo.

▶ **20.** [*22*] Formulate the condition "strongly T-fifo" in terms of a fairly simple rule about invalid configurations of tape labels, analogous to (4′).

21. [*18*] Draw the tree representation for the read-forwards merge pattern defined by the vectors in exercise 7. Is this tree strongly 3-fifo?

22. [*28*] (R. M. Karp.) Show that the tree representations for polyphase and cascade merging, with perfect distributions, are exactly the same for both the read-backwards and the read-forwards case, except for the numbers which label the internal nodes. Find a larger class of vector representations of merging patterns for which this is true.

23. [*24*] (R. M. Karp.) Let us say that a segment $y^{(q)} \ldots y^{(r)}$ of a merge pattern is a *stage* if no output tape is subsequently used as an input tape, i.e. if there do not exist i, j, k with $q \geq i > k \geq r$ and $y_j^{(i)} = -1$ and $y_j^{(k)} = +1$. The purpose of this exercise is to prove that *cascade merge minimizes the number of stages*, over all merge patterns having the same number of tapes and initial runs.

It is convenient to define some notation. Let us write $v \rightarrow w$ if v and w are T-vectors such that there is a merge pattern which reduces w to v in its first stage. (Thus there is a merge pattern $y^{(m)} \ldots y^{(0)}$ such that $y^{(m)} \ldots y^{(l+1)}$ is a stage, $w = y^{(m)} + \cdots + y^{(0)}$, and $v = y^{(l)} + \cdots + y^{(0)}$.) Let us write $v \preceq w$ if v and w are T-vectors such that the sum of the largest k elements of v is $\leq$ the sum of the largest k elements of w, for $1 \leq k \leq T$. Thus, for example, $(2, 1, 2, 2, 2, 1) \preceq (1, 2, 3, 0, 3, 1)$, since $2 \leq 3$, $2+2 \leq 3+3, \ldots, 2+2+2+2+1+1 \leq 3+3+2+1+1+0$. Finally, if $v = (v_1, \ldots, v_T)$, let $C(v) = (s_T, s_{T-2}, s_{T-3}, \ldots, s_1, 0)$, where s_k is the sum of the largest k elements of v.

(a) Prove that $v \rightarrow C(v)$. (b) Prove that $v \preceq w$ implies $C(v) \preceq C(w)$. (c) Assuming the result of exercise 24, prove that cascade merge minimizes the number of stages.

24. [*M35*] In the notation of exercise 23, prove that $v \rightarrow w$ implies $w \preceq C(v)$.

25. [*M36*] (R. M. Karp.) Let us say that a segment $y^{(q)} \ldots y^{(r)}$ of a merge pattern is a *phase* if no tape is used both for input and for output, i.e., if there do not exist i, j, k with $q \geq i$, $k \geq r$ and $y_j^{(i)} = +1$ and $y_j^{(k)} = -1$. The purpose of this exercise is to investigate merge patterns that minimize the number of phases. We shall write $v \Rightarrow w$ if w can be reduced to v in one phase (cf. the similar notation introduced in exercise 23), and we let $D_k(v) = (s_k + t_{k+1}, s_k + t_{k+2}, \ldots, s_k + t_T, 0, \ldots, 0)$, where t_j denotes the jth largest element of v and $s_k = t_1 + \cdots + t_k$. (a) Prove that $v \Rightarrow D_k(v)$ for $1 \leq k < T$. (b) Prove that $v \preceq w$ implies $D_k(v) \preceq D_k(w)$, for $1 \leq k < T$. (c) Prove that $v \Rightarrow w$ implies $w \preceq D_k(v)$, for some k, $1 \leq k < T$. (d) Consequently,

a merge pattern which sorts the maximum number of initial runs on T tapes in q phases can be represented by a sequence of integers $k_1\, k_2 \ldots k_q$, such that the initial distribution is $D_{k_q}(\ldots(D_{k_2}(D_{k_1}(u)))\ldots)$, where $u = (1, 0, \ldots, 0)$. This minimum-phase strategy has a strongly T-fifo representation, and it also belongs to the class of patterns in exercise 22. When $T = 3$ it is the *polyphase* merge, and for $T = 4, 5, 6, 7$ it is a variation of the *balanced* merge.

26. [*M46*] (R. M. Karp.) Is the optimum sequence $k_1\, k_2 \ldots k_q$ mentioned in exercise 25 always equal to $1\lceil T/2\rceil\lfloor T/2\rfloor\lceil T/2\rceil\lfloor T/2\rfloor\ldots$, for all $T \geq 4$ and all sufficiently large q?

5.4.5. The Oscillating Sort

A somewhat different approach to merge sorting was introduced by Sheldon Sobel in *JACM* **9** (1962), 372–375. Instead of starting with a distribution pass where all the initial runs are dispersed to tapes, he proposed an algorithm which oscillates back and forth between distribution and merging, so that much of the sorting takes place before the input has been completely examined.

Suppose, for example, that there are five tapes available for merging. Sobel's method would sort 16 initial runs as follows:

	Operation	T1	T2	T3	T4	T5	"Cost"
Phase 1.	Distribute	A_1	A_1	A_1	A_1	—	4
Phase 2.	Merge	—	—	—	—	D_4	4
Phase 3.	Distribute	—	A_1	A_1	A_1	D_4A_1	4
Phase 4.	Merge	D_4	—	—	—	D_4	4
Phase 5.	Distribute	D_4A_1	—	A_1	A_1	D_4A_1	4
Phase 6.	Merge	D_4	D_4	—	—	D_4	4
Phase 7.	Distribute	D_4A_1	D_4A_1	—	A_1	D_4A_1	4
Phase 8.	Merge	D_4	D_4	D_4	—	D_4	4
Phase 9.	Merge	—	—	—	A_{16}	—	16

Here as in Section 5.4.4 we use A_r and D_r to stand respectively for ascending and descending runs of relative length r. The method begins by writing an initial run onto each of four tapes, and merges them (reading backwards) onto the fifth tape. Distribution resumes again, this time cyclically shifted one place to the right with respect to the tapes, and a second merge produces another run D_4. When four D_4's have been formed in this way, an additional merge creates A_{16}. We could go on to create three more A_{16}'s, merging them into a D_{64}, and so on until the input is exhausted. It isn't necessary to know the length of the input in advance.

When the number of initial runs, S, is 4^m, it is not difficult to see that this method processes each record exactly $m + 1$ times (once during the distribution and m times during a merge). When S is between 4^{m-1} and 4^m, we could assume that dummy runs are present, bringing S up to 4^m; hence the total sorting time

would essentially amount to $\lceil \log_4 S \rceil + 1$ passes over all the data. This is just what would be achieved by a balanced sort on *eight* tapes; in general, oscillating sort with T work tapes is equivalent to balanced merging with $2(T - 1)$ tapes, since it makes $\lceil \log_{T-1} S \rceil + 1$ passes over the data. When S is a power of $T - 1$, this is the best *any* T-tape method could possibly do, since it achieves the lower bound in Eq. 5.4.4–9. On the other hand when S is $(T - 1)^{m-1} + 1$, just one higher than a power of $T - 1$, the method wastes nearly a whole pass.

Exercise 2 shows how to eliminate part of this penalty for non-perfect-powers S, by using a special ending routine. A further refinement was discovered in 1966 by Dennis L. Bencher, who called his procedure the "criss-cross merge" [see H. Wedekind, *Datenorganisation* (Berlin: W. de Gruyter, 1970), 164–166; and *U.S. Patent 3540000* (November 10, 1970)]. The main idea is to delay merging until more knowledge of S has been gained. We shall discuss a slightly modified form of Bencher's original scheme.

This improved oscillating sort proceeds as follows:

	Operation	T1	T2	T3	T4	T5	"Cost"
Phase 1.	Distribute	—	A_1	A_1	A_1	A_1	4
Phase 2.	Distribute	—	A_1	A_1A_1	A_1A_1	A_1A_1	3
Phase 3.	Merge	D_4	—	A_1	A_1	A_1	4
Phase 4.	Distribute	D_4A_1	—	A_1	A_1A_1	A_1A_1	3
Phase 5.	Merge	D_4	D_4	—	A_1	A_1	4
Phase 6.	Distribute	D_4A_1	D_4A_1	—	A_1	A_1A_1	3
Phase 7.	Merge	D_4	D_4	D_4	—	A_1	4
Phase 8.	Distribute	D_4A_1	D_4A_1	D_4A_1	—	A_1	3
Phase 9.	Merge	D_4	D_4	D_4	D_4	–	4

We do not merge the D_4's into an A_{16} at this point (unless the input happens to be exhausted); only after building up to

Phase 15.	Merge	D_4D_4	D_4D_4	D_4D_4	D_4	—	4

will we get

Phase 16.	Merge	D_4	D_4	D_4	—	A_{16}	16.

The second A_{16} will occur after three more D_4's have been made,

Phase 22.	Merge	D_4D_4	D_4D_4	D_4	—	$A_{16}D_4$	4
Phase 23.	Merge	D_4	D_4	—	A_{16}	A_{16}	16

and so on (cf. phases 1–5). The advantage of Bencher's scheme can be seen for example if there are only five initial runs: Oscillating sort as modified in exercise 2 would do a four-way merge (in Phase 2) followed by a two-way merge, for a total cost of $4 + 4 + 1 + 5 = 14$, while Bencher's scheme would do a two-way merge (in Phase 3) followed by a four-way merge, for a total cost of

$4 + 1 + 2 + 5 = 12$. (Both methods also involve a small additional cost, namely one unit of rewind before the final merge.)

A precise description of Bencher's method appears in Algorithm B below. Unfortunately it seems to be a procedure which is harder to understand than to code; it is much easier to explain the technique to a computer than to a computer scientist! This is partly because it is a recursive method that has been expressed in iterative form and then "optimized" somewhat; the reader may find it necessary to trace through the operation of this algorithm several times before he really discovers what is going on.

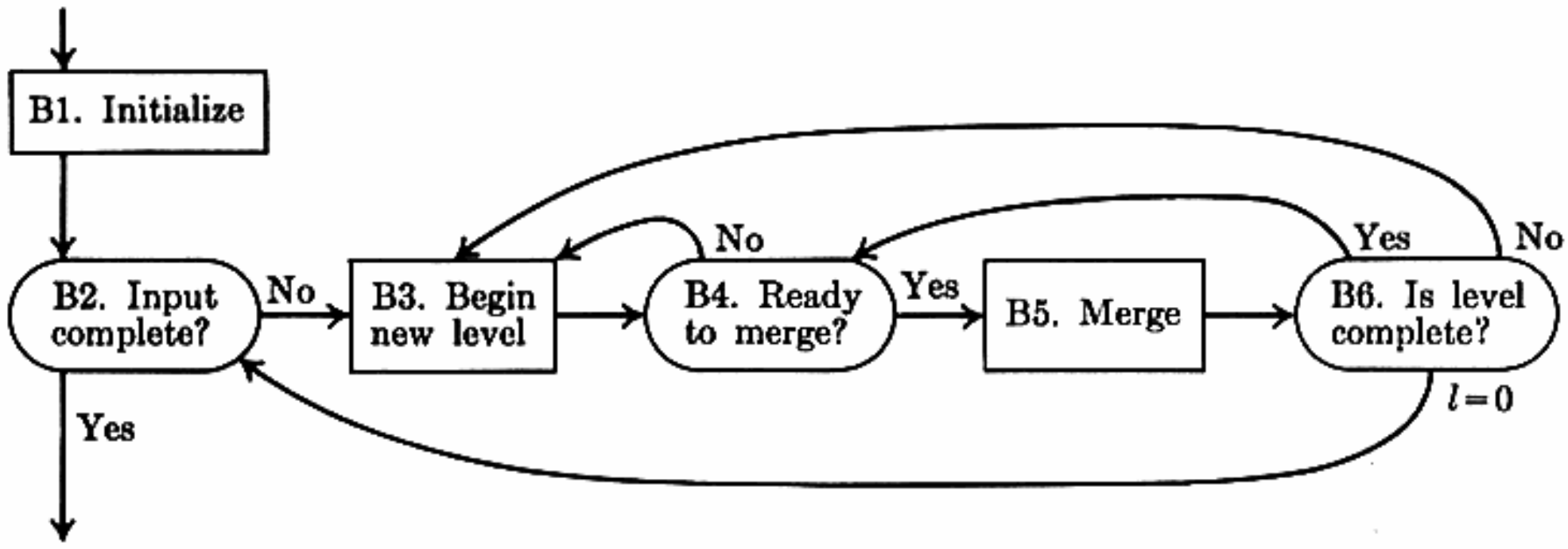


Fig. 77. Oscillating sort, with a "criss-cross" distribution.

Algorithm B (*Oscillating sort with "criss-cross" distribution*). This algorithm takes initial runs and disperses them to tapes, occasionally interrupting the distribution process in order to merge some of the tape contents. The algorithm uses P-way merging, assuming that $T = P + 1 \geq 3$ tape units are available (*not* counting the unit which may be necessary to hold the input data). The tape units must allow reading in both forward and backward directions, and they are designated by the numbers $0, 1, \ldots, P$. The following tables are maintained:

$\texttt{D}[j]$, $0 \leq j \leq P$ — Number of dummy runs assumed to be present at the end of tape j.

$\texttt{A}[l, j]$, $0 \leq l \leq L$, $0 \leq j \leq P$ — Here L is a number which is large enough that at most P^{L+1} initial runs will be input. When $\texttt{A}[l, j] = k \geq 0$, a run of nominal length P^k is present on tape j, corresponding to "level l" of the algorithm's operation. This run is ascending if k is even, descending if k is odd. When $\texttt{A}[l, j] = -1$, level l does not use tape j.

The statement "Write an initial run on tape j" is an abbreviation for the following operations:

> Set $\texttt{A}[l, j] \leftarrow 0$. If the input is exhausted, increase $\texttt{D}[j]$ by 1; otherwise write an initial run (in ascending order) onto tape j.

The statement "Merge to tape j" is an abbreviation for the following operations:

If $\texttt{D}[i] > 0$ for all $i \neq j$, decrease $\texttt{D}[i]$ by 1 for all $i \neq j$ and increase $\texttt{D}[j]$ by 1. Otherwise merge one run to tape j, from all tapes $i \neq j$ such that $\texttt{D}[i] = 0$, and decrease $\texttt{D}[i]$ by 1 for all other $i \neq j$.

B1. [Initialize.] Set $\texttt{D}[j] \leftarrow 0$ for $0 \leq j \leq P$. Then write an initial run on tape j, for $1 \leq j \leq P$. Set $\texttt{A}[0, 0] \leftarrow -1$, $l \leftarrow 0$, $q \leftarrow 0$.

B2. [Input complete?] (At this point tape q is empty and the other tapes contain at most one run each.) If there is more input, go on to step B3. But if the input is exhausted, rewind all tapes $j \neq q$ such that $\texttt{A}[0, j]$ is even; then merge to tape q, reading forward on tapes just rewound, and reading backward on the other tapes. This completes the sort, with the output in ascending order on tape q.

B3. [Begin new level.] Set $l \leftarrow l + 1$, $r \leftarrow q$, $s \leftarrow 0$, and $q \leftarrow (q + 1) \bmod T$. Write an initial run on tape $(q + j) \bmod T$, for $1 \leq j \leq T - 2$. (Thus an initial run is written onto each tape except tapes q and r.) Set $\texttt{A}[l, q] \leftarrow -1$ and $\texttt{A}[l, r] \leftarrow -1$.

B4. [Ready to merge?] If $\texttt{A}[l - 1, q] \neq s$, go back to step B3.

B5. [Merge.] (At this point $\texttt{A}[l - 1, q] = \texttt{A}[l, j] = s$ for all $j \neq q$, $j \neq r$.) Merge to tape r. (See the definition of this operation above.) Then set $s \leftarrow s + 1$, $l \leftarrow l - 1$, $\texttt{A}[l, r] \leftarrow s$, and $\texttt{A}[l, q] \leftarrow -1$. Set $r \leftarrow (2q - r) \bmod T$. (In general, we have $r = (q - 1) \bmod T$ when s is even, $r = (q + 1) \bmod T$ when s is odd.)

B6. [Is level complete?] If $l = 0$, go to B2. Otherwise if $\texttt{A}[l, j] = s$ for all $j \neq q$ and $j \neq r$, go to B4. Otherwise return to B3. ∎

We can use a "recursion induction" style of proof to show that this algorithm is valid, just as we have done for Algorithm 2.3.1T. Suppose that we begin at step B3 with $l = l_0$, $q = q_0$, $s_+ = \texttt{A}[l_0, (q_0 + 1) \bmod T]$, and $s_- = \texttt{A}[l_0, (q_0 - 1) \bmod T]$; and assume furthermore that either $s_+ = 0$ or $s_- = 1$ or $s_+ = 2$ or $s_- = 3$ or $\cdots$. It is possible to verify by induction that the algorithm will eventually get to step B5 without changing rows 0 through l_0 of A, and with $l = l_0 + 1$, $q = q_0 \pm 1$, $r = q_0$, and $s = s_+$ or s_-, where we choose the $+$ sign if $s_+ = 0$ or ($s_+ = 2$ and $s_- \neq 1$) or ($s_+ = 4$ and $s_- \neq 1, 3$) or $\cdots$, and we choose the $-$ sign if ($s_- = 1$ and $s_+ \neq 0$) or ($s_- = 3$ and $s_+ \neq 0, 2$) or $\cdots$. The proof sketched here is not very elegant, but the algorithm has been stated in a form more suited to implementation than to verification.

Figure 78 shows the efficiency of Algorithm B, in terms of the average number of times each record is merged as a function of the number S of initial runs, assuming that the initial runs are approximately equal in length. (Corresponding graphs for polyphase and cascade sort have appeared in Figs. 70 and 74.) A slight improvement, mentioned in exercise 3, has been used in preparing this chart.

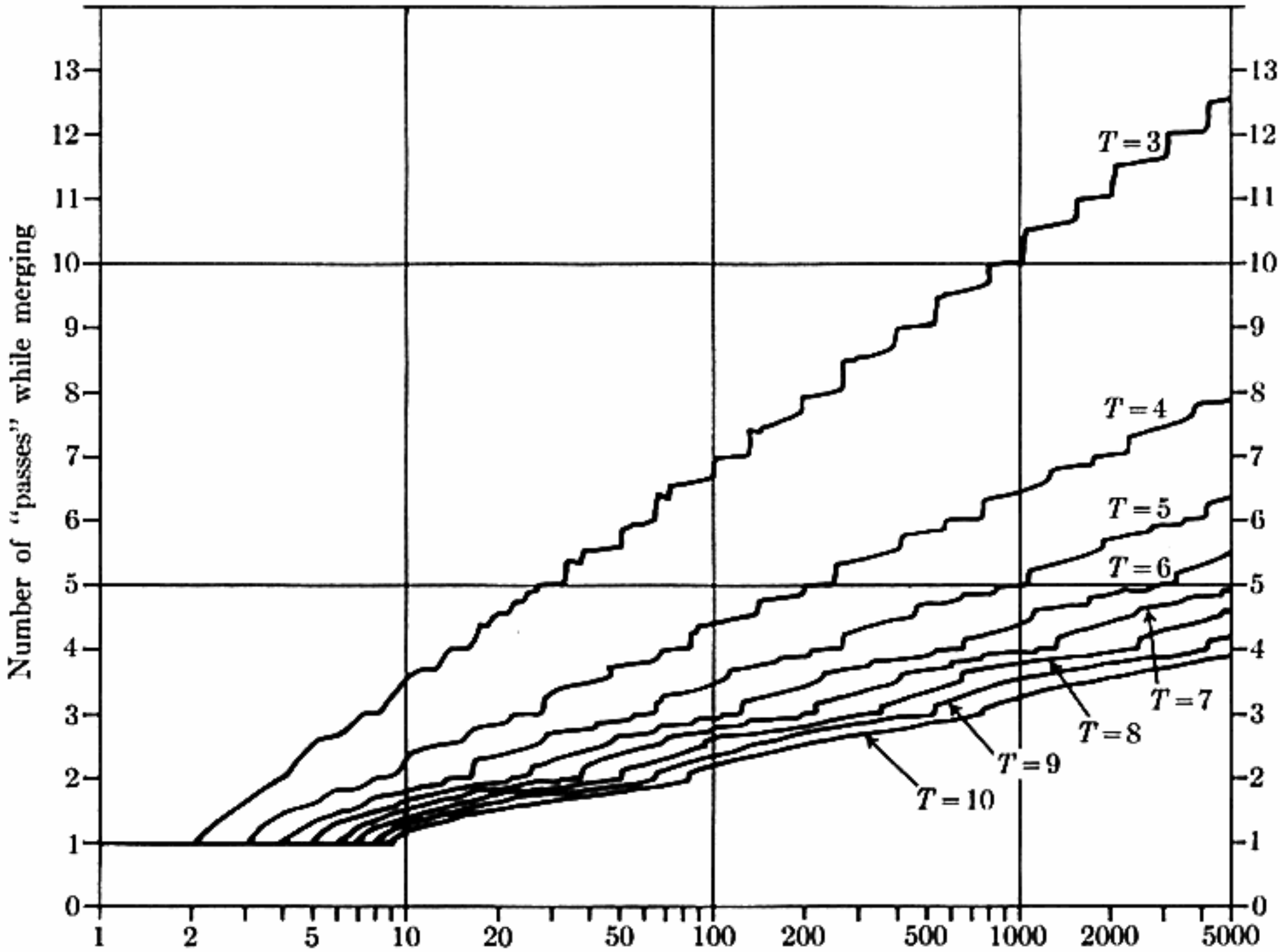


Fig. 78. Efficiency of oscillating sort, using the technique of Algorithm B and exercise 3.

Reading forwards. The oscillating sort pattern appears to require a read-backwards capability, since we need to store long runs somewhere as we merge newly-input short runs. However, M. A. Goetz [*Proc. AFIPS Spring Jt. Comp. Conf.* **25** (1964), 599–607] has discovered a way to perform an oscillating sort using only forward reading and simple rewinding. His method is radically different from the other schemes we have seen in this chapter, because

a) Data is sometimes written at the front of the tape, with the understanding that the existing data in the *middle* of the tape is not destroyed.

b) All initial strings have a fixed maximum length.

Condition (a) violates the "first-in-first-out" property we have assumed to be characteristic of forward reading, but it can be implemented reliably if a sufficient amount of blank tape is left between runs and if "parity errors" are ignored at appropriate times. Condition (b) tends to be somewhat incompatible with an efficient use of replacement selection.

Goetz's read-forward oscillating sort has the somewhat dubious distinction of being one of the first algorithms to be patented as an algorithm instead of as a physical device [*U.S. Patent 3380029* (April 23, 1968)]; unless successfully contested, this means it is illegal to use the algorithm in a program without

permission of the patentee. Bencher's read-backwards oscillating sort technique was patented by IBM several years later. [Thus, we have reached the end of the era when the joy of discovering a new algorithm was satisfaction enough! Since programming is strongly analogous to the fabrication of a machine, and since computer programs are now worth money, patented algorithms are inevitable. Of course the specter of people keeping new techniques completely secret is far worse than the public appearance of algorithms which are proprietary for a limited time.]

The central idea in Goetz's method is to arrange things so that each tape begins with a run of relative length 1, followed by one of relative length P, then P^2, etc. For example, when $T = 5$ the sort begins as follows, using "." to indicate the current position of the read-write head on each tape:

	Operation	T1	T2	T3	T4	T5	"Cost"	Remarks
Phase 1.	Distribute	$.A_1$	$.A_1$	$.A_1$	$.A_1$	$A_1.$	5	[T5 not rewound]
Phase 2.	Merge	$\aleph_1.$	$\aleph_1.$	$\aleph_1.$	$\aleph_1.$	$A_1\,A_4.$	4	[Now rewind all]
Phase 3.	Distribute	$.A_1$	$.A_1$	$.A_1$	$A_1.$	$.A_1\,A_4$	4	[T4 not rewound]
Phase 4.	Merge	$\aleph_1.$	$\aleph_1.$	$\aleph_1.$	$A_1\,A_4.$	$\aleph_1.A_4$	4	[Now rewind all]
Phase 5.	Distribute	$.A_1$	$.A_1$	$A_1.$	$.A_1\,A_4$	$.A_1\,A_4$	4	[T3 not rewound]
Phase 6.	Merge	$\aleph_1.$	$\aleph_1.$	$A_1\,A_4.$	$\aleph_1.A_4$	$\aleph_1.A_4$	4	[Now rewind all]
Phase 7.	Distribute	$.A_1$	$A_1.$	$.A_1\,A_4$	$.A_1\,A_4$	$.A_1\,A_4$	4	[T2 not rewound]
Phase 8.	Merge	$\aleph_1.$	$A_1\,A_4.$	$\aleph_1.A_4$	$\aleph_1.A_4$	$\aleph_1.A_4$	4	[Now rewind all]
Phase 9.	Distribute	$A_1.$	$.A_1\,A_4$	$.A_1\,A_4$	$.A_1\,A_4$	$.A_1\,A_4$	4	[T1 not rewound]
Phase 10.	Merge	$A_1A_4.$	$\aleph_1.A_4$	$\aleph_1.A_4$	$\aleph_1.A_4$	$\aleph_1.A_4$	4	[No rewinding]
Phase 11.	Merge	$A_1A_4A_{16}.$	$\aleph_1\,\aleph_4.$	$\aleph_1\,\aleph_4.$	$\aleph_1\,\aleph_4.$	$\aleph_1\,\aleph_4.$	16	[Now rewind all]

And so on. During Phase 1, T1 was rewinding while T2 was receiving its input, then T2 was rewinding while T3 was receiving input, etc. Eventually, when the input is exhausted, dummy runs will start to appear, and it sometimes becomes necessary to imagine that they have been written explicitly on the tape at full length. For example, if $S = 18$, the A_1's on T4 and T5 would be dummies during Phase 9; we would have to skip forward on T4 and T5 while merging from T2 and T3 to T1 during Phase 10, because we have to get to the A_4's on T4 and T5 in preparation for Phase 11. On the other hand, the dummy A_1 on T1 need not appear explicitly. Thus the "endgame" is a bit tricky.

A further example of this method appears in the next section.

EXERCISES

1. [*22*] The text illustrates Sobel's original oscillating sort for $T = 5$ and $S = 16$. Give a precise specification of an algorithm which generalizes the procedure, sorting $S = P^L$ initial runs on $T = P + 1 \geq 3$ tapes. Strive for an algorithm which can be described very simply.

2. [*24*] If $S = 6$ in Sobel's original method, we could pretend that $S = 16$ and that 11 dummy runs were present. Then Phase 3 in the text's example would put dummy runs A_0 on T4 and T5; Phase 4 would merge the A_1's on T2 and T3 into a D_2 on T1; Phases 5–8 would do nothing; and Phase 9 would produce A_6 on T4. It

would be better to rewind T2 and T3 just after Phase 3, then to produce A_6 immediately on T4 by three-way merging.

Show how to modify the algorithm of exercise 1, so that an improved ending like this is obtained when S is not a perfect power of P.

▶ **3.** [*24*] Prepare a chart showing the behavior of Algorithm B when $T = 3$, assuming that there are nine initial runs. Show that the procedure is obviously inefficient in one place, and prescribe corrections to Algorithm B which will remedy the situation.

4. [*21*] Step B3 sets both A[l, q] and A[l, r] to -1. Show that one of these two operations is always superfluous, since the corresponding A table entry is never looked at.

5. [*M25*] Let S be the number of initial runs present in the input to Algorithm B. Which values of S require *no rewinding* in step B2?

5.4.6. Practical Considerations for Tape Merging

Here comes the nitty-gritty: Now that we have discussed the various families of merge patterns, it is time to see how they actually apply to real configurations of computers and magnetic tapes, and to compare them in a meaningful way. Our study of internal sorting showed that we can't adequately judge the efficiency of a sorting method merely by counting the number of comparisons it performs; similarly we can't properly evaluate an external sorting method by simply knowing the number of passes it makes over the data.

In this section we shall discuss the characteristics of typical tape units, and the way they affect initial distribution and merging. In particular we shall study some schemes for buffer allocation, and the corresponding effects on running time. We also shall consider briefly the construction of *sort generator* programs.

How tape works. There is considerable variation in the characteristics of tape units provided by different manufacturers. For convenience, we shall define a hypothetical MIXT tape unit, which is reasonably typical of the equipment being manufactured at the time this book was written. By studying how to design a sorting algorithm for MIXT tapes, it will be easier to understand how to deal with other particular tape units.

MIXT reads and writes 800 characters per inch of tape, at a rate of 75 inches per second. This means that one character is read or written every $\frac{1}{60}$ ms, i.e. $16\frac{2}{3}$ microseconds, when the tape is active. [Actual tape units which are currently being sold have densities ranging from 200 to 1600 characters per inch, and tape speeds ranging from $37\frac{1}{2}$ to 150 inches per second, so their effective speed varies from $\frac{1}{8}$ to 4 times as fast as MIXT. In practice, much sorting is done in commercial environments on relatively small and inexpensive equipment which is slower than that considered here. On the other hand, tape units may soon change drastically, making the present assumptions obsolete. Our main concern here is not to obtain particular answers; it is to learn how to combine theory and practice in a reasonable way.]

One of the important considerations to keep in mind is that individual tapes aren't infinitely long. Each reel contains 2400 feet of tape or less, hence

there is room for at most 23,000,000 or so characters per reel of MIXT tape; and it takes about 23000000/3600000 $\approx$ 6.4 minutes to read them all. If larger files must be sorted, it is generally best to sort one reelfull at a time, and then to merge the individual sorted reels, in order to avoid excess tape handling. This means that the number of initial runs, S, actually present in the merge patterns we have been studying is never extremely large. We will never find $S > 5000$, even with a very small internal memory which produces initial runs only 5000 characters long. Consequently the formulas which give asymptotic efficiency of the algorithms as $S \to \infty$ are primarily of academic interest.

Data appears on tape in *blocks* (Fig. 79), and each read/write instruction transmits a single block. Blocks are often called "records," but we shall avoid this terminology because it conflicts with the fact that we are sorting a file of "records" in another sense. On many of the early sorting programs written during the 1950's, this distinction was unnecessary, since one record was written per block; but we shall see that it is usually advantageous to have quite a few records in every block on the tape.

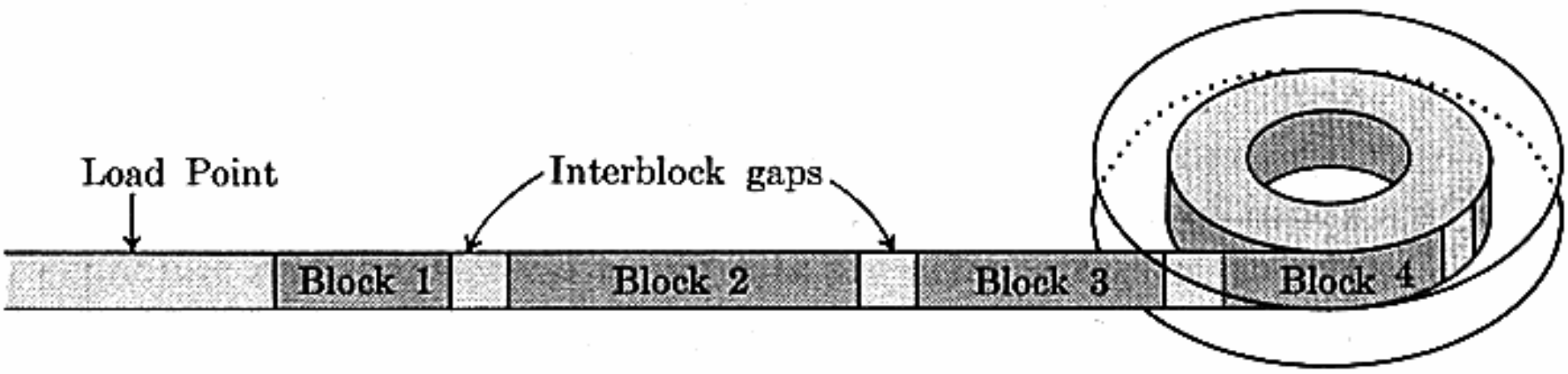


Fig. 79. Magnetic tape with variable-size blocks.

An *interblock gap*, 480 character positions long, appears between adjacent blocks, in order to allow the tape to stop and to start between individual read or write commands. The effect of interblock gaps is to decrease the number of characters per reel of tape, depending on the number of characters per block (see Fig. 80); and the average number of characters transmitted per second decreases in the same way, since tape moves at a fairly constant speed.

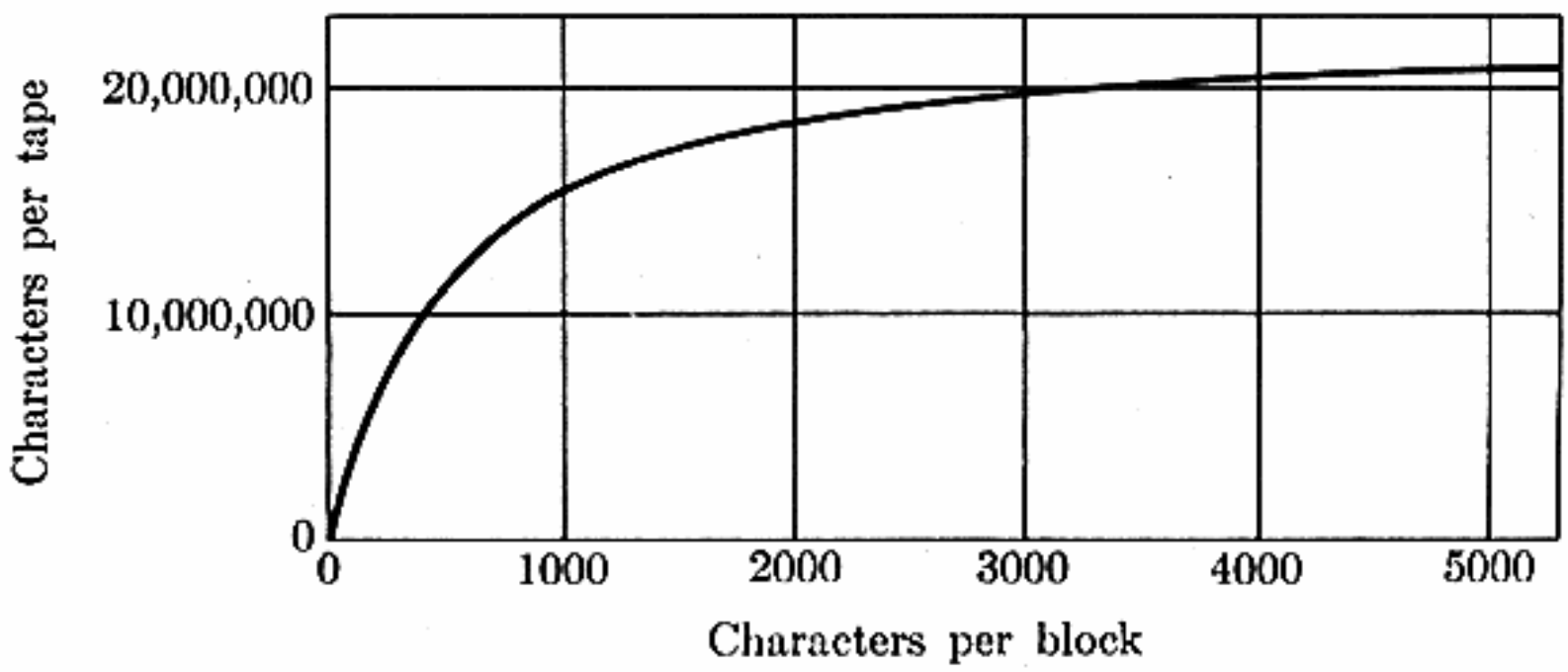


Fig. 80. The number of characters per reel of MIXT tape, as a function of the block size.

Many "old fashioned" computers have fixed block sizes which are rather small. For example, MIX as defined in Chapter 1 always reads and writes 100-word blocks; since this represents about 500 characters per block, and 480 characters per gap, almost half the tape is wasted! Most machines now allow the block size to be variable, and so we shall discuss the choice of appropriate block sizes below.

At the end of a read or write operation, the tape unit "coasts" at full speed over the first 66 characters (or so) of the gap. If the next operation for the same tape is initiated during this time, the tape motion continues without interruption. But if the next operation doesn't come soon enough, the tape will stop and it will also require some time to accelerate to full speed on the next operation. The combined stop-start time delay is 5 ms, 2 for the stop and 3 for the start (see Fig. 81). Thus if we just miss the chance to have continuous full-speed reading, the effect on running time is essentially the same as if there were 780 characters instead of 480 in the interblock gap.

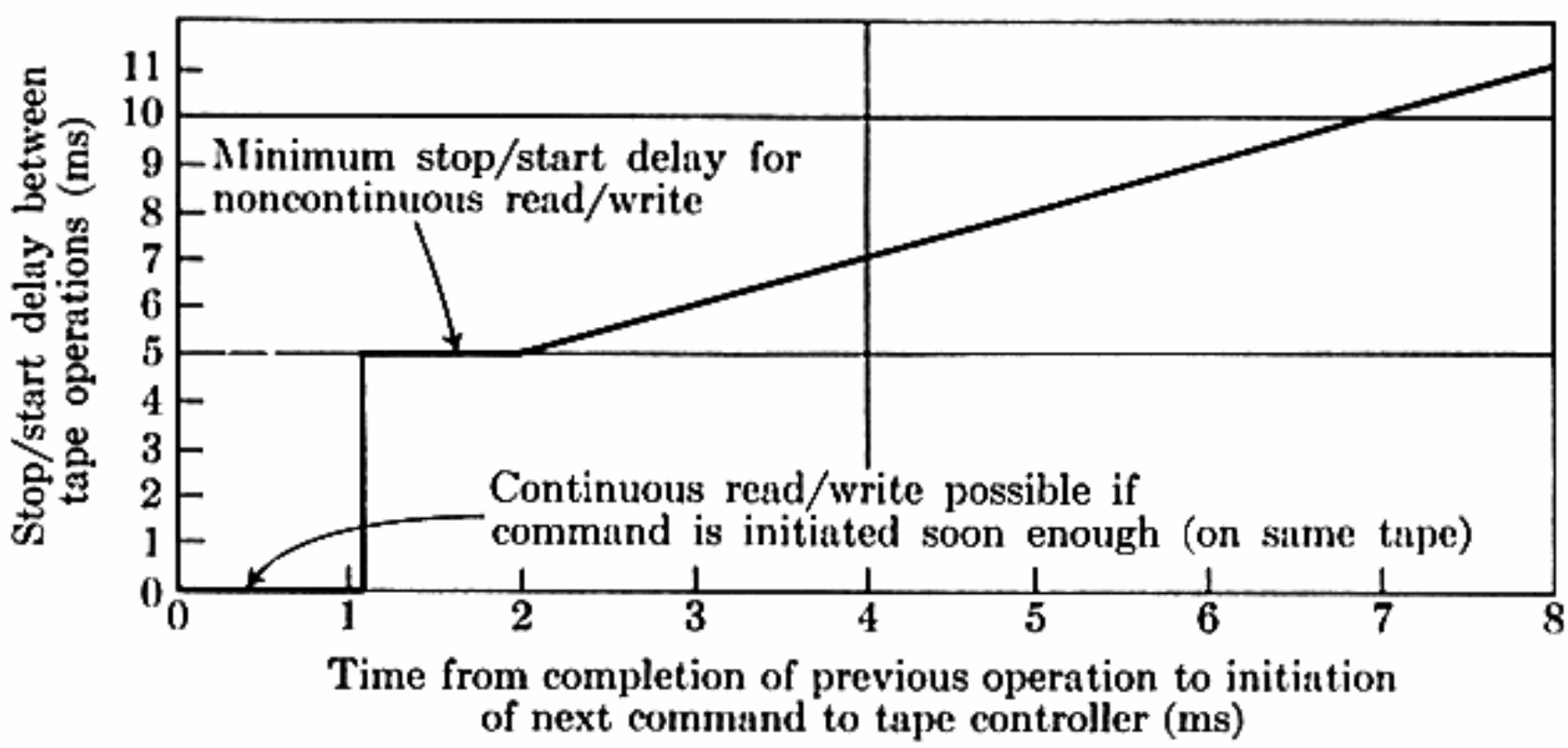


Fig. 81. How to compute the stop/start delay time. (This gets added to the time used for reading or writing the blocks and the gaps.)

Now let us consider the operation of *rewinding*. Unfortunately, it is generally difficult to characterize the exact time needed to rewind over a given number n of characters. On some machines there is a "high-speed rewind" which applies only when n is greater than 5 million or so; for smaller values of n, rewinding goes at normal read/write speed. On other machines there is a special motor which is used on all rewind operations; it gradually accelerates the tape reel to a certain number of revolutions per minute, then puts on the brakes when it is time to stop, and the actual tape speed varies with the fullness of the reel. For simplicity, we shall assume that MIXT requires max (30, $n/150$) ms to rewind over n character positions (including gaps), roughly two-fifths as long as it took to write them. This is a reasonably good approximation to the behavior of many actual tape units, where the ratio of read/write time to rewind time is generally between 2 and 3, but it does not adequately model the

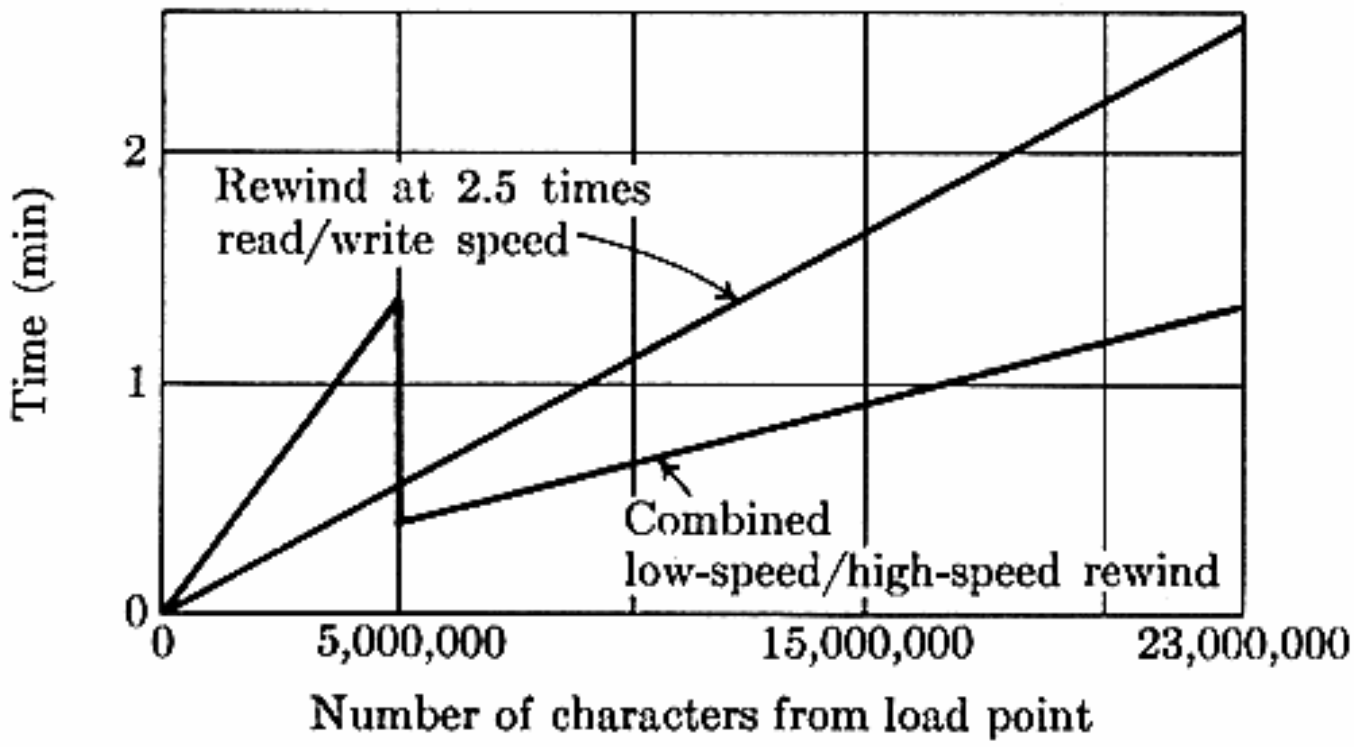


Fig. 82. Approximate running time for two commonly used rewind techniques.

effect of combined low-speed and high-speed rewind which is present on many other machines. (See Fig. 82.)

Initial loading and/or rewinding will position a tape at "load point", and an extra 110 ms are necessary for any read or write operation initiated at load point. When the tape is not at load point, it may be read backwards; an extra 32 ms is added to the time of any backward operation following a forward operation or any forward operation following a backward one.

Finally we must consider the allowable simultaneity of input and output. For economic reasons, several tape units are often attached to a *tape controller*, which is able to deal with only one or two of its units at a time because there are a limited number of data paths between it and the computer. Sometimes controllers are not capable of dealing with more than one unit at a time, but they are often able to read on one unit while writing on another. Somewhat rarer are controllers which can read on two units simultaneously; and the author has never seen a controller that can write on two units at once. Rewinding is a special case: 30 ms after a `MIXT` rewind has been initiated, the tape unit is "detached" from its controller, which becomes free to access other units. In this way a very large number of tape units could be rewinding simultaneously.

Nearly all machines allow input and output to proceed concurrently with computing, although many computers run 20 to 40 percent more slowly when input/output is in progress due to "cycle sharing."

Merging revisited. Let us now look again at the process of P-way merging, with an emphasis on input and output activities, assuming that $P + 1$ tape units are being used for the input files and the output file. Our goal is to overlap the input/output operations as much as possible with each other and with the computations of the program, so that the overall merging time is minimized.

It is instructive to consider the following special case, in which serious restrictions are placed on the amount of simultaneity possible. Suppose that

a) At most one tape may be written on at any one time.
b) At most one tape may be read from at any one time.

c) Reading, writing, and computing may take place simultaneously only when the read and write operations have been initiated simultaneously.

It turns out that a system of $2P$ input buffers and 2 output buffers is sufficient to keep the tape moving at essentially its maximum speed, even though these three restrictions are imposed, unless the computer is unusually slow. Note that (a) is not really a restriction, since there is only one output tape. Furthermore the amount of input is equal to the amount of output, so there is only one tape being read, on the average, at any given time; if condition (b) is not satisfied, there will necessarily be periods when no input at all is occurring. Thus we can minimize the merging time if we keep the output tape busy.

An important technique called prediction or *forecasting* leads to the desired effect. While we are doing a P-way merge, we generally have P "current input buffers," which are being used as the source of data; some of them are more full than others, depending on how much of their data has already been scanned. If all of them become empty at about the same time, we will need to do a lot of reading before we can proceed further, unless we have foreseen this eventuality in advance. Fortunately it is always possible to tell which buffer will empty first, by simply looking at the *last* record in each buffer. The buffer whose last record has the smallest key will always be the first one empty, regardless of the values of any other keys; so we always know which file should be the source of our next input command. The following algorithm spells out this principle in detail.

Algorithm F (*Forecasting with floating buffers*). This algorithm controls the buffering during a P-way merge of long input files, for $P \geq 2$. Assume that the input tapes and files are numbered $1, 2, \ldots, P$. The algorithm uses $2P$ input buffers `I`$[1], \ldots,$ `I`$[2P]$; two output buffers `O`$[0]$ and `O`$[1]$; and the following auxiliary tables:

`A`$[j]$, $1 \leq j \leq 2P$: 0 if `I`$[j]$ is available for input, 1 otherwise;

`B`$[i]$, $1 \leq i \leq P$: The buffer containing the last block read so far from file i;

`C`$[i]$, $1 \leq i \leq P$: The buffer currently being used for the input from file i;

`L`$[i]$, $1 \leq i \leq P$: The last key read so far from file i;

`S`$[j]$, $1 \leq j \leq 2P$: The buffer to use when `I`$[j]$ becomes empty.

The algorithm described here does not terminate; an appropriate way to "shut it off" is discussed below.

F1. [Initialize.] Read the first block from tape i into buffer `I`$[i]$, set `A`$[i] \leftarrow 1$, `A`$[P + i] \leftarrow 0$, `B`$[i] \leftarrow i$, `C`$[i] \leftarrow i$, and set `L`$[i]$ to the key of the final record in `I`$[i]$, for $1 \leq i \leq P$. Then find m such that `L`$[m] = \min \{$`L`$[1], \ldots,$ `L`$[P]\}$; and set $t \leftarrow 0$, $k \leftarrow P + 1$. Begin to read from tape m into buffer `I`$[k]$.

F2. [Merge.] Merge records from buffers `I`$[$`C`$[1]], \ldots,$ `I`$[$`C`$[P]]$ to `O`$[t]$, until `O`$[t]$ is full. If during this process an input buffer, say `I`$[$`C`$[i]]$, becomes empty and `O`$[t]$ is not yet full, set `A`$[$`C`$[i]] \leftarrow 0$, `C`$[i] \leftarrow$ `S`$[$`C`$[i]]$, and continue to merge.

F3. [I/O complete.] Wait until the previous read (or read/write) operation is

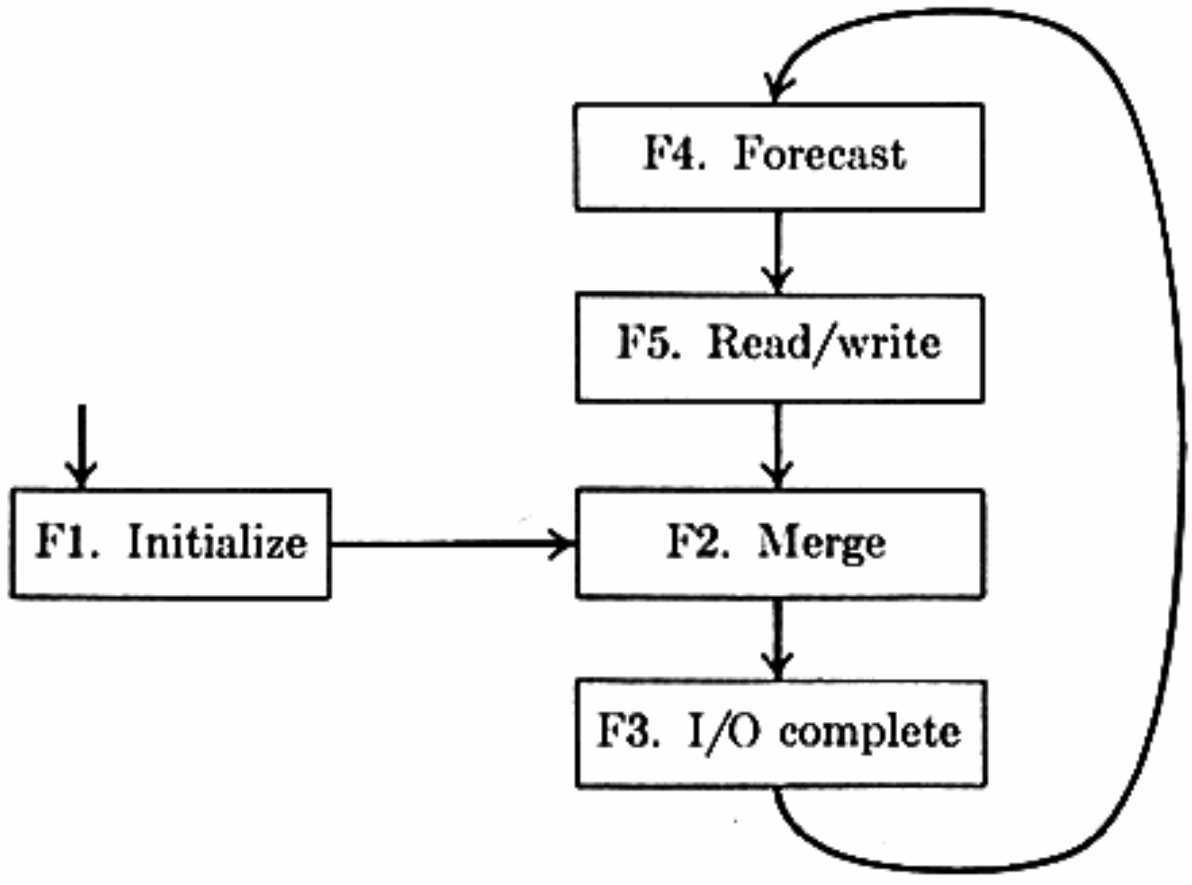


Fig. 83. Forecasting with floating buffers.

complete. Then set A[k] ← 1, S[B[m]] ← k, B[m] ← k, and set L[m] to the key of the final record in I[k].

F4. [Forecast.] Find m such that L[m] = min {L[1], . . . , L[P]}, and find k such that A[k] = 0.

F5. [Read/write.] Begin to read from tape m into buffer I[k], and to write from buffer O[t] onto the output tape. Then set $t \leftarrow 1 - t$ and return to F2. ▮

The example in Fig. 84 shows how forecasting works, when $P = 2$, assuming that each block on tape contains only two records. The input buffer contents are illustrated each time we get to the beginning of step F2. Algorithm F essentially forms P *queues of buffers*, with C[i] pointing to the front and B[i]

Fig. 84. Buffer queuing, according to Algorithm F.

to the rear of the ith queue, and with S[j] pointing to the successor of buffer I[j]; these pointers are shown as arrows in Fig. 84. Line 1 illustrates the state of affairs after initialization: There is one buffer for each input file, and another block is being read from File 1 (since 03 < 05). Line 2 shows the status of things after the first block has been merged: we are outputting a block containing "01 02", and inputting the next block from File 2 (since 05 < 09). Note that in line 3, three of the four input buffers are essentially committed to File 2, since we are reading from that file and we already have a full buffer and a partly full buffer in its queue. This "floating buffer" arrangement is an important feature of Algorithm F, since we would be unable to proceed in line 4 if we had chosen File 1 instead of File 2 for the input on line 3.

In order to prove that Algorithm F is valid, we must show two things:

i) There is always an input buffer available (i.e., we can always find a k in step F4).

ii) If an input buffer is exhausted while merging, its successor is already present in memory (i.e., S[C[i]] is meaningful in step F2).

Suppose (i) is false, so that all buffers are unavailable at some point when we reach step F4. Each time we get to that step, the total amount of unprocessed data among all the buffers is exactly P bufferloads, i.e. just enough data to fill P buffers if it were relocated, since we are inputting and outputting data at the same rate. Some of the buffers are only partially full; but at most one buffer for each file is partially full, so at most P buffers are partly full. By hypothesis all $2P$ of the buffers are unavailable, so at least P of them must be completely full. This can happen only if P are full and P are empty, otherwise we would have too much data. But at most one buffer can be unavailable and empty at any one time; hence (i) cannot be false.

Suppose (ii) is false, so that we have no unprocessed records in memory, for some file, but the current output buffer is not yet full. By the principle of forecasting, we must have no more than one block of data for all the other files, since we do not read in a block for a file unless that block will be needed before the buffers on any other file are exhausted. Therefore the total number of unprocessed records amounts to at most $P - 1$ blocks; adding the unfilled output buffer leads to less than P bufferloads of data in memory, a contradiction.

This argument establishes the validity of Algorithm F; and it also indicates the possibility of pathological circumstances under which the algorithm just barely avoids disaster. An important subtlety that we have not mentioned, regarding the possibility of equal keys, is discussed in exercise 5.

One way to terminate Algorithm F gracefully is to set L[p] to ∞ in step F3 if the block just read is the last of a run. (It is customary to indicate the end of a run in some special way.) After all of the data on all of the files has been read, we will eventually find all of the L's equal to ∞ in step F4; then it is usually possible to begin reading the first blocks of the next run on each file, beginning initialization of the next merge phase as the final $P + 1$ blocks are output.

Thus we can keep the output tape going at essentially full speed, without reading more than one tape at a time. An exception to this rule occurs in step F1, where it would be beneficial to read several tapes at once in order to get things going in the beginning; but step F1 can usually be arranged to overlap with the preceding part of the computation.

The idea of looking at the last record in each block, to predict which buffer will empty first, was discovered in 1953 by F. E. Holberton. The technique was first published by E. H. Friend [*JACM* **3** (1956), 144–145, 165]. His rather complicated algorithm used $3P$ input buffers, with three dedicated to each input file; Algorithm F improves the situation by making use of "floating buffers," allowing any single file to claim as many as $P + 1$ input buffers at once, yet never needing more than $2P$ in all.

A discussion of merging with less than $2P$ input buffers appears at the end of this section. Some computers have a "scatter read—gather write" capability, which allows input/output from nonconsecutive cells in memory; the implications of such a feature are beyond the scope of this book.

Comparative behavior of merge patterns. Let us now use what we know about tapes and merging, to compare the effectiveness of the various merge patterns that we have studied in Sections 5.4.2 through 5.4.5. It is very instructive to work out the details of each method as applied to a particular "unbiased" example. Consider therefore the problem of sorting a file whose records each contain 100 characters, when there are 100,000 character positions of memory available for data storage (exclusive of the program and its auxiliary variables, and the comparatively small amount of space needed for links in a selection tree). The input appears in random order on tape, in blocks of 5000 characters each, and the output is to appear in the same format. There are five "scratch tapes" to work with, in addition to the unit containing the input tape.

The total number of records to be sorted is 100,000, but this information is not known in advance to the sorting algorithm.

Chart A (located between pages 340 and 341) summarizes the actions which transpire when ten different merging schemes are applied to this data. The best way to look at this important illustration is to imagine that you are actually watching the sort take place: scan each line slowly from left to right, pretending that you can actually see six tapes reading, writing, rewinding, and/or reading backward, as indicated on the diagram. During a P-way merge the input tapes will be moving only $1/P$ times as often as the output tape. Note that when the original input tape has been completely read (and rewound "with lock"), Chart A assumes that a skilled computer operator dismounts it and replaces it with a scratch tape, in just 30 seconds. In examples 2, 3, and 4 this is "critical path time" when the computer is idly waiting for the operator to finish; but in the remaining examples, the dismount-reload operation is overlapped by other processing.

Example 1. Read-forward balanced merge. Let's review the specifications of the problem: The records are 100 characters long, there is enough internal

memory to hold 1000 records at a time, and each block on the input tape contains 5000 characters (50 records). There are 100,000 records (= 10,000,000 characters = 2000 blocks) in all.

We are free to choose the block size for intermediate files. A six-tape balanced merge uses three-way merging, so the technique of Algorithm F calls for 8 buffers; we may therefore use blocks containing 1000/8 = 125 records (= 12500 characters) each.

The initial distribution pass can make use of replacement selection (Section 5.4.1), and in order to keep the tapes running smoothly we may use two input buffers of 50 records each, plus two output buffers of 125 records each. This leaves room for 650 records in the replacement selection tree. Most of the initial runs will therefore be about 1300 records long (10 or 11 blocks); it turns out that 78 initial runs are produced in Chart A, the last one being rather short.

The first merge pass indicated shows nine runs merged to tape 4, instead of alternating between tapes 4, 5, and 6. This makes it possible to do useful work while the computer operator is loading a scratch tape onto unit 6; since the total number S of runs is known once the initial distribution has been completed, the algorithm knows that $\lceil S/9 \rceil$ runs should be merged to tape 4, then $\lceil (S-3)/9 \rceil$ to tape 5, then $\lceil (S-6)/9 \rceil$ to tape 6.

The entire sorting procedure for this example can be summarized in the following way, using the notation introduced in Section 5.4.2:

1^{26}	1^{26}	1^{26}	—	—	—
—	—	—	3^9	3^9	3^8
9^3	9^3	$9^2 6^1$	—	—	—
—	—	—	27^1	27^1	24^1
78^1	—	—	—	—	—

Example 2. Read-forward polyphase merge. The second example in Chart A carries out the polyphase merge, according to Algorithm 5.4.2D. In this case we do five-way merging, so the memory is split into 12 buffers of 83 records each. During the initial replacement selection we have two 50-record input buffers and two 83-record output buffers, leaving 734 records in the tree; so the initial runs this time are about 1468 records long (17 or 18 blocks). The situation illustrated shows that $S = 70$ initial runs were obtained, the last two actually being only four blocks and one block long, respectively. The merge pattern can be summarized thus:

$0^{13}1^{18}$	$0^{13}1^{17}$	$0^{13}1^{15}$	$0^{12}1^{12}$	$0^8 1^8$	—
1^{15}	1^{14}	1^{12}	1^8	—	$0^8 1^4 2^1 5^3$
1^7	1^6	1^4	—	4^8	$1^4 2^1 5^3$
1^3	1^2	—	8^4	4^4	$2^1 5^3$
1^1	—	$16^1 19^1$	8^2	4^2	5^2
—	34^1	19^1	8^1	4^1	5^1
70^1	—	—	—	—	—

Curiously, polyphase actually took about 25 seconds *longer* than the far less sophisticated balanced merge! There are two main reasons for this:

1) Balanced merge was particularly lucky in this case, since $S = 78$ is so close to a perfect power of 3. If 82 initial runs had been produced, the balanced merge would have taken an extra pass.

2) Polyphase merge wasted 30 seconds while the input tape was being changed, and a total of more than 5 minutes went by while it was waiting for rewind operations to be completed. By contrast the balanced merge needed comparatively little rewind time. In the second phase of the polyphase merge, 13 seconds were saved because the 8 dummy runs on tape 6 could be assumed present even while that tape was rewinding; but no other rewind overlap occurred. Therefore polyphase lost out even though it required significantly less read/write time.

Example 3. Read-forward cascade merge. This case is analogous to the preceding, but using Algorithm 5.4.3C. The merging may be summarized thus:

1^{14}	1^{15}	1^{12}	1^{14}	1^{15}	—
1^{5}	1^{9}	—	1^{14}	1^{15}	$1^{3}2^{3}3^{6}$
$5^{1}6^{3}$	5^{3}	$5^{3}6^{2}$	—	1^{1}	2^{2}
—	12^{1}	6^{1}	18^{1}	18^{1}	16^{1}
70^{1}	—	—	—	—	—

(Remember to watch each of these examples in action, by scanning Chart A.)

Example 4. Tape-splitting polyphase merge. This procedure, described at the end of Section 5.4.2, allows most of the rewind time to be overlapped. It uses four-way merging, so we divide the memory into ten 100-record buffers; there are 700 records in the replacement selection tree, so it turns out that 72 initial runs are formed. The last run, again, is very short. A distribution scheme analogous to Algorithm 5.4.2D has been used, followed by a simple but somewhat ad hoc method of placing dummy runs:

1^{21}	1^{19}	1^{15}	1^{8}	—	$0^{2}1^{9}$
$0^{2}1^{17}$	$0^{2}1^{15}$	$0^{2}1^{11}$	$0^{2}1^{4}$	—	$0^{2}1^{9}4^{4}$
1^{13}	1^{11}	1^{7}	—	$0^{2}4^{4}$	$0^{2}1^{9}4^{4}$
1^{10}	1^{8}	1^{4}	—	$0^{2}4^{4}3^{2}4^{1}$	$1^{8}4^{4}$
1^{6}	1^{4}	—	4^{4}	$0^{2}4^{4}3^{2}4^{1}$	$1^{4}4^{4}$
1^{5}	1^{3}	—	$4^{4}3^{1}$	$0^{1}4^{4}3^{2}4^{1}$	$1^{3}4^{4}$
1^{2}	—	$3^{1}7^{2}$	$4^{4}3^{1}$	$4^{2}3^{2}4^{1}$	4^{4}
1^{1}	—	$3^{1}7^{2}13^{1}$	$4^{3}3^{1}$	$4^{1}3^{2}4^{1}$	4^{3}
—	13^{1}	$3^{1}7^{2}13^{1}$	$4^{2}3^{1}$	$3^{2}4^{1}$	4^{2}
—	$13^{1}14^{1}$	$7^{2}13^{1}$	$4^{1}3^{1}$	$3^{1}4^{1}$	4^{1}
18^{1}	$13^{1}14^{1}$	$7^{1}13^{1}$	3^{1}	4^{1}	—
18^{1}	14^{1}	13^{1}	—	—	27^{1}
—	—	—	72^{1}	—	—

This turns out to give the best running time of all the examples in Chart A that do not read backward. Since S will never get to be very large, it would be possible to develop a more complicated algorithm which places dummy runs in an even better way (cf. Eq. 5.4.2–26).

Example 5. Cascade merge with rewind overlap. This procedure runs almost as fast as the previous example, although the algorithm governing it is much simpler. We simply use the cascade sort method as in Algorithm 5.4.3C for the initial distribution, but with $T = 5$ instead of $T = 6$. Then each phase of each "cascade" staggers the tapes so that we ordinarily don't write on a tape until after it has had a chance to be rewound. The pattern, very briefly, is

1^{21}	1^{22}	1^{19}	1^{10}	—	—
1^{4}	1^{7}	—	—	$1^{2}2^{2}3^{5}$	4^{10}
7^{2}	—	8^{3}	$7^{2}8^{2}$	—	4^{1}
—	26^{1}	—	8^{1}	22^{1}	16^{1}
72^{1}	—	—	—	—	—

Example 6. Read-backward balanced merge. This is like example 1 but with all the rewinding eliminated:

A_1^{26}	A_1^{26}	A_1^{26}	—	—	—
—	—	—	D_3^{9}	D_3^{9}	D_3^{8}
A_9^{3}	A_9^{3}	$A_9^{2}A_6^{1}$	—	—	—
—	—	—	D_{24}^{1}	D_{27}^{1}	D_{27}^{1}
A_{78}^{1}	—	—	—	—	—

Since there was comparatively little rewinding in example 1, this scheme is not a great deal better than the read-forward case. In fact, it turns out to be slightly slower than tape-splitting polyphase, in spite of the fortunate value $S = 78$.

Example 7. Read-backward polyphase merge. In this example only five of the six tapes are used, in order to eliminate the time for rewinding and changing the input tape. Thus, only four-way merging is used, and the buffer allocation is like that in examples 4 and 5. A distribution like Algorithm 5.4.2D is used, but with alternating directions of runs, and with tape 1 fixed as the final output tape. First an ascending run is written on tape 1; then descending runs on tapes 2, 3, 4; then ascending runs on 2, 3, 4; then descending on 1, 2, 3; etc. Each time we switch direction, replacement selection usually produces a shorter run, so it turns out that 77 initial runs are formed instead of the 72 in examples 4 and 5.

This procedure results in a distribution of (22, 21, 19, 15) runs, and the next perfect distribution is (29, 56, 52, 44). Exercise 5.4.4–5 shows how to generate strings of merge numbers which can be used to place dummy runs in

an "optimum" way; such a procedure is feasible in practice because the finiteness of a tape reel ensures that S is never too large. Therefore the example in Chart A has been constructed using such a method for dummy run placement (see exercise 7). This turns out to be the fastest of all the examples illustrated.

Example 8. Read-backward cascade merge. As in example 7, only five tapes are used here. This procedure follows Algorithm 5.4.3C, using rewind and forward read to avoid one-way merging (since rewinding is more than twice as fast as reading on MIXT units). Distribution is therefore the same as in example 6. The pattern may be summarized briefly as follows, using ↓ to denote rewinding:

A_1^{21}	A_1^{22}	A_1^{19}	A_1^{10}	—
$A_1^4\downarrow$	$A_1^7\downarrow$	—	$D_1^2D_2^2D_3^5$	D_4^{10}
$A_8A_7^2$	A_5^2	A_9^4	—	$D_4^1\downarrow$
—	D_{17}	$A_9\downarrow$	D_{25}	D_{21}
A_{72}	—	—	—	—

Example 9. Read-backward oscillating sort. Oscillating sort with $T = 5$ (Algorithm 5.4.5B) can use buffer allocation as in examples 4, 5, 7, and 8, since it does four-way merging. However, replacement selection does not behave in the same way, since a run of length 700 (not 1400 or so) is output just before entering each merge phase, in order to clear the internal memory. Consequently 85 runs are produced in this example, instead of 72. Some of the key steps in the process are

—	A_1	A_1A_1	A_1A_1	A_1A_1
D_4	—	A_1	A_1	A_1
. . . . .				
D_4D_4	D_4D_4	D_4D_4	D_4	—
D_4	D_4	D_4	—	A_{16}
. . . . .				
D_4	$A_{16}D_4D_4$	$A_{16}D_4$	$A_{16}D_4A_1$	A_{16}
D_4	$A_{16}D_4D_4$	$A_{16}D_4D_1$	$A_{16}D_4$	A_{16}
—	$A_{16}D_4$	$A_{16}D_4$	A_{16}	$A_{16}A_{13}$
—	$A_{16}D_4$	A_{16}	$A_{16}A_4$	$A_{16}A_{13}$
—	A_{16}	$A_{16}A_4$	$A_{16}A_4$	$A_{16}A_{13}$
D_{37}	—	$A_{16}\downarrow$	$A_{16}\downarrow$	$A_{16}\downarrow$
—	A_{85}	—	—	—

Example 10. Read-forward oscillating sort. In the final example, replacement selection is not used because all initial runs must be the same length. There-

fore full core loads of 1000 records are internally sorted whenever an initial run is required; this makes $S = 100$. Some key steps in the process are

A_1	A_1	A_1	A_1	A_1
—	—	—	—	A_1A_4
. . . . .				
A_1	A_1	A_1	A_1	A_1A_4
—	—	—	A_1A_4	$\not{A}_1A_4$
—	—	—	A_1A_4	A_1A_4
. . . . .				
A_1	A_1A_4	A_1A_4	A_1A_4	A_1A_4
A_1A_4	$\not{A}_1A_4$	$\not{A}_1A_4$	$\not{A}_1A_4$	$\not{A}_1A_4$
$A_1A_4A_{16}$	—	—	—	—
. . . . .				
—	A_1A_4	A_1A_4	A_1A_4	$A_1A_4A_{16}A_{64}$
A_4	$\not{A}_1A_4$	$\not{A}_1A_4$	$\not{A}_1A_4$	$\not{A}_1A_4A_{16}A_{64}$
A_4A_{16}	—	—	—	$\not{A}_1\not{A}_4A_{16}A_{64}$
$\not{A}_4A_{16}$	A_4	—	—	$\not{A}_1\not{A}_4A_{16}A_{64}$
—	—	—	A_{36}	$\not{A}_1\not{A}_4\not{A}_{16}A_{64}$
A_{100}	—	—	—	—

This routine turned out to be slowest of all, partly because it did not use replacement selection, but mostly because of its rather awkward ending (a two-way merge).

Estimating the running time. Let's see now how to figure out the approximate execution time of a sorting method using MIXT tapes. Could we have predicted the outcomes shown in Chart A without carrying out a detailed simulation?

One way that has traditionally been used to compare different merge patterns is to superimpose graphs such as we have seen in Figs. 70, 74, and 78. These graphs show the effective number of passes over the data, as a function of the number of initial runs, assuming that each initial run has approximately the same length. (See Fig. 85.) But this is *not* a very realistic comparison, because we have seen that different methods lead to different numbers of initial runs; furthermore there is a different "overhead time" caused by the relative frequency of interblock gaps, and the rewind time also has significant effects. All of these machine-dependent features make it impossible to prepare charts which make a valid comparison of the methods on a machine-independent basis. On the other hand, Fig. 85 does show us that, except for balanced merge, the effective number of passes can be reasonably well approximated by smooth curves of the form $\alpha \ln S + \beta$. Therefore we can make a fairly good comparison

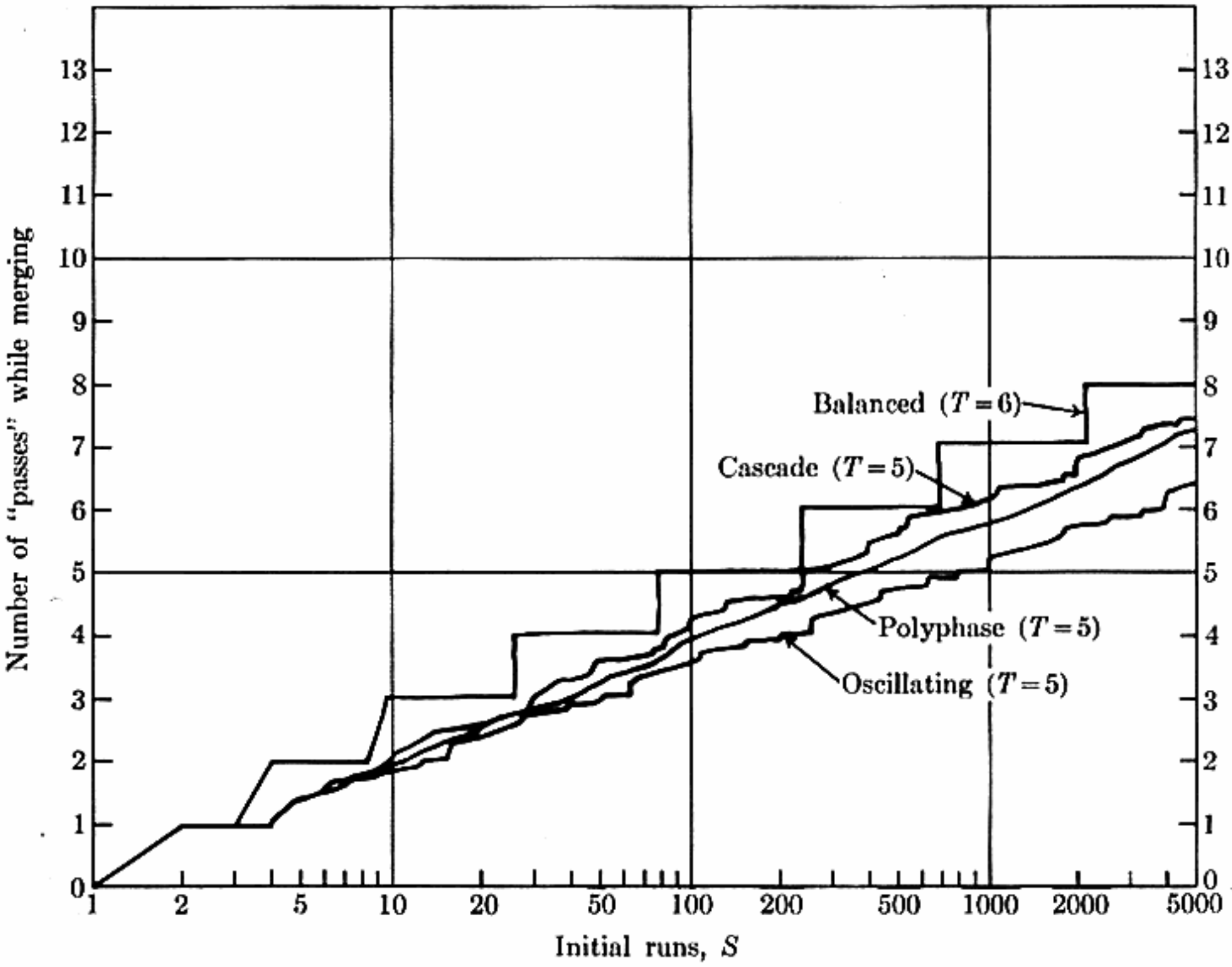


Fig. 85. A somewhat misleading way to compare merge patterns.

of the methods in any particular situation, by studying formulas that approximate the running time. Our goal, of course, is to find formulas that are simple yet sufficiently realistic.

Let us now attempt to develop such formulas, in terms of the following parameters:

N = number of records to be sorted,
C = number of characters per record,
M = number of character positions available in the internal memory (assumed to be a multiple of C),
τ = number of seconds to read or write one character,
$\rho\tau$ = number of seconds to rewind over one character,
$\sigma\tau$ = number of seconds for stop-start time delay,
γ = number of characters per interblock gap,
δ = number of seconds for operator to dismount and replace input tape,
B_i = number of characters per block in the unsorted input,
B_o = number of characters per block in the sorted output.

For MIXT we have $\tau = 1/60000$, $\rho = 2/5$, $\sigma = 300$, $\gamma = 480$. The example application treated above has $N = 100000$, $C = 100$, $M = 100000$, $B_i = B_o = 5000$, $\delta = 30$. These parameters are usually the machine and data characteristics that affect sorting time most critically (although rewind time is often given by a more complicated expression than a simple ratio ρ). Given the above parameters and a merge pattern, we shall compute further quantities such as

$P =$ maximum order of merge in the pattern,

$P' =$ number of records in replacement selection tree,

$S =$ number of initial runs,

$\pi = \alpha \ln S + \beta =$ approximate average number of times each character is read and written, not counting the initial distribution or the final merge,

$\pi' = \alpha' \ln S + \beta' =$ approximate average number of times rewinding over each character during intermediate merge phases,

$B =$ number of characters per block in the intermediate merge phases,

$\omega_i, \omega, \omega_o =$ "overhead ratio," the effective time required to read or write a character (due to gaps and stop/start) divided by the hardware time τ.

The examples of Chart A have chosen block and buffer sizes according to the formula

$$B = \left\lfloor \frac{M}{C(2P+2)} \right\rfloor C, \tag{1}$$

so that the blocks can be as large as possible consistent with the buffering scheme of Algorithm F. (In order to avoid trouble during the final pass, P should be small enough that (1) makes $B \geq B_o$.) The size of the tree during replacement selection is then

$$P' = (M - 2B_i - 2B)/C. \tag{2}$$

For random data the number of initial runs S can be estimated as

$$S \approx \left\lceil \frac{N}{2P'} + \frac{7}{6} \right\rceil, \tag{3}$$

using the results of Section 5.4.1. Assuming that $B_i < B$ and that the input tape can be run at full speed during the distribution (see below), it takes about $NC\omega_i\tau$ seconds to distribute the initial runs, where

$$\omega_i = (B_i + \gamma)/B_i. \tag{4}$$

While merging, the buffering scheme allows simultaneous reading, writing, and computing, but the frequent switching between input tapes means that we must add the stop/start time penalty; therefore we set

$$\omega = (B + \gamma + \sigma)/B, \tag{5}$$

and the merge time is approximately

$$(\pi + \rho\pi')NC\omega\tau. \tag{6}$$

This formula penalizes rewind slightly, since ω includes stop/start time, but other considerations (such as rewind interlock and the penalty for reading from load point) usually compensate for this. The final merge pass, assuming that $B_o \leq B$, is constrained by the overhead ratio

$$\omega_o = (B_o + \gamma)/B_o. \tag{7}$$

We may estimate the running time of the final merge and rewind as

$$NC(1 + \rho)\omega_o\tau;$$

in practice it might take somewhat longer due to the presence of unequal block lengths (input and output are not synchronized as in Algorithm F), but the running time will be pretty much the same for all merge patterns.

Before going into more specific formulas for individual patterns, let us try to justify two of the assumptions made above.

a) *Can replacement selection keep up with the input tape?* In the examples of Chart A it probably can, since it takes about ten iterations of the inner loop of Algorithm 5.4.1R to select the next record, and we have $C\omega_i\tau >$ 1667 microseconds in which to do this. With careful programming of the replacement selection loop, this can be done on many (but not all) machines. Note that the situation is somewhat less critical while merging: the computation time per record is almost always less than the tape time per record during a P-way merge, since P isn't very large.

b) *Should we really choose B to be the maximum possible buffer size, as in* (1)? A large buffer size cuts down the overhead ratio ω in (5); but it also increases the number of initial runs S, since P' is decreased. It is not immediately clear which factor is more important. Considering the merging time as a function of $x = CP'$, we can express it in the form

$$\left(\theta_1 \ln\left(\frac{N}{x} + \frac{7}{6}\right) + \theta_2\right)\left(\frac{\theta_3 - x}{\theta_4 - x}\right) \tag{8}$$

for some appropriate constants θ_1, θ_2, θ_3, θ_4, with $\theta_3 > \theta_4$. Differentiating with respect to x shows that there is some N_0 such that for all $N \geq N_0$ it does not pay to increase x at the expense of buffer size. In the sorting

application of Chart A, for example, N_0 turns out to be roughly 10000; when sorting more than 10000 records the large buffer size is superior.

Note however that with balanced merge the number of passes jumps sharply when S passes a power of P. If an approximation to N is known in advance, the buffer size should be chosen so that S will most likely be slightly less than a power of P. For example, the buffer size for the first line of Chart A was 12500; since $S = 78$, this was very satisfactory, but if S had turned out to be 82 it would have been much better to decrease the buffer size a little.

Formulas for the ten examples. Returning to Chart A, let us try to give formulas which approximate the running time in each of the ten methods. In most cases the basic formula

$$NC\omega_i\tau + (\pi + \rho\pi')NC\omega\tau + (1 + \rho)NC\omega_o\tau \tag{9}$$

will be a sufficiently good approximation to the overall sorting time, once we have specified the number of intermediate merge passes $\pi = \alpha \ln S + \beta$ and the number of intermediate rewind passes $\pi' = \alpha' \ln S + \beta'$. Sometimes it is necessary to add a further correction to (9); details for each method can be worked out as follows:

Example 1. Read-forward balanced merge. The formulas $\pi = \lceil \ln S/\ln P \rceil - 1$, $\pi' = \lceil \ln S/\ln P \rceil/P$ may be used for P-way merging on $2P$ tapes.

Example 2. Read-forward polyphase merge. We may take $\pi' \approx \pi$, since every phase is usually followed by a rewind of about the same length as the previous merge. From Table 5.4.2–1 we get the values $\alpha = 0.795$, $\beta = 0.864 - 2$, in the case of six tapes. (This "-2" is due to the fact that the table entry includes the initial and final passes as well as the intermediate ones.) The time for rewinding the input tape after the initial distribution, namely $\rho NC\omega_i\tau + \delta$, should be added to (9).

Example 3. Read-forward cascade merge. Table 5.4.3–1 gives the values $\alpha = 0.773$, $\beta = 0.808 - 2$. Rewind time is comparatively difficult to estimate; perhaps setting $\pi' \approx \pi$ is accurate enough. As in example 2, we need to add the initial rewind time to (9).

Example 4. Tape-splitting polyphase merge. Table 5.4.2–5 gives $\alpha = 0.752$, $\beta = 1.024 - 2$. The rewind time is almost overlapped except after the initialization ($\rho NC\omega_i\tau + \delta$) and two phases near the end ($2\rho NC\omega\tau$ times 36 percent). We may also subtract 0.18 from β since the first half phase is overlapped by the initial rewind.

Example 5. Cascade merge with rewind overlap. Here we use Table 5.4.3–1 for $T = 5$, to get $\alpha = 0.897$, $\beta = 0.800 - 2$. Nearly all of the unoverlapped rewind occurs just after the initial distribution and just after each two-way merge. After a perfect initial distribution, the longest tape contains about $1/g$ of the data, where g is the "growth ratio." After each two-way merge the amount of rewind in the six-tape case is $d_k\, d_{n-k}$ (cf. exercise 5.4.3–5), and it can be shown that in the T-tape case the amount of rewind after two-way merges is approximately

$$(2/(2T - 1))(1 - \cos(4\pi/(2T - 1)))$$

of the file. In our case ($T = 5$) this is $\frac{2}{9}(1 - \cos 80°) \approx 0.183$ of the file, and the number of times it occurs is $0.946 \ln S + 0.796 - 2$.

Example 6. Read-backward balanced merge. This is like example 1, except that

most of the rewinding is eliminated. The change in direction from forward to backward causes some delays, but they are not significant. There is a 50–50 chance that rewinding before the final pass will be necessary, so we may take $\pi' = 1/(2P)$.

Example 7. Read-backward polyphase merge. Since replacement selection in this case produces runs that change direction about every P times, we must replace (3) by another formula for S. A reasonably good approximation (cf. exercise 5.4.1–24) is $S = \lceil N(3 + 1/P)/6P' \rceil + 1$. All rewind time is eliminated, and Table 5.4.2–1 gives $\alpha = 0.863, \beta = 0.921 - 2$.

Example 8. Read-backward cascade merge. From Table 5.4.3–1 we have $\alpha = 0.897$, $\beta = 0.800 - 2$. The rewind time can be estimated as twice the difference "passes with copying" minus "passes without copying" in that table, plus $1/(2P)$ in case the final merge must be preceded by rewinding to get ascending order.

Example 9. Read-backward oscillating sort. In this case replacement selection has to be started and stopped many times; bursts of $P - 1$ to $2P - 1$ runs are distributed at a time, averaging P in length; the average length of runs therefore turns out to be approximately $P'(2P - 4/3)/P$, and we may estimate $S = \lceil N/((2 - 4/(3P))P') \rceil + 1$. A little time is used to switch from merging to distribution and vice-versa; this is approximately the time to read in P' records from the input tape, namely $P'C\omega_i\tau$, and it occurs about S/P times. Rewind time and merging time may be estimated as in example 6.

Example 10. Read-forward oscillating sort. This method is not easy to analyze, because the final "cleanup" phases performed after the input is exhausted are not as efficient as the earlier phases. Ignoring this troublesome aspect, and simply calling it one extra pass, we can estimate the merging time by setting $\alpha = 1/\ln P$, $\beta = 0$, and $\pi' = \pi/P$. The distribution of runs is somewhat different in this case, since replacement selection is not used; we set $P' = M/C$ and $S = \lceil N/P' \rceil$. With care it is possible to overlap computing, reading, and writing during the distribution, with an additional factor of about $(M + 2B)/M$ in the overhead. The "mode-switching" time mentioned in example 9 is not needed in the present case because it is overlapped by rewinding. So the estimated sorting time in this case is (9) plus $2BNC\omega_i\tau/M$.

Table 1

SUMMARY OF SORTING TIME ESTIMATES

Example	P	B	P'	S	ω	α	β	α'	β'	(9)	Additions to (9)	Estimated total	Actual total
1	3	12500	650	79	1.062	0.910	−1.000	0.303	0.000	1064		1064	1076
2	5	8300	734	70	1.094	0.795	−1.136	0.795	−1.136	1010	$\rho NC\omega_i\tau + \delta$	1113	1103
3	5	8300	734	70	1.094	0.773	−1.192	0.773	−1.192	972	$\rho NC\omega_i\tau + \delta$	1075	1127
4	4	10000	700	73	1.078	0.752	−0.994	0.000	0.720	844	$\rho NC\omega_i\tau + \delta$	947	966
5	4	10000	700	73	1.078	0.897	−1.200	0.173	0.129	972		972	992
6	3	12500	650	79	1.062	0.910	−1.000	0.000	0.167	981		981	980
7	4	10000	700	79	1.078	0.863	−1.079	0.000	0.000	922		922	907
8	4	10000	700	73	1.078	0.897	−1.200	0.098	0.117	952		952	949
9	4	10000	700	87	1.078	0 721	−1.000	0.000	0.125	846	$P'SC\omega_i\tau/P$	874	928
10	4	10000	—	100	1.078	0.721	0.000	0.180	0.000	1095	$2BNC\omega_i\tau/M$	1131	1158

Table 1 shows that the estimates are not too bad in these examples, although in a few cases there is a discrepancy of 50 seconds or so. The formulas in examples 2 and 3 indicate that cascade merge should be preferable to poly-

phase on six tapes, yet in practice polyphase was better! The reason is that graphs like Fig. 85 (which shows the five-tape case) are more nearly straight lines for the polyphase algorithm; cascade is superior to polyphase on six tapes for $14 \le S \le 15$ and $43 \le S \le 55$, near the "perfect" cascade numbers 15 and 55, but the polyphase distribution of Algorithm 5.4.2D is equal or better for all other $S \le 100$. Cascade will win over polyphase as $S \to \infty$, but S doesn't actually approach ∞! The underestimate in example 9 is due to similar circumstances; polyphase was superior to oscillating even though the asymptotic theory tells us that oscillating will be best for large S.

Some miscellaneous remarks. It is now appropriate to make a few more or less random observations about tape merging:

1) The above formulas show that the sorting is essentially a function of N times C, not of N and C independently. Except for a few relatively minor considerations (such as the fact that B was taken to be a multiple of C), our formulas say that it takes about as long to sort one million records of 10 characters each as to sort 100 thousand records of 100 characters each. Actually there may be a difference, not revealed in our formulas, because of the space used by link fields during replacement selection. In any event the size of the *key* makes hardly any difference, unless keys get so long and complicated that internal computation cannot keep up with the tapes.

With long records and short keys it is tempting to "detach" the keys, sort them first, and then somehow rearrange the records as a whole. But this idea doesn't seem to work out, it merely postpones the agony, because the final rearrangement procedure takes about as long as a conventional merge sort would take.

2) We have seen that it takes from 15 to 19 minutes to sort 100,000 100-character records under the above assumptions. How long would it take to sort them on a card sorter? This question is of practical interest because card sorters are cheaper than computers. Assuming that each record can be squeezed onto an 80-column card, and that the alphabetic key occupies six columns, requiring an average of $1\frac{2}{3}$ passes to sort on each column, we must run each card through the machine about 10 times. At 1000 cards per minute that would take 1000 minutes, which is almost 17 hours. (During this time there is a fairly good chance that some cards will get inadvertently shuffled or jammed in the machine.)

3) When writing a sort routine that is to be used repeatedly, it is wise to estimate its running time very carefully and to compare the theory with actual observed performance. Since the theory of sorting has been fairly well developed, this procedure has been known to turn up bugs in the input/output hardware or software on existing systems; the service was slower than it should have been, yet nobody had noticed it until the sorting routine ran too slowly!

4) Some computer systems have two "banks" of tape units, attached to separate "channels" in such a way that simultaneous reading and writing is

permitted only on tapes from opposite banks. Balanced merging is especially well suited to such configurations. Consider, for example, the case of six tapes with three in each bank, and suppose that we want to do polyphase merging with $T = 6$. During a five-way merge, two of the input tapes will be on the wrong bank, so roughly two-fifths of the input time cannot be overlapped with output. This adds approximately 40 percent to the sorting time, so that balanced merge turns out to be better than polyphase even in the read-backward case.

5) Our analaysis of replacement selection has been carried out for "random" files, but the files which actually arise in practice very often have a good deal of existing order. (In fact, sometimes people will sort a file that is already in order, just to be sure.) Therefore experience has shown that replacement selection is preferable to other kinds of internal sort, even more so than our formulas indicate. This advantage is slightly mitigated in the case of read-backward polyphase sorting, since a number of descending runs must be produced; indeed, R. L. Gilstad (who first published the polyphase merge) originally rejected the read-backward technique for that reason. But he noticed later that alternating directions will still pick up long ascending runs. Furthermore, read-backward polyphase is the only standard technique which likes descending input files as well as ascending ones.

6) Another advantage of replacement selection is that it allows simultaneous reading, writing, and computing. If we merely did the internal sort in an obvious way—filling the memory, sorting it, then writing it out as it becomes filled with the next load—the distribution pass would take about twice as long!

The only other internal sort we have discussed which appears to be amenable to simultaneous reading, writing, and computing is heapsort. (This idea has been used in preparing example 10 of Chart A.) Suppose for convenience that the internal memory holds 1000 records, and that each block on tape holds 100. We may proceed as follows, letting $B_1\, B_2 \ldots B_{10}$ stand for the contents of memory divided into ten 100-record blocks.

Step 0. Fill memory, and make the elements of $B_2 \ldots B_{10}$ satisfy the inequalities for a heap (with smallest element at the root).

Step 1. Make $B_1 \ldots B_{10}$ into a heap, then select out the least 100 records and move them to B_{10}.

Step 2. Write out B_{10}, while selecting the smallest 100 records of $B_1 \ldots B_9$ and moving them to B_9.

Step 3. Read into B_{10}, and write out B_9, while selecting the smallest 100 records of $B_1 \ldots B_8$ and moving them to B_8.

⋮

Step 9. Read into B_4, and write out B_3, while selecting the smallest 100 records of $B_1\, B_2$ and moving them to B_2 and while making the heap inequalities valid in $B_5 \ldots B_{10}$.

Step 10. Read into B_3, and write out B_2, while sorting B_1 and while making the heap inequalities valid in $B_4 \ldots B_{10}$.

Step 11. Read into B_2, and write out B_1, while making the heap inequalities valid in $B_3 \ldots B_{10}$.

Step 12. Read into B_1, while making the heap inequalities valid in $B_2 \ldots B_{10}$. Return to step 1. ▮

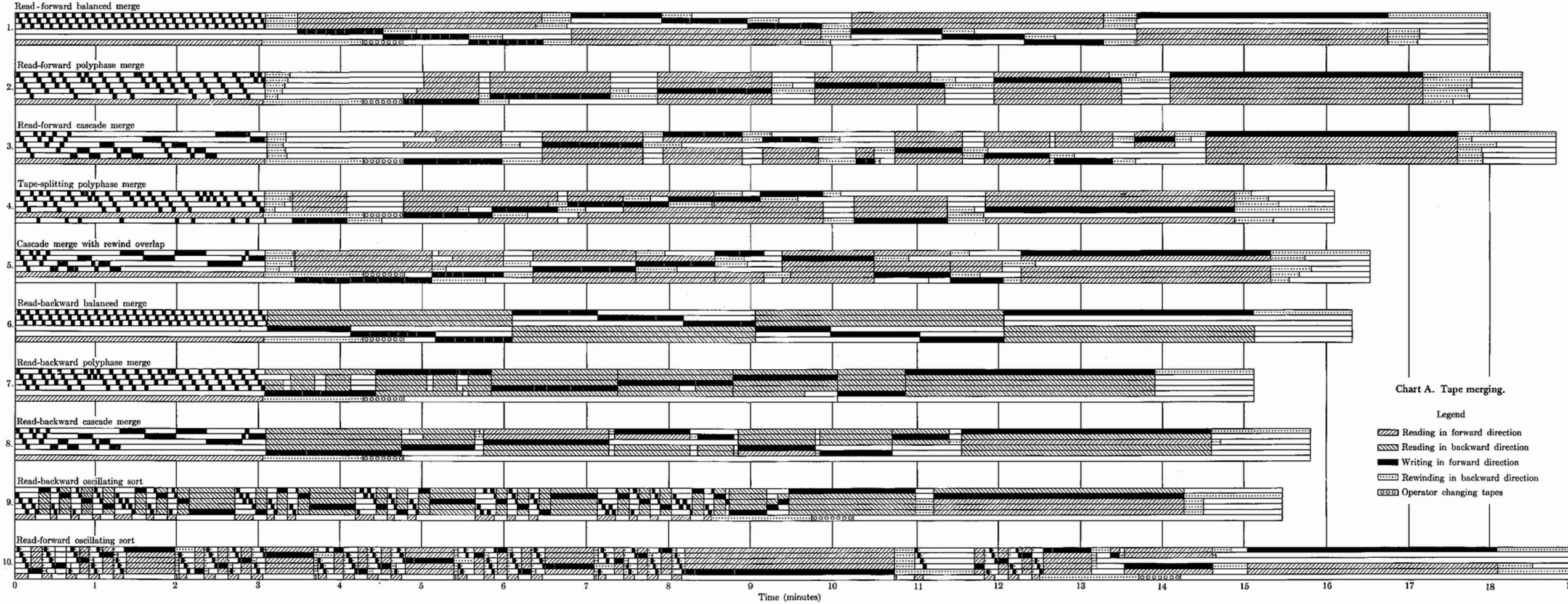
7) We have been assuming that the number N of records to be sorted is not known in advance. Actually in most computer applications it would be possible to keep track of the number of records in all files at all times, and we could assume that our computer system is capable of telling us the value of N. How much help would this be? Unfortunately, not very much! We have seen that replacement selection is very advantageous, but it leads to an unpredictable number of initial runs. In a balanced merge we could use information about N to set the buffer size B in such a way that S will probably be just less than a power of P; and in a polyphase distribution with optimum placement of dummy runs we could use information about N to decide what level to shoot for (cf. Table 5.4.2–2).

Alternatively, without using replacement selection we could apply the preorder merge, described at the close of Section 5.4.4, making it into an oscillating sort that distributes initial runs in the appropriate direction at the appropriate moment (instead of taking them from "tape A" as described there). This method is essentially *optimum* among all methods that do internal sorting without replacement selection, since it merges according to the best possible P-ary tree; but it actually runs more slowly than methods based on replacement selection.

8) Tape drives tend to be the least reliable part of a computer. Maintenance men typically have more calls to fix tape units than any other component of a machine, and computer operators must learn how to restart jobs after a tape failure. The author has never yet seen an installation with 10 or more tape units all simultaneously in good working condition! Therefore it is considered axiomatic that *the original input tape should never be destroyed until it is known that the entire sort has been satisfactorily completed.* The "operator dismount time" is annoying in some of the examples of Chart A, but it would be too risky to overwrite the input in view of the probability that something might go wrong during a long sort.

9) When changing from forward write to backward read, we can save some time by never writing the last bufferload onto tape; it will just be read back in again anyway! Chart A shows that this trick actually saves comparatively little time, except in the oscillating sort where directions are reversed frequently.

10) If we have a large number of tape units available, it is not always best to use as many of them as possible in order to get a "high order of merge." For example, a higher order of merge usually implies a smaller block size; and


Read-forward balanced merge
1.
Read-forward polyphase merge
2.
Read-forward cascade merge
3.
Tape-splitting polyphase merge
4.
Cascade merge with rewind overlap
5.
Read-backward balanced merge
6.
Read-backward polyphase merge
7.
Read-backward cascade merge
8.
Read-backward oscillating sort
9.
Read-forward oscillating sort
10.
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Time (minutes)
Chart A. Tape merging.
Legend
Reading in forward direction
Reading in backward direction
Writing in forward direction
Rewinding in backward direction
Operator changing tapes

the percentage difference between $\log_P S$ and $\log_{P+1} S$ is not very great when P is large. Consider also the poor computer operator who has to mount all those scratch tapes. On the other hand, exercise 12 describes an interesting way to make use of additional tape units, grouping them so as to overlap input/output time without increasing the order of merge.

11) On machines like MIX, which have fixed rather small block sizes, hardly any internal memory is needed while merging. Oscillating sort then becomes more attractive, because it becomes possible to maintain the replacement selection tree in memory while merging. In fact we can improve on oscillating sort in this case (as suggested by Colin J. Bell in 1962), merging a new initial run into the output every time we merge from the working tapes!

12) We have observed that multireel files should be sorted one reel at a time, in order to avoid excessive tape handling. This is sometimes called a "reel time" application. Actually a balanced merge on six tapes can sort *three* reelfulls, up until the time of the final merge, if it has been programmed carefully.

In order to merge a fairly large number of individually sorted reels, a minimum-path-length merging tree will be fastest (cf. Section 5.4.4). This construction was first made by E. H. Friend [*JACM* **3** (1956), 166–167]; then W. H. Burge [*Information and Control* **1** (1958), 181–197] pointed out that an optimum way to merge runs of given (possibly unequal) lengths is obtained by constructing a tree with minimum *weighted* path length, using the run lengths as weights (cf. Sections 2.3.4.5 and 5.4.9), if we ignore tape handling time. But multireel files should probably be kept on a disk or other large memory instead of tape.

13) Our discussions have blithely assumed that we have direct control over the input/output instructions for tape units, and that no complicated system interface keeps us from using tape as efficiently as the tape designers intended. These idealistic assumptions give us insights into the tape merging problem, and may give some insights into the proper design of operating systems, but we should realize that multiprogramming and multiprocessing can make the situation considerably more complicated.

14) The issues we have studied in this section were first discussed in print by E. H. Friend [*JACM* **3** (1956), 134–168], W. Zoberbier [*Elektron. Datenverarb.* **5** (1960), 28–44], and M. A. Goetz [*Digital Computer User's Handbook* (New York: McGraw-Hill, 1967) 1.292–1.320].

Summary. We can sum up what we have learned about comparing different merge patterns in the following way:

Theorem A. *It is difficult to decide which merge pattern is best in a given situation.* ∎

The examples we have seen in Chart A show how 100,000 randomly ordered 100-character records (or 1 million 10-character records) might be sorted using

six tapes under realistic assumptions. This much data fills about half of a tape, and it can be sorted in about 15 to 19 minutes; however, there is considerable variation in available tape equipment, and running times for such a job may vary between about four minutes and about two hours on different machines. About 3 minutes of the time in our examples were used for initial distribution of runs and internal sorting; about $4\frac{1}{2}$ minutes were used for the final merge and rewinding the output tape; and about $7\frac{1}{2}$ to $11\frac{1}{2}$ minutes were spent in intermediate stages of merging.

Given six tapes that cannot read backward, the best sorting method under our assumptions was the "tape-splitting polyphase merge" (example 4); and for tapes that do allow backward reading, the best method turned out to be read-backward polyphase with a complicated placement of dummy runs (example 7). Oscillating sort (example 9) was a close second. In both cases the cascade merge provided a simpler alternative which was only slightly slower (examples 5 and 8). In the read-forward case, a straightforward balanced merge (example 1) was surprisingly effective, partly by luck in this particular example but partly also because it spends comparatively little time rewinding.

The situation would change somewhat if we had a different number of available tapes.

Sort generators. Given the wide variability of data and equipment characteristics, it is almost impossible to write a single external sorting program that is satisfactory in very many applications. And it is also rather difficult to prepare a program that really handles tapes efficiently. Therefore the preparation of sorting software is a particularly challenging job. A *sort generator* is a program that produces machine code specially tailored to particular sorting applications, based on parameters which describe the data format and the hardware configuration. Such a program is often tied to high-level languages such as COBOL or PL/I, or it might be written as a set of macros to be used with a macroassembler.

One of the features normally provided by a sort generator is the ability to insert "own coding," special instructions to be incorporated into the first and last passes of the sorting routine. First-pass own coding is usually used to edit the input records, often shrinking them or slightly expanding them into a form which is easier to sort. For example, suppose that the input records are to be sorted on a nine-character key which represents a date in month-day-year format:

```
JUL041776  OCT311517  NOV051605  JUL141789  NOV201917
```

On the first pass the three-letter month code can be looked up in a table, and the month codes can be replaced by numbers with the most significant fields at the left:

```
17760704  15171031  16051105  17890714  19171120
```

This decreases the record length and makes subsequent comparisons much

simpler. (An even more compact code could also be substituted.) Last-pass own coding can be used to restore the original format, and/or to make other desired changes to the file, and/or to compute some function of the output records. The merging algorithms we have studied are organized in such a way that it is easy to distinguish the last pass from other merge phases. Note that when own coding is present there must be at least two passes over the file even if it is initially in order. Own coding which changes the record size can make it difficult for the oscillating sort to overlap some of its input/output operations.

Sort generators also take care of system details like tape label conventions, and they often provide for "hash totals" or other checks to make sure that none of the data has been lost or altered. Sometimes there are provisions for stopping the sort at convenient places and resuming later. The fanciest generators allow records to have dynamically varying lengths [cf. D. J. Waks, *CACM* **6** (1963), 267–272].

***Merging with fewer buffers.** We have seen that $2P + 2$ buffers are sufficient to keep tapes moving rapidly during a P-way merge. Let us conclude this section by making a mathematical analysis of the merging time when *less* than $2P + 2$ buffers are present.

Two output buffers are clearly desirable, since we can be writing from one while forming the next block of output in the other. Therefore we may ignore the output question entirely, and concentrate only on the input.

Suppose there are $P + Q$ input buffers, where $1 \le Q \le P$. We shall use the following approximate model of the situation, as suggested by L. J. Woodrum [*IBM Systems J.* **9** (1970), 118–144]: It takes one unit of time to read a block of tape. During this time there is a probability p_0 that no input buffers have been emptied, p_1 that one has been emptied, $p_{\ge 2}$ that two or more have been, etc. When completing a tape read we are in one of $Q + 1$ states:

State 0. Q buffers are empty; we begin to read a block into one of them from the appropriate file, using the forecasting technique explained earlier in this section. After one unit of time we go to state 1 with probability p_0, otherwise we remain in state 0.

State 1. $Q - 1$ buffers are empty; we begin to read into one of them, forecasting the appropriate file. After one unit of time we go to state 2 with probability p_0, to state 1 with probability p_1, and to state 0 with probability $p_{\ge 2}$.

⋮

State $Q - 1$. One buffer is empty; we begin to read into it, forecasting the appropriate file. After one unit of time we go to state Q with probability p_0, to state $Q - 1$ with probability $p_1, \ldots,$ to state 1 with probability p_{Q-1}, and to state 0 with probability $p_{\ge Q}$.

State Q. All buffers are filled. Tape reading stops for an average of μ units of time and then we go to state 1.

We start in state 0. This model of the situation corresponds to a *Markov process* (cf. exercise 2.3.4.2–26), which can be analyzed via generating functions in the following interesting way: Let z be an arbitrary parameter, and assume that each time we have a chance to read from tape we make a decision to do so with probability z, but we decide to terminate the algorithm with probability $1 - z$. Now let $g_Q(z) = \sum_{n\ge 0} a_n^{(Q)} z^n (1 - z)$ be the average number of times that state Q occurs in such a process; it follows that $a_n^{(Q)}$ is the average number of times state Q occurs when exactly n blocks have been read. Then $n + a_n\mu$ is the average total time for input + computation. If we had perfect overlap, as in the $(2P + 2)$-buffer algorithm, the total time would be only n units, so $a_n\mu$ represents the "reading hangup" time.

Let A_{ij} be the probability that we go from state i to state j in this process, for $0 \le i, j \le Q + 1$, where $Q + 1$ is a new "stopped" state. For example, the A-matrix takes the following forms for small Q:

$$Q = 1: \quad \begin{pmatrix} p_{\ge 1}z & p_0 z & 1 - z \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

$$Q = 2: \quad \begin{pmatrix} p_{\ge 1}z & p_0 z & 0 & 1 - z \\ p_{\ge 2}z & p_1 z & p_0 z & 1 - z \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix},$$

$$Q = 3: \quad \begin{pmatrix} p_{\ge 1}z & p_0 z & 0 & 0 & 1 - z \\ p_{\ge 2}z & p_1 z & p_0 z & 0 & 1 - z \\ p_{\ge 3}z & p_2 z & p_1 z & p_0 z & 1 - z \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Exercise 2.3.4.2–26b tells us that $g_Q(z) = \text{cofactor}_{Q0}(I - A)/\det(I - A)$. Thus for example when $Q = 1$ we have

$$g_1(z) = \det\begin{pmatrix} 0 & -p_0 z & z - 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \Big/ \det\begin{pmatrix} 1 - p_{\ge 1}z & -p_0 z & z - 1 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$= \frac{p_0 z}{1 - p_1 z - p_0 z} = \frac{p_0 z}{1 - z} = \sum_{n\ge 0} n p_0 z^n (1 - z),$$

so $a_n^{(1)} = np_0$. This of course was obvious *a priori*, since the problem is very simple when $Q = 1$. A similar calculation when $Q = 2$ (see exercise 14) gives the less obvious formula

$$a_n^{(2)} = \frac{p_0^2 n}{1 - p_1} - \frac{p_0^2 (1 - p_1^n)}{(1 - p_1)^2}. \tag{10}$$

In general we can show that $a_n^{(Q)}$ has the form $\alpha^{(Q)}n + O(1)$ as $n \to \infty$, where the constant $\alpha^{(Q)}$ is not terribly difficult to calculate. (See exercise 15.) It turns out that $\alpha^{(3)} = p_0^3/((1 - p_1)^2 - p_0p_2)$.

The nature of merging makes it fairly reasonable to assume that $\mu = 1/P$ and that we have a binomial distribution

$$p_k = \binom{P}{k}\left(\frac{1}{P}\right)^k\left(\frac{P-1}{P}\right)^{P-k}.$$

For example, when $P = 5$ we have $p_0 = .32768$, $p_1 = .4096$, $p_2 = .2048$, $p_3 = .0512$, $p_4 = .0064$, and $p_5 = .00032$; hence $\alpha^{(1)} = 0.328$, $\alpha^{(2)} = 0.182$, and $\alpha^{(3)} = 0.127$. In other words, if we use $5 + 3$ input buffers instead of $5 + 5$, we can expect an additional "reading hangup" time of about $0.127/5 \approx$ 2.5 percent.

Of course this model is only a very rough approximation; we know that when $Q = P$ there is no hangup time at all, but the model says there is. The extra reading hangup time for smaller Q just about counterbalances the savings in overhead gained by having larger blocks, so the simple $Q = P$ scheme seems to be vindicated.

EXERCISES

1. [*13*] Give a formula for the exact number of characters per tape, when every block on the tape contains n characters. Assume that the tape could hold exactly 23000000 characters if there were no interblock gaps.

2. [*15*] Explain why the first buffer for File 2, in line 6 of Fig. 84, is completely blank.

3. [*20*] Would Algorithm F work properly if there were only $2P - 1$ input buffers instead of $2P$? If so, prove it; if not, give an example where it fails.

4. [*20*] How can Algorithm F be changed so that it works also when $P = 1$?

▶ **5.** [*21*] When equal keys are present on different files, it is necessary to be very careful in the forecasting process. Explain why, and show how to avoid difficulty by defining the merging and forecasting operations of Algorithm F more precisely.

6. [*22*] What changes should be made to Algorithm 5.4.3C, in order to convert it into an algorithm for cascade merge *with rewind overlap*, on $T + 1$ tapes?

▶ **7.** [*26*] The initial distribution in example 7 of Chart A produces

$$(A_1D_1)^{11} \qquad D_1(A_1D_1)^{10} \qquad D_1(A_1D_1)^{9} \qquad D_1(A_1D_1)^{7}$$

on tapes 1–4, where $(A_1D_1)^7$ means $A_1D_1A_1D_1A_1D_1A_1D_1A_1D_1A_1D_1A_1D_1$. Show how to insert additional A_0's and D_0's in a "best possible" way (in the sense that the overall number of initial runs processed while merging is minimized), bringing the distribution up to

$$A(DA)^{14} \qquad (DA)^{28} \qquad (DA)^{26} \qquad (DA)^{22}.$$

Hint: To preserve parity it is necessary to insert many of the A_0's and D_0's as adjacent

pairs. The merge numbers for each initial run may be computed as in exercise 5.4.4–5; some simplification occurs since adjacent runs always have adjacent merge numbers.

8. [*20*] Chart A shows that most of the schemes for initial distribution of runs (with the exception of the initial distribution for the cascade merge) tend to put consecutive runs onto different tapes. If consecutive runs went onto the same tape we could save the stop/start time; would it therefore be a good idea to modify the distribution algorithms so that they switch tapes less often?

▸ **9.** [*22*] Estimate how long the read-backward polyphase algorithm would have taken in Chart A, if we had used all $T = 6$ tapes for sorting, instead of $T = 5$ as in example 7. Was it wise to avoid using the input tape?

10. [*M23*] Use the analyses in Sections 5.4.2 and 5.4.3 to show that the length of each rewind during a standard six-tape polyphase or cascade merge is rarely more than about 54 percent of the file (except for the initial and final rewinds, which cover the entire file).

11. [*23*] By modifying the appropriate entries in Table 1, estimate how long the first nine examples of Chart A would have taken if we had a combined low speed/high speed rewind. Assume that $\rho = 1$ when the tape is less than about one-fourth full, and that the rewind time for fuller tapes is approximately five seconds plus the time which would be obtained for $\rho = \frac{1}{5}$. Change example 8 so that it uses cascade merge *with* copying, since rewinding and reading forward is slower than copying in this case. [*Hint:* Use the result of exercise 10.]

12. [*40*] Consider partitioning six tapes into three pairs of tapes, with each pair playing the role of a single tape in a polyphase merge with $T = 3$. One tape of each pair will contain blocks 1, 3, 5, . . . and the other tape will contain blocks 2, 4, 6, . . . ; in this way we can essentially have two input tapes and two output tapes active at all times while merging, effectively doubling the merging speed. (a) Find an appropriate way to extend Algorithm F to this situation. How many buffers should there be? (b) Estimate the total running time which would be obtained if this method were used to sort 100,000 100-character records, considering both the read-forward and read-backward cases.

13. [*20*] Can a five-tape oscillating sort, as defined in Algorithm 5.4.5B, be used to sort four reelfulls of input data, up until the time of the final merge?

14. [*M19*] Derive (10).

15. [*HM29*] Prove that $g_Q(z) = h_Q(z)/(1 - z)$, where $h_Q(z)$ is a rational function of z having no singularities inside the unit circle; hence $a_n^{(Q)} = h_Q(1)n + O(1)$ as $n \to \infty$. In particular, show that

$$h_3(1) = \det\begin{pmatrix} 0 & -p_0 & 0 & 0 \\ 0 & 1-p_1 & -p_0 & 0 \\ 0 & -p_2 & 1-p_1 & -p_0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \Big/ \det\begin{pmatrix} 1 & -p_0 & 0 & 0 \\ 1 & 1-p_1 & -p_0 & 0 \\ 1 & -p_2 & 1-p_1 & -p_0 \\ 0 & 0 & -1 & 1 \end{pmatrix}.$$

16. [*M46*] Analyze merging with $P + Q$ input buffers more carefully than is done in the text, using a more accurate model.

17. [*41*] Carry out detailed studies of the problem of sorting 100,000 100-character records, drawing charts such as those in Chart A, assuming that 3, 4, or 5 tapes are available.

*5.4.7. External Radix Sorting

The previous sections have discussed the process of tape sorting by merging; but there is another way to sort with tapes, based on the radix sorting principle used in mechanical card sorters (cf. Section 5.2.5). This method is sometimes called distribution sorting, column sorting, pocket sorting, digital sorting, separation sorting, etc.; it turns out to be essentially the *opposite* of merging!

Suppose, for example, that we have four tapes and that there are only eight possible keys: 0, 1, 2, 3, 4, 5, 6, 7. If the input data is on tape T1, we can begin by transferring all even keys to T3, all odd keys to T4:

	T1	T2	T3	T4
Given	{0, 1, 2, 3, 4, 5, 6, 7}	—	—	—
Pass 1	—	—	{0, 2, 4, 6}	{1, 3, 5, 7}

Now we rewind, and read T3 and then T4, putting {0, 1, 4, 5} on T1 and {2, 3, 6, 7} on T2:

Pass 2	{0, 4} {1, 5}	{2, 6} {3, 7}	—	—

(The notation "{0, 4} {1, 5}" stands for a file which contains some records whose keys are all 0 or 4, followed by records whose keys are all 1 or 5. Note that T1 now contains those keys whose middle binary digit is 0.) After rewinding again and distributing 0, 1, 2, 3 to T3 and 4, 5, 6, 7 to T4, we have

Pass 3			{0} {1} {2} {3}	{4} {5} {6} {7}

Now we can finish up by copying T4 to the end of T3. In general, if the keys range from 0 to $2^k - 1$, we could sort the file in an analogous way using k passes, followed by a final "collection" phase which copies about half of the data from one tape to another. With six tapes we could use radix 3 representations in a similar way, to sort keys from 0 to $3^k - 1$ in k passes.

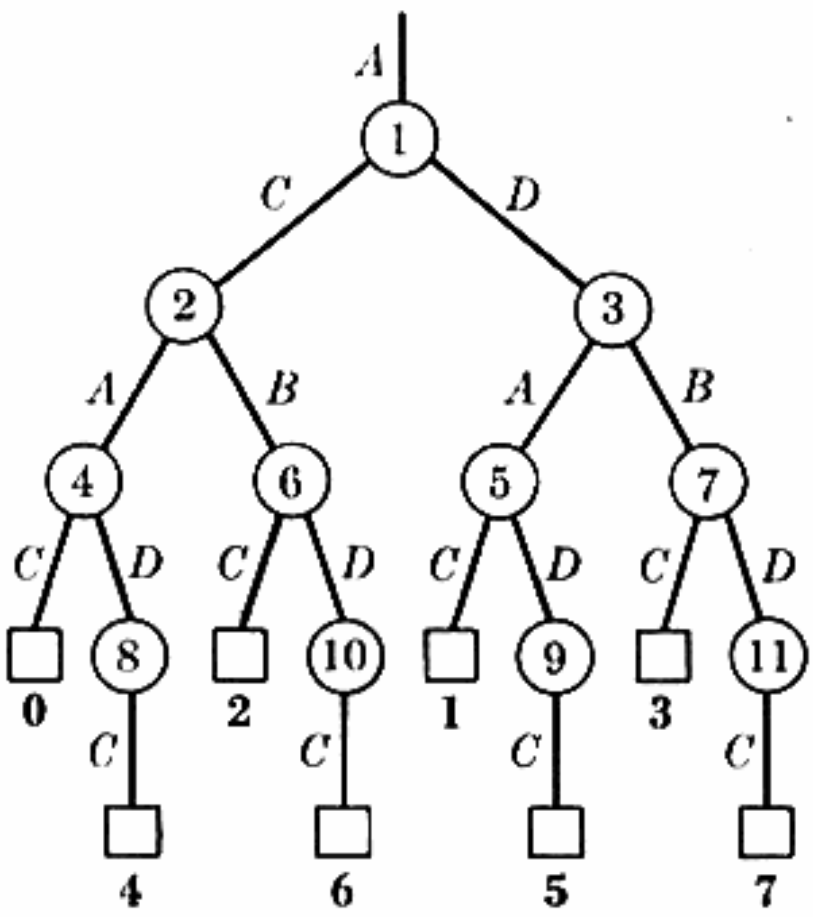
Partial-pass methods can also be used. For example, suppose that there are ten possible keys {0, 1, . . . , 9}, and consider the following procedure due to R. L. Ashenhurst [*Theory of Switching* 7 (Harvard Univ. Comp. Laboratory: May, 1954), I.1–I.76]:

	T1	T2	T3	T4	
Given	{0, 1, . . . , 9}	—	—	—	
Phase 1.	—	{0, 2, 4, 7}	{1, 5, 6}	{3, 8, 9}	1.0 pass
Phase 2.	{0}	—	{1, 5, 6} {2, 7}	{3, 8, 9} {4}	0.4 pass
Phase 3.	{0} {1} {2}	{6} {7}	—	{3, 8, 9} {4} {5}	0.5 pass
Phase 4.	{0} {1} {2} {3}	{6} {7} {8}	{9}	{4} {5}	0.3 pass
Collection	{0} {1} {2} {3} {4} . . . {9}				0.6 pass
					2.8 passes

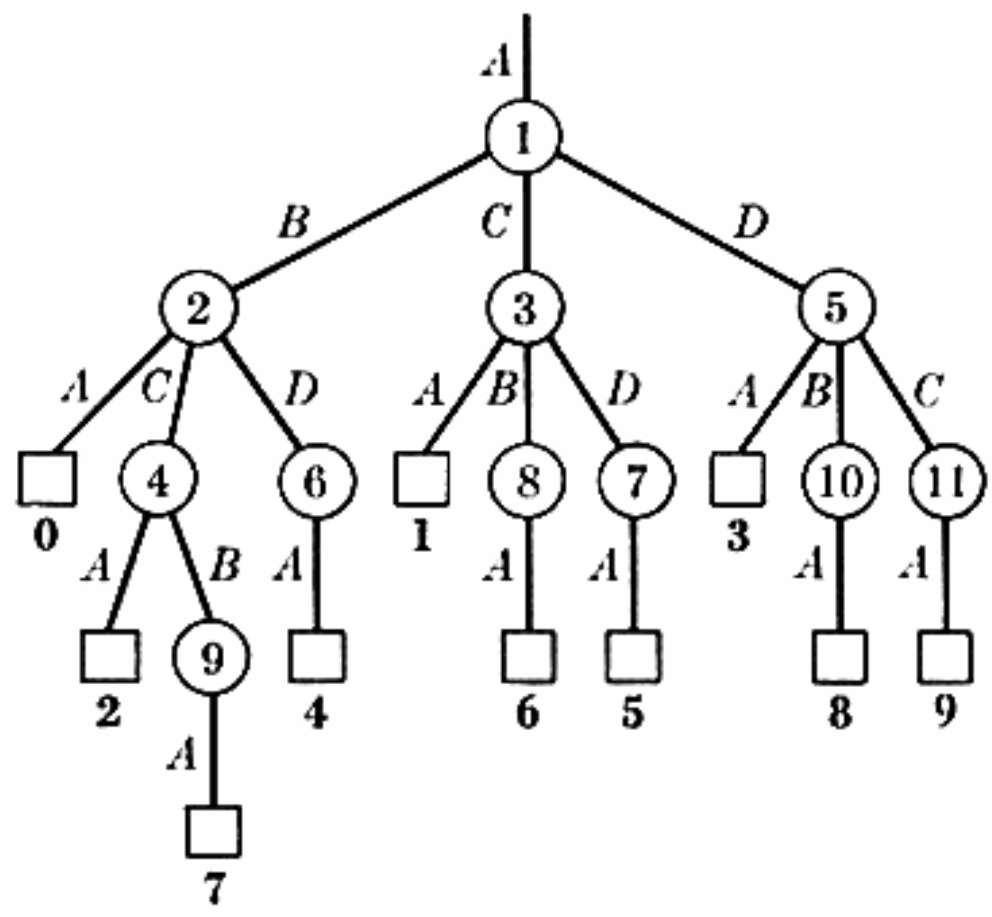
If each key value occurs about one-tenth of the time, the above procedure takes only 2.8 passes to sort ten keys, while the first example required 3.5 passes to sort only eight keys. Therefore we find that a clever distribution pattern can make a significant difference, for radix sorting as well as for merging.

The distribution patterns in the above examples can conveniently be represented as tree structures:

Example 1



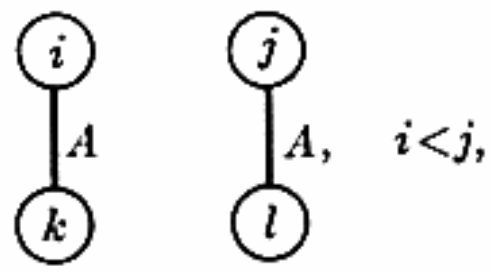
Example 2



The circular internal nodes of these trees are numbered 1, 2, 3, . . . , corresponding to steps 1, 2, 3, . . . of the process. Tape names A, B, C, D (instead of T1, T2, T3, T4) have been placed next to the lines of the trees, in order to show where the records go. Square external nodes represent portions of a file which contain only one key, and that key is shown in boldface type just below the node. The lines just above square nodes all carry the name of the output tape (C in the first example, A in the second).

Thus, step 3 of example 1 consists of reading from tape D and writing 1's and 5's on tape C, 3's and 7's on tape D. It is not difficult to see that the number of passes performed is equal to the *external path length* of the tree divided by the number of external nodes, if we assume that each key occurs equally often.

Because of the sequential nature of tape, and the first-in-first-out discipline of forwards reading, we can't simply use *any* labeled tree as the basis of a distribution pattern. In the tree of example 1, data gets written on tape A during step 2 and step 3; it is necessary to use the data written during step 2 before we use the data written during step 3. In general if we write onto a tape during steps i and j, where $i < j$, we must use the data written during step i first; when the tree contains two branches of the form



we must have $k < l$. Furthermore we cannot write anything onto tape A *between* steps k and l, because we must rewind between reading and writing.

The reader who has worked the exercises of Section 5.4.4 will now immediately perceive that the allowable trees for read-forward radix sorting on T tapes are precisely the "strongly T-fifo" trees which characterize read-forward *merge* sorting on T tapes! (See exercise 5.4.4–20.) The only difference is that all of the external nodes on the trees we are considering here have the same tape labels. We could remove this restriction by assuming a final "collection" phase which transfers all records to an output tape, or we could add that restriction to the rules for T-fifo trees by requiring that the initial distribution pass of a merge sort be explicitly represented in the corresponding merge tree.

In other words, *every merge pattern corresponds to a distribution pattern, and every distribution pattern corresponds to a merge pattern.* A moment's reflection shows why this is so: Consider running a merge sort backwards, "unmerging" the final output into subfiles which are unmerged into others, etc.; finally we will have unmerged the file into S runs. Such a pattern is possible with tapes if and only if the corresponding radix sort distribution pattern, for S keys, is possible. This duality between merging and distribution is almost perfect; it breaks down only in one respect, namely that the input tape must be saved at different times.

The eight-key example treated at the beginning of this section is clearly dual to a balanced merge on four tapes. The ten-key example with partial passes corresponds to the following ten-run merge pattern (if we suppress the copy phases, steps 6–11 in the tree):

	T1	T2	T3	T4
Initial distribution	1^4	1^3	1^1	1^2
Tree step 5	1^3	1^2	—	$1^2 3^1$
Tree step 4	1^2	1^1	2^1	$1^2 3^1$
Tree step 3	1^1	—	$2^1 3^1$	$1^1 3^1$
Tree step 2	—	4^1	3^1	3^1
Tree step 1	10^1	—	—	—

If we compare this to the radix sort, we see that the methods have essentially the same structure but are reversed in time, with the tape contents also reversed from back to front: $1^2 3^1$ (two runs each of length 1 followed by one of length 3) corresponds to $\{3, 8, 9\}\{4\}\{5\}$ (two subfiles each containing 1 key preceded by one subfile containing 3).

Going the other way, we can in principle construct a radix sort which is dual to polyphase merge, cascade merge, etc. For example, the 21-run polyphase merge on three tapes, illustrated at the beginning of Section 5.4.2, corresponds to the following interesting radix sort:

	T1	T2	T3
Given	{0, 1, . . . , 20}	—	—
Phase 1.	—	{0, 2, 4, 5, 7, 9, 10, 12, 13, 15, 17, 18, 20}	{1, 3, 6, 8, 11, 14, 16, 19}
Phase 2.	{0, 5, 10, 13, 18}	—	{1, 3, 6, 8, 11, 14, 16, 19} {2, 4, 7, 9, 12, 15, 17, 20}
Phase 3.	{0, 5, 10, 13, 18} {1, 6, 11, 14, 19} {2, 7, 12, 15, 20}	{3, 8, 16} {4, 9, 17}	—
Phase 4.	—	{3, 8, 16} {4, 9, 17} {5, 10, 18} {6, 11, 19} {7, 12, 20}	{0, 13} {1, 14} {2, 15}
Phase 5.	{8} {9} {10} {11} {12}	—	{0, 13} {1, 14} {2, 15} {3, 16} . . . {7, 20}
Phase 6.	{8} {9} {10} {11} {12} {13} . . . {20}	{0} {1} . . . {7}	—

The distribution rule used here to decide which keys go on which tapes at each step appears to be magic, but in fact it has a simple connection with the Fibonacci number system! (See exercise 2.)

Reading backward. Duality between radix sorting and merging applies also to algorithms that read tapes backward. We have defined "T-lifo trees" in Section 5.4.4, and it is easy to see that they correspond to radix sorts as well as to merge sorts.

A read backward radix sort was actually considered by John Mauchly already in 1946, in one of the first papers ever to be published about sorting (cf. Section 5.5); Mauchly essentially gave the following construction:

	T1	T2	T3	T4
Given	—	{0, 1, 2, . . . , 9}	—	—
Phase 1.	{4, 5}	—	{2, 3, 6, 7}	{0, 1, 8, 9}
Phase 2.	{4, 5} {2, 7}	{3, 6}	—	{0, 1, 8, 9}
Phase 3.	{4, 5} {2, 7} {0, 9}	{3, 6} {1, 8}	—	—
Phase 4.	{4, 5} {2, 7}	{3, 6} {1, 8}	{9}	{0}
. . . . .				
Phase 8.	—	—	{9} {8} {7} {6} {5}	{0} {1} {2} {3} {4}
Final collection	—	—	—	{0} {1} {2} {3} {4} {5} . . . {9}

This scheme is not the most efficient one possible, but it is interesting because it shows that partial pass methods were considered for radix sorting already in 1946, although they did not appear in the literature for merging until about 1960.

An efficient construction of read-backward distribution patterns has been suggested by A. Bayes [*CACM* **11** (1968), 491–493]: Given $P + 1$ tapes and S keys, divide the keys into P subfiles each containing $\lfloor S/P \rfloor$ or $\lceil S/P \rceil$ keys, and apply this procedure recursively to each subfile. When $S < 2P$, one subfile should consist of the smallest key alone, and it should be written onto the output file. (R. M. Karp's general preorder construction, which appears at the end of Section 5.4.4, includes this method as a special case.)

Backward reading makes merging a little more complicated because it reverses the order of runs. There is a corresponding effect on radix sorting: the outcome is stable or "anti-stable" depending on what level is reached in the tree. After a read-backward radix sort in which some of the external nodes are at odd levels and some are at even levels, the relative order of different

records with equal keys will be the *same* as the original order for some keys, but it will be the *opposite* of the original order for the other keys. (Cf. exercise 6.)

Oscillating merge sorts have their counterparts too, under duality. In an *oscillating radix sort* we continue to separate out the keys until reaching subfiles that have only one key or are small enough to be internally sorted; such subfiles are sorted and written onto the output tape, then the separation process is resumed. For example, if we have three work tapes and one output tape, and if the keys are binary numbers, we may start by putting keys of the form '0x' on tape T1, keys '1x' on T2. If T1 receives more than one memory load, we scan it again and put '00x' on T2 and '01x' on T3. Now if the '00x' subfile is short enough to be internally sorted, we do so and output the result, then continue by processing the '01x' subfile. Such a method was called a "cascading pseudo-radix sort" by E. H. Friend [*JACM* **3** (1956), 157–159]; it has been further developed by H. Nagler [*JACM* **6** (1959), 459–468], who gave it the colorful name "amphisbaenic sort," and by C. H. Gaudette [*IBM Tech. Disclosure Bull.* **12** (April, 1970), 1849–1853].

Does radix sorting beat merging? One important consequence of the duality principle is that *radix sorting is usually inferior to merge sorting*. This happens because the technique of replacement selection gives merge sorting a definite advantage; there is no apparent way to arrange radix sorts so that we can make use of internal sorts encompassing more than one memory load at a time. Indeed, the oscillating radix sort will often produce subfiles which are somewhat smaller than one memory load, so the distribution pattern corresponds to a tree with many more external nodes than would be present if merging and replacement selection were used. Consequently the external path length of the tree (i.e., the sorting time) is increased. (Cf. exercise 5.3.1–33.)

There are, however, some important applications for external radix sorting. Suppose for example that we have a file containing the names of all employees of a large corporation, in alphabetic order; the corporation has 10 divisions, and it is desired to sort the file by division, *retaining* the alphabetic order of the employees in each division. This is perfect situation in which to apply a stable radix sort, if the file is long, since the number of records which belong to each of the 10 divisions is likely to be more than the number of records which would be obtained in initial runs produced by replacement selection. In general, if the range of key values is so small that the collection of records having a given key would fill the internal memory more than twice, it is wise to use a radix sort technique.

We have seen in Section 5.2.5 that *internal* radix sorting is superior to merging, on certain high-speed computers, because the "inner loop" of the radix sort algorithm avoids complicated branching. If the external memory is especially fast, it may be impossible for such machines to merge data rapidly enough to keep up with the input/output equipment. Radix sorting may therefore turn out to be superior to merging in such a situation, especially if the keys are known to be uniformly distributed.

EXERCISES

1. [*20*] The general T-tape balanced merge with parameter P, $1 \leq P < T$, was defined near the beginning of Section 5.4. Show that this corresponds to a radix sort which uses a mixed radix system.

2. [*M28*] The text illustrates the three-tape polyphase radix sort for 21 keys. Generalize to the case of F_n keys; explain what keys appear on what tapes at the end of each phase. [*Hint:* Consider the Fibonacci number system, exercise 1.2.8–34.]

3. [*M40*] Extend the results of exercise 2 to the polyphase merge on four or more tapes. (Cf. exercise 5.4.2–10.)

4. [*M20*] Prove that Ashenhurst's distribution pattern is the best way to sort 10 keys on four tapes without reading backwards, in the sense that the associated tree has minimum external path length over all "strongly 4-fifo trees." (Thus, it is essentially the best method if we ignore rewind time.)

5. [*15*] Draw the 4-lifo tree corresponding to Mauchly's read-backwards radix sort for 10 keys.

▸ **6.** [*20*] A certain file contains two-digit keys 00, 01, . . . , 99. After performing Mauchly's radix sort on the unit's digit, we can repeat the same scheme on the ten's digits, interchanging the roles of tapes T2 and T4. In what order will the keys finally appear on T2?

7. [*21*] Does the duality principle apply also to multi-reel files?

Since we need three tapes to carry out a merge process without excessive tape motion, it is interesting to speculate about how we could perform a reasonable external sort using only two tapes.

One approach, suggested by H. B. Demuth in 1956, is sort of a combined replacement-selection and bubble sort. Assume that the input is on tape T1, and begin by reading $P + 1$ records into memory. Now output the record whose key is smallest, to tape T2, and replace it by the next input record. Continue outputting a record whose key is currently the smallest in memory, maintaining a selection tree or a priority queue of $P + 1$ elements. When the input is finally exhausted, the largest P keys of the file will be present in memory; output them is ascending order. Now rewind both tapes and repeat the process by reading from T2 and writing to T1; each such pass puts at least P more records into their proper place. A simple test can be built into the program which determines when the entire file is in sort. At most $\lceil (N - 1)/P \rceil$ passes will be necessary.

A few moment's reflection shows that each pass of this procedure is essentially equivalent to P consecutive passes of the bubble sort (Algorithm 5.2.2B)! If an element has P or more inversions, it will be smaller than everything in the tree when it is input, so it will be output immediately (thereby

losing P inversions). If an element has less than P inversions, it will go into the selection tree and will be output before all greater keys (thereby losing all its inversions). When $P = 1$, this is exactly what happens in the bubble sort, by Theorem 5.2.2I.

The total number of passes will therefore be $\lceil I/P \rceil$, where I is the maximum number of inversions of any element. By the theory developed in Section 5.2.2, the average value of I is $N - \sqrt{\pi N/2} + \frac{2}{3} + O(1/\sqrt{N})$.

If the file is not too much larger than the memory size, or if it is nearly in order to begin with, this order-P bubble sort will be fairly rapid; in fact, it might be advantageous to use it even when extra tape units are available, since it might not take as long as would be required for the computer operator to mount a third tape! On the other hand, it will run quite slowly on fairly long, randomly ordered files, since its running time is approximately proportional to N^2.

Let us consider how this method might be implemented for the 100,000-record example of Section 5.4.6. We need to choose P intelligently, in order to compensate for interblock gaps while doing simultaneous reading, writing, and computing. Since the example assumes that each record is 100 characters long and that 100,000 characters will fit into memory, we can make room for two input buffers and two output buffers of size B by setting

$$100(P + 1) + 4B = 100000. \tag{1}$$

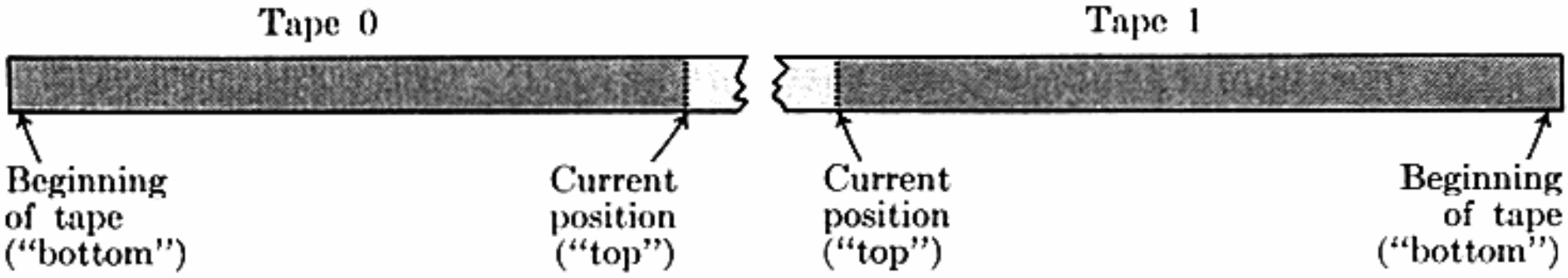
Using the notation of Section 5.4.6, the running time for each pass will be about

$$NC\omega\tau(1 + \rho), \qquad \omega = (B + \gamma)/B. \tag{2}$$

Since the number of passes is inversely proportional to P, we want to choose B to be a multiple of 100 which minimizes the quantity ω/P. Elementary calculus shows that this occurs when B is approximately $\sqrt{24975\gamma + \gamma^2} - \gamma$, so we take $B = 3000$, $P = 879$. Setting $N = 100000$ in the above formulas shows that the number of passes $\lceil I/P \rceil$ will be about 114, and the total estimated running time will be approximately 8.57 hours (assuming for convenience that the initial input and the final output also have $B = 3000$). This represents approximately 0.44 of a reelfull of data; a full reel would take about five times as long. Some improvements could be made if the algorithm were interrupted periodically, writing the records with largest keys onto an auxiliary tape which is dismounted, since such records are merely copied back and forth once they have been put into order.

Application of quicksort. Another internal sorting method which traverses the data in a nearly sequential manner is the partition exchange or quicksort procedure, Algorithm 5.2.2Q. Can we adapt it to two tapes? [N. B. Yoash, *CACM* **8** (1965), 649.]

It is not difficult to see how this can be done, using backward reading. Assume that the two tapes are numbered 0 and 1, and imagine that the file is laid out as follows:



Each tape serves as a stack; putting them together like this makes it possible to view the file as a linear list in which we can move the current position left or right by copying from one stack to the other. The following recursive subroutines define a suitable sorting procedure:

- SORT00 [Sort the top subfile on tape 0 and return it to tape 0].

If the subfile fits in the internal memory, sort it internally and return it to tape. Otherwise select one record R from the subfile, and let its key be K. Reading backward on tape 0, copy all records whose key is $> K$, forming a new subfile on the top of tape 1. Now read forward on tape 0, copying all records whose key is $= K$ onto tape 1. Then read backward again, copying all records whose key is $< K$ onto tape 1. Complete the sort by executing SORT10 on the $< K$ keys, then copying the $= K$ keys to tape 0, and finally executing SORT10 on the $> K$ keys.

- SORT01 [Sort the top subfile on tape 0 and write it on tape 1].

Same as SORT00, but the final "SORT10" is changed to "SORT11" followed by copying the $\leq K$ keys to tape 1.

- SORT10 [Sort the top subfile on tape 1 and write it on tape 0].

Same as SORT10, interchanging 0 with 1 and $<$ with $>$.

- SORT11 [Sort the top subfile on tape 1 and return it to tape 1].

Same as SORT00, interchanging 0 with 1 and $<$ with $>$.

The recursive nature of these subroutines can be handled without difficulty by storing appropriate control information on the tapes.

The running time for this algorithm can be estimated as follows, if we assume that the data are in random order, with negligible probability of equal keys. Let M be the number of records that fit into internal memory. Let X_N be the average number of records read while applying SORT00 or SORT11 to a subfile of N records, when $N > M$, and let Y_N be the corresponding quantity for SORT01 or SORT10. Then we have

$$X_N = \begin{cases} 0, & \text{if } N \leq M; \\ 3N + 1 + \frac{1}{N} \sum_{0 \leq k < N} (Y_k + Y_{N-1-k}), & \text{if } N > M; \end{cases}$$

$$Y_N = \begin{cases} 0, & \text{if } N \leq M; \\ 3N + 2 + \frac{1}{N} \sum_{0 \leq k < N} (Y_k + X_{N-1-k} + k), & \text{if } N > M. \end{cases} \quad (3)$$

The solution to these recurrences (see exercise 2) shows that the total amount of tape reading during the external partitioning phases will be $6\frac{2}{3}N \ln N + O(N)$, on the average, as $N \to \infty$. We also know from Eq. 5.2.2–25 that the average number of internal sort phases will be $2(N+1)/(M+2) - 1$.

If we apply this analysis to the 100,000-record example of Section 5.4.6, using 25,000-character buffers and assuming that the sorting time is $2nC\omega\tau$ for a subfile of $n \leq M = 1000$ records, we obtain an average sorting time of approximately 103 minutes (including the final rewind as in Chart A). Thus the quicksort method isn't bad, on the average; but of course its *worst* case turns out to be even more awful than the bubble sort discussed above.

Radix sorting. The radix exchange method (Algorithm 5.2.2R) can be adapted to two-tape sorting in a similar way, since it is so much like quicksort. The "trick" which makes both of these methods work is the idea of reading a file more than once, something we never did in our previous tape algorithms.

The same trick can be used to do a conventional "least-significant-digit-first" radix sort on two tapes. Given the input data on T1, we copy all records onto T2 whose key ends with 0 in binary notation; then after rewinding T1 we read it again, copying the records whose key ends with 1. Now both tapes are rewound and a similar pair of passes is made, interchanging the roles of T1 and T2, and using the *second* least significant binary digit. At this point T1 will contain all records whose keys are $(\ldots 00)_2$, followed by those whose keys are $(\ldots 01)_2$, then $(\ldots 10)_2$, then $(\ldots 11)_2$. If the keys are b bits long, we need only $2b$ passes over the file in order to complete the sort.

Such a radix sort could be applied only to the *leading* b bits of the keys, for some judiciously chosen number b; this would reduce the number of inversions by a factor of about 2^b, if the keys were uniformly distributed, so a few passes of the P-way bubble sort could be used to complete the job.

A novel but somewhat more complicated approach to two-tape distribution sorting has been suggested by A. I. Nikitin and L. I. Sholmov [*Kibernetika* **2**, 6 (1966), 79–84]. Counts are made of the number of keys having each possible configuration of leading bits, and artificial keys $\kappa_1, \kappa_2, \ldots, \kappa_M$ based on these counts are constructed so that the number of actual keys lying between κ_i and κ_{i+1} is between predetermined limits P_1 and P_2, for each i. Thus, M lies between $\lceil N/P_2 \rceil$ and $\lceil N/P_1 \rceil$. If the leading bit counts do not give sufficient information to determine such $\kappa_1, \kappa_2, \ldots, \kappa_M$, one or more further passes are made to count the frequency of less significant bit patterns, for certain configurations of most significant bits. After the table of artificial keys $\kappa_1, \kappa_2, \ldots, \kappa_M$ have been constructed, $2\lceil \log_2 M \rceil$ passes will suffice to complete the sort. (This method requires memory space proportional to N, so it can't be used for external sorting as $N \to \infty$. In practice we would not use the technique for multireel files, so M will be comparatively small and the table of artificial keys will fit comfortably in memory.)

Simulation of more tapes. F. C. Hennie and R. E. Stearns have devised a general technique for simulating k tapes on only two tapes, in such a way that

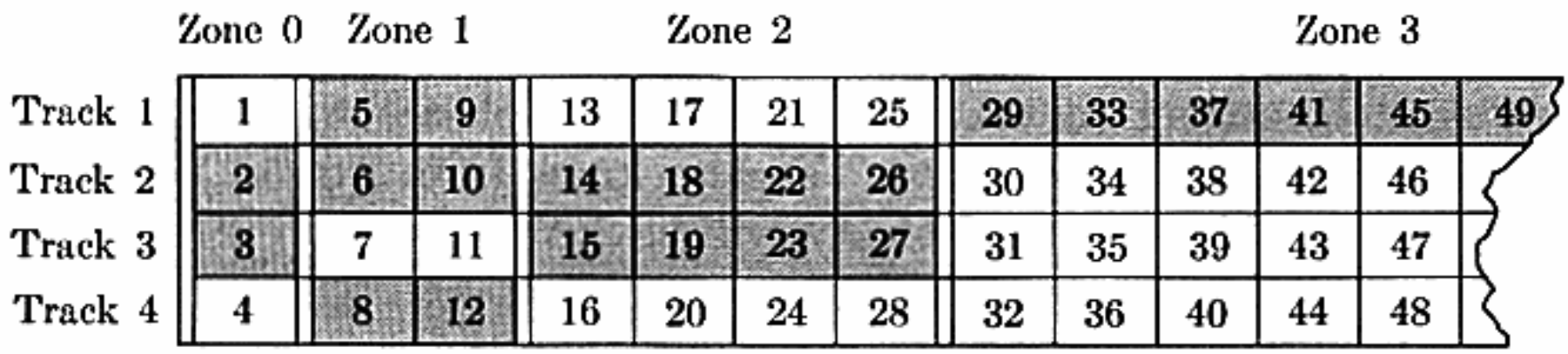


Fig. 86. Layout of tape T1 in the Hennie-Stearns construction; nonblank zones are shaded.

the tape motion required is increased by a factor of only $O(\log L)$, where L is the maximum distance to be traveled on any one tape [*JACM* **13** (1966), 533–546]. Their construction can be simplified slightly in the case of sorting, as in the following method suggested by R. M. Karp.

We shall simulate an ordinary four-tape balanced merge, using two tapes T1 and T2. The first of these, T1, holds the simulated tape contents in a way that may be diagrammed as in Fig. 86; we imagine that the data is written in four "tracks," one for each simulated tape. (In actual fact the tape doesn't have such tracks; blocks 1, 5, 9, 13, . . . are thought of as track 1, and blocks 2, 6, 10, 14, . . . as track 2, etc.) The other tape, T2, is used only for auxiliary storage, to help move things around on T1.

The blocks of each track are divided into *zones*, containing, respectively, 1, 2, 4, 8, . . . , 2^k, . . . blocks per zone. Zone k on each track is either filled with exactly 2^k blocks of data, or it is completely "blank." In Fig. 86, for example, track 1 has data in zones 1 and 3; track 2 in zones 0, 1, 2; track 3 in zones 0 and 2; track 4 in zone 1; and the other zones are blank.

Suppose that we are merging data from Tracks 1 and 2 to Track 3. The internal computer memory contains two buffers used for input to a two-way merge, plus a third buffer for output. When the input buffer for Track 1 becomes empty, we can refill it as follows: Find the first nonempty zone on Track 1, say zone k, and copy its first block into the input buffer; then copy the other $2^k - 1$ blocks of data onto T2, and move them to zones 0, 1, . . . , $k - 1$ of track 1. (Zones 0, 1, . . . , $k - 1$ are now full and zone k is blank.) An analogous procedure is used to refill the input buffer for Track 2, whenever it becomes empty. When the output buffer is ready to be written on Track 3, we reverse the process, scanning across T1 to find the first *blank* zone on T3, say zone k, while copying the data from zones 0, 1, . . . , $k - 1$ onto T2. The data on T2, augmented by the contents of the output buffer, is now used to fill zone k of Track 3.

This procedure requires the ability to write in the middle of tape T1, without destroying subsequent information on that tape. As in the case of read-forward oscillating sort (Section 5.4.5), it is possible to do this reliably if suitable precautions are taken.

The amount of tape motion required to bring $2^l - 1$ blocks of Track 1 into memory is $\sum_{0 \le k < l} 2^{l-1-k} \cdot c \cdot 2^k = cl2^{l-1}$, for some constant c, since we scan up to zone k only once in every 2^k steps. Thus each merge pass requires

$O(N \log N)$ steps. Since there are $O(\log N)$ passes in a balanced merge, the the constant in $O(N(\log N)^2)$ is much too high to be satisfactory when N is asymptotically much better than the worst case of quicksort.

But this method wouldn't actually work very well if we applied it to the 100,000-record example of Section 5.4.6, since the information specified for tape T1 would overflow the contents of one tape reel. Even if we ignore this fact, and if we use optimistic assumptions about read/write/compute overlap and interblock gap lengths, etc., we find that roughly 37 hours would be required to complete the sort! So this method is purely of academic interest; the constant in $O(N(\log N)^2)$ is much too high to be satisfactory when N is in a practical range.

One-tape sorting. Could we live with only one tape? It is not difficult to see that the order-P bubble sort described above could be converted into a one-tape sort, but the result would be ghastly.

H. B. Demuth [Ph. D. thesis (Stanford University, 1956), 85] observed that a computer with bounded internal memory cannot reduce the number of inversions of a permutation by more than a bounded amount as it moves a bounded distance on tape, hence every one-tape sorting algorithm must take at least $d\,N^2$ units of time on the average (for some positive constant d which depends on the computer configuration).

R. M. Karp has pursued this topic in a very interesting way, discovering what amounts to an *optimum* way to sort with one tape. It is convenient to discuss Karp's algorithm by reformulating the problem as follows: *What is the fastest way to transport people between floors using a single elevator?*

Consider a building with n floors, having room for exactly c people on each floor. The building contains no doors, windows, or stairs, but it does have an elevator that can stop on each floor. There are cn people in the building, and exactly c of them want to go to each particular floor. The elevator holds at most b people, and it takes one unit of time to go from floor i to floor $i+1$. We wish to find the quickest way to get all the people onto the proper floors, if the elevator is required to start and finish on floor 1.

The connection between this elevator problem and one-tape sorting is not hard to see: The people are the records and the building is the tape. The floors are individual blocks on the tape, and the elevator is the internal computer memory. A computer program has more flexibility than an elevator operator (it can, for example, duplicate people, or temporarily chop them into two parts on different floors, etc.); but the solution below solves the problem in the fastest conceivable time without doing such operations.

The following two auxiliary tables are required by Karp's algorithm.

$$\begin{aligned} &u_k,\ 1 \le k \le n\text{: Number of people on floors} \le k \text{ whose destination is} > k;\\ &d_k,\ 1 \le k \le n\text{: Number of people on floors} \ge k \text{ whose destination is} < k. \end{aligned} \tag{4}$$

When the elevator is empty, we always have $u_k = d_{k+1}$ for $1 \le k < n$, since there are c people on every floor; the number of misfits on floors $\{1, \ldots, k\}$

must equal the corresponding number on floors $\{k+1, \ldots, n\}$. By definition, $u_n = d_1 = 0$.

It is clear that the elevator must make at least $\lceil u_k/b \rceil$ trips from floor k to floor $k+1$, for $1 \le k < n$, since only b passengers can ascend on each trip. Similarly it must make at least $\lceil d_k/b \rceil$ trips from floor k to floor $k-1$. Therefore the elevator must necessarily take at least

$$\sum_{1 \le k \le n} (\lceil u_k/b \rceil + \lceil d_k/b \rceil) \tag{5}$$

units of time on any correct schedule. Karp discovered that this lower bound can actually be achieved, when $u_1, \ldots, u_{n-1}$ are nonzero.

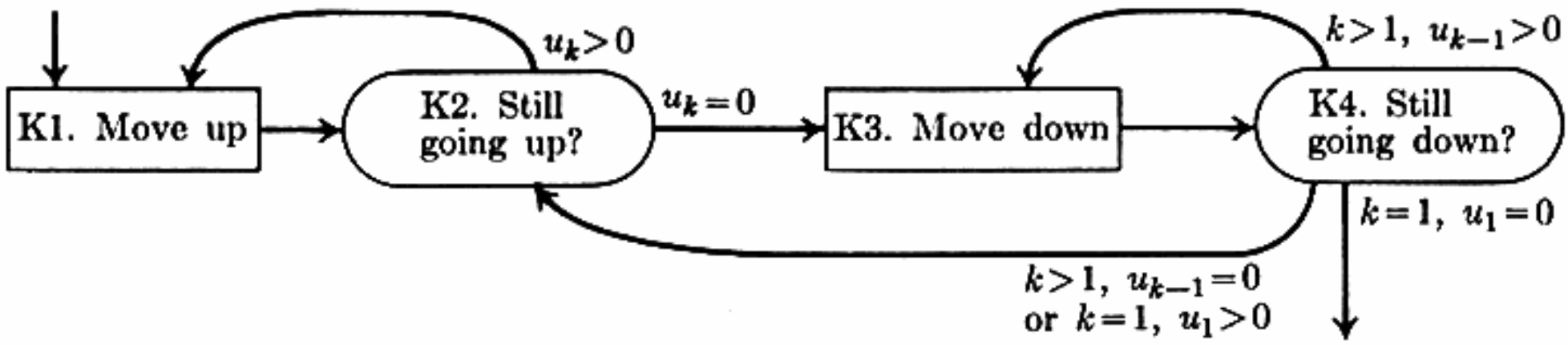


Fig. 87. Karp's elevator algorithm.

Theorem K. *If $u_k > 0$ for $1 \le k < n$, there is an elevator schedule which delivers everyone to his destination in the minimum time* (5).

Proof. Assume that there are b extra people in the building; they start in the elevator and their destination floor is artifically set to 0. The elevator can operate according to the following algorithm, starting with k (the current floor) equal to 1:

K1. [Move up.] From among the $b + c$ people currently in the elevator or on floor k, those b with the highest destinations get into the elevator, and the others remain on floor k.

Let there be u people now in the elevator whose destination is $> k$, and d whose destination is $\le k$. (It will turn out that $u = \min(b, u_k)$; if $u_k < b$ we may therefore be transporting some people away from their destination. This represents their sacrifice to the common good.) Decrease u_k by u, increase d_{k+1} by d, and then increase k by 1.

K2. [Still going up?] If $u_k > 0$, return to step K1.

K3. [Move down.] From among the $b + c$ people currently in the elevator or on floor k, those b with the lowest destinations get into the elevator, and the others remain on floor k.

Let there be u people now in the elevator whose destination is $\ge k$, and d whose destination is $< k$. (It will always turn out that $u = 0$ and $d = b$, but the algorithm is described here in terms of general u and d in order to make the proof a little clearer.) Decrease d_k by d, increase u_{k-1} by u, and then decrease k by 1.

K4. [Still going down?] If $k > 1$ and $u_{k-1} > 0$, return to step K3. If $k = 1$ and $u_1 = 0$, terminate the algorithm (everyone has moved to his destination and the b "extras" are back in the elevator). Otherwise return to step K2.

Figure 88 shows an example of this algorithm, with a nine-floor building and $b = 3$, $c = 2$. Note that one of the 6's is temporarily transported away from his destination in spite of the fact that the elevator travels the minimum possible distance. The idea of testing u_{k-1} in step K4 is the crucial thing which makes the algorithm work, as we shall see.

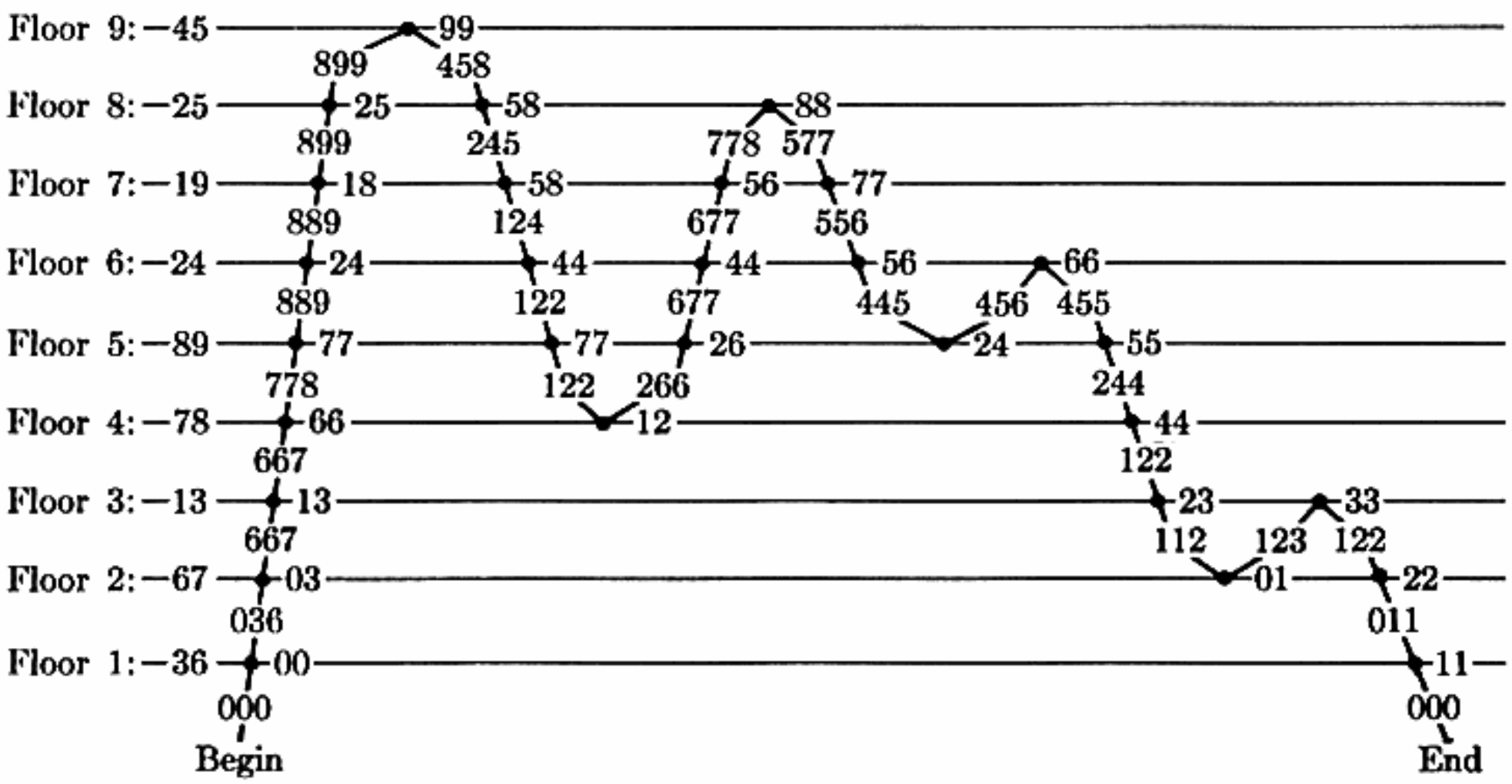


Fig. 88. An optimum way to rearrange people using a small, slow elevator. (Each person is represented by the number of his destination floor.)

To verify the validity of this algorithm, we note that steps K1 and K3 always keep the u and d tables (4) up to date, if we regard the people in the elevator as being on the "current" floor k. It is now possible to prove by induction that the following properties hold at the beginning of each step:

$$u_l = d_{l+1} \quad \text{for} \quad k \le l < n; \tag{6}$$

$$u_l = d_{l+1} - b, \quad \text{for} \quad 1 \le l < k; \tag{7}$$

$$\text{if} \quad u_l = 0 \quad \text{and} \quad k \le l < n \quad \text{then} \quad u_{l+1} = 0. \tag{8}$$

Furthermore, at the beginning of step K1, the min (u_k, b) people with highest destinations, among all people on floors $\le k$ with destination $> k$, are in the elevator or on floor k. At the beginning of step K3, the min (d_k, b) people with lowest destinations, among all people on floors $\ge k$ with destination $< k$, are in the elevator or on floor k. These conditions may also be verified inductively, by considering how we can reach step K1 or K3 (cf. exercise 5).

From these properties it follows that the parenthesized remarks in steps K1 and K3 are valid. Each execution of step K1 therefore decreases $\lceil u_k/b \rceil$ by 1 and leaves $\lceil d_{k+1}/b \rceil$ unchanged; each execution of K3 decreases $\lceil d_k/b \rceil$ by

1 and leaves $\lceil u_{k-1}/b \rceil$ unchanged. The algorithm must therefore terminate in a finite number of steps, and everybody must then be at his destination because of (6) and (8). ∎

When $u_k = 0$ and $u_{k+1} > 0$ we have a "disconnected" situation; the elevator must journey up to floor $k+1$ in order to rearrange the people up there, even though nobody wants to move from floors $\leq k$ to floors $\geq k+1$. Without loss of generality, we may assume that $u_{n-1} > 0$; then every valid elevator schedule must include at least

$$2 \sum_{1 \leq k < n} \max(1, \lceil u_k/b \rceil) \tag{9}$$

moves, since we require the elevator to return to floor 1. A schedule achieving this lower bound is readily constructed (exercise 4).

EXERCISES

1. [*17*] The order-P bubble sort discussed in the text uses only forward reading and rewinding. Can the algorithm be modified to take advantage of *backward* reading?

2. [*M26*] Find explicit "closed form" solutions for the numbers X_N, Y_N defined in (3). [*Hint:* Study the solution to Eq. 5.2.2–19.]

3. [*38*] Is there a two-tape sorting method, based only on comparisons of keys (not digital properties), whose tape motion is $O(N \log N)$ in the worst case, when sorting N records? [Quicksort achieves this on the average, but not in the worst case, and the Hennie-Stearns method (Fig. 86) achieves $O(N(\log(N)^2)$.]

4. [*M23*] In the elevator problem, suppose there are indices p, q with $q \geq p + 2$, $u_p > 0$, $u_q > 0$, and $u_{p+1} = \cdots = u_{q-1} = 0$. Explain how to construct a schedule requiring at most (9) units of time.

▸ **5.** [*M23*] True or false: After step K1 of the algorithm in Theorem K, nobody on the elevator has a lower destination than any person on floors $< k$.

6. [*M30*] (R. M. Karp.) Generalize the elevator problem (Fig. 87) to the case that there are c_j passengers initially on floor j, and c'_j passengers whose destination is floor j, for $1 \leq j \leq n$. Show that a schedule exists which takes $2\sum_{1 \leq k < n} \max(1, \lceil u_k/b \rceil, \lceil d_{k+1}/b \rceil)$ units of time, never allowing more than $\max(c_j, c'_j)$ passengers to be on floor j at any one time. [*Hint:* Introduce fictitious people, if necessary, to make $c_j = c'_j$ for all j.]

7. [*M40*] (R. M. Karp.) Generalize the problem of exercise 6, replacing the linear path followed by the elevator by a network of roads to be traveled by a bus, given that the network forms any *free tree*. The bus has finite capacity, and it is desired to transport passengers to their destinations in such a way that the bus travels a minimum distance.

8. [*M32*] Let $c = 1$ in the elevator problem treated in the text. How many permutations of the n people on the n floors will make $u_k \leq 1$ for $1 \leq k \leq n$ in (4)? [For example, 3 1 4 5 9 2 6 8 7 is such a permutation.]

▸ **9.** [*M25*] Find a significant connection between the "cocktail shaker sort" described

in Section 5.2.2, Fig. 16, and the numbers $u_1, u_2, \ldots, u_n$ of (4) in the case $c = 1$.

10. [*20*] How would you sort a multireel file with only two tapes?

5.4.9. Disks and Drums

So far we have considered tapes as the vehicles for external sorting, but more flexible types of mass storage devices are often available. Although such "bulk memory" or "direct-access storage" units come in many different forms, they may be roughly characterized by the following properties:

i) It doesn't take very long to access any specified part of the stored information;

ii) Blocks of consecutive words can be transmitted rapidly between the internal and external memory.

Magnetic tape satisfies (ii) but not (i), because it takes a long time to get from one end of a tape to the other. Some devices satisfy (i) but not (ii), e.g. bulk core memory units in which the access time to each word is about ten times slower than the internal memory speed.

Every external memory unit has idiosyncrasies which ought to be studied carefully before major programs are written for it; but technology changes so rapidly, it is impossible to give a complete discussion here of all the available varieties of hardware. Therefore we shall consider only some typical memory devices which illustrate useful approaches to the sorting problem.

One of the most common types of external memories satisfying (i) and (ii) is a disk file or disk pack module (see Fig. 89). Data is kept on a number of rapidly-rotating circular disks, covered with magnetic material; a comb-like access arm, containing one or more "read/write heads" for each disk surface, is used to store and retrieve the information. Each individual surface is divided into concentric rings called *tracks*, so that an entire track of data passes a read/write head every time the disk completes one revolution. The access arm can move in and out, shifting the read/write heads from track to track; but this motion takes time. A set of tracks which can be read or written without repositioning the access arm is called a *cylinder* or a *stratum*. For example, Fig. 89

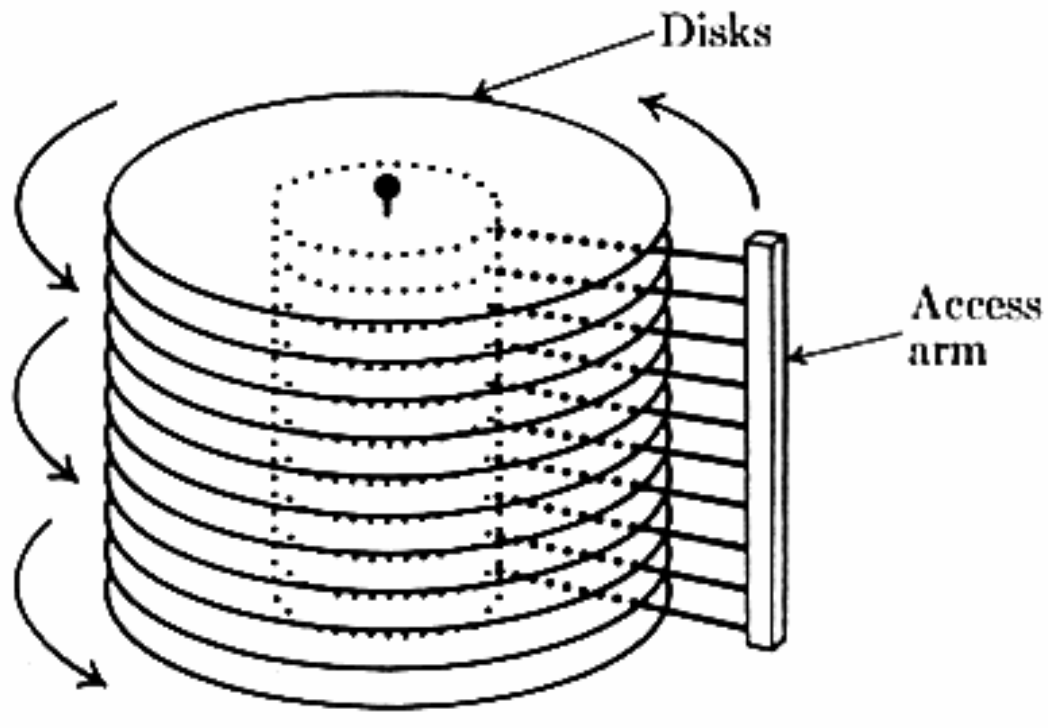


Fig. 89. A disk pack module.

illustrates a disk file which has just one read/write head per surface; the dotted lines show one of the cylinders, consisting of all tracks currently being scanned by the read/write heads.

To fix the ideas, let us consider hypothetical MIXTEC disk pack units, for which

$$\begin{aligned} 1 \text{ track} &= 5000 \text{ characters} \\ 1 \text{ cylinder} &= \quad 20 \text{ tracks} \\ 1 \text{ disk unit} &= \quad 200 \text{ cylinders} \end{aligned}$$

Such a disk unit contains 20 million characters, slightly less than the amount of data that can be stored on a single magnetic tape. On some machines, tracks near the center have fewer characters than tracks near the rim; this tends to make the programming much more complicated, and MIXTEC fortunately avoids such problems.

The amount of time required to read or write on a disk file is essentially the sum of three quantities,

- Seek time (the time to move the access arm to the proper cylinder);
- Latency time (rotational delay until the read/write head reaches the right spot);
- Transmission time (rotational delay while the data passes the read/write head).

On MIXTEC devices the seek time required to go from cylinder i to cylinder j is $25 + \frac{1}{2}|i - j|$ milliseconds. If i and j are randomly-selected integers between 1 and 200, the average value of $|i - j|$ is $2\binom{201}{3}/200^2 \approx 66.7$, so the average seek time is about 60 ms. MIXTEC disks rotate once every 25 ms, so the latency time averages about 12.5 ms. The transmission time for n characters is $(n/5000) \times 25 \text{ ms} = 5n\mu\text{s}$. (This is about $3\frac{1}{3}$ times as fast as the transmission rate of MIXT tapes used in the examples of Section 5.4.6.)

Thus the main differences between MIXTEC disks and MIXT tapes, relative to sorting, are

a) Tapes can only be accessed sequentially;

b) Individual disk operations tend to require significantly more overhead (seek time + latency time compared to stop/start time);

c) The disk transmission rate is faster.

By using clever merge patterns on tape, we were able to compensate somewhat for disadvantage (a). Our goal now is to think of some clever algorithms for disk sorting that will compensate for disadvantage (b).

Overcoming latency time. Let us consider first the problem of minimizing the delays caused by the fact that the disks aren't always positioned properly when we want to start an I/O command. We can't make the disk spin faster, but we can still apply some tricks that reduce or even eliminate all of the latency time.

The addition of more access arms would obviously help, but that would be an expensive hardware modification. Here are some "software" ideas:

1) If we read or write several tracks of a cylinder at a time, we avoid the latency time (*and* the seek time) on all tracks but the first. In general it is often possible to synchronize the computing time with the disk movement in such a way that a sequence of input/output instructions can be carried out without latency delays.

2) Consider the problem of reading half a track of data (Fig. 90): If the read command begins when the heads are at point A, there is no latency delay, and the total time for reading is just the transmission time, $\frac{1}{2} \times 25$ ms. If the command begins with the heads at B, we need $\frac{1}{4}$ of a revolution for latency and $\frac{1}{2}$ for transmission, totalling $\frac{3}{4} \times 25$ ms. The most interesting case occurs when the heads are initially at C: With proper hardware and software we need *not* waste $\frac{3}{4}$ of a revolution for latency delay. Reading can begin immediately, into the second half of the input buffer; then after a $\frac{1}{2} \times 25$ ms pause, reading can resume into the first half of the buffer, so that the instruction is completed when point C is reached again. In a similar manner, we can ensure that the total latency + transmission time will never exceed the time for one revolution, regardless of the initial position of the disk. The average amount of latency delay is reduced by this scheme from half a revolution to $\frac{1}{2}(1 - x^2)$ of a revolution, if we are reading or writing a given fraction x of a track, for $0 < x \leq 1$. When an entire track is being read or written ($x = 1$), this technique eliminates *all* the latency time.

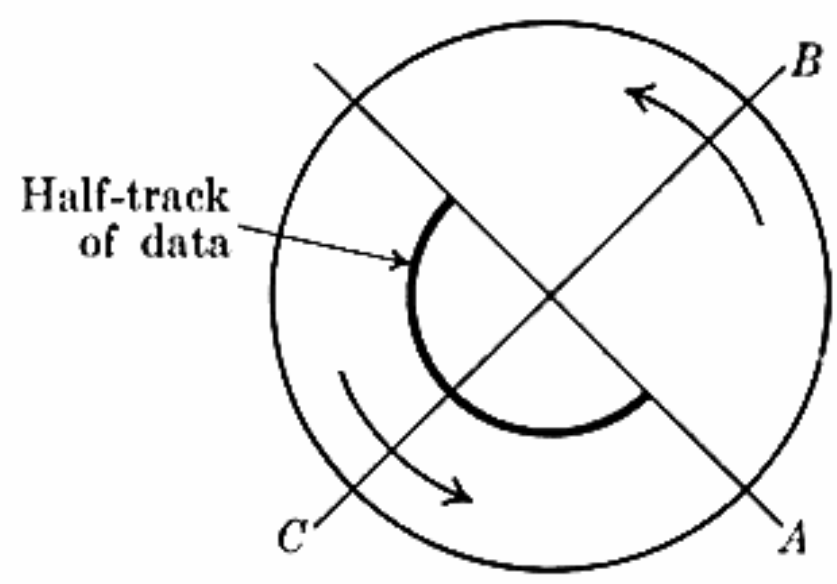


Fig. 90. Analysis of the latency time when reading half of a track.

Drums: The no-seek case. Some external memory units eliminate the seek time by having one read/write head for every track. If the technique of Fig. 90 is employed on such devices, both seek time and latency time reduce to zero, provided that we always read or write a track at a time; this is the ideal situation in which transmission time is the only limiting factor.

Let us consider again the example application of Section 5.4.6, sorting 100,000 records of 100 characters each, with a 100,000-character internal memory. The total amount of data to be sorted fills half of a MIXTEC disk. It is usually impossible to read and write simultaneously on a single disk unit; we shall assume that two disks are available, so that reading and writing can overlap

each other. For the moment we shall assume, in fact, that the "disks" are actually *drums*, containing 4000 tracks of 5000 characters each, with no seek time required.

What sorting algorithm should be used? The method of merging is a fairly natural choice; other methods of internal sorting do not lend themselves so well to a disk implementation, except possibly radix techniques, but the considerations of Section 5.4.7 show that radix sorting is usually inferior to merging for general applications. (It is not hard to see that the duality theorem of that section applies to disks as well as to tapes.)

To begin a merge sort for this problem we can use replacement selection, with two 5000-character input buffers and two 5000-character output buffers. In fact, it is possible to reduce this to *three* 5000-character buffers, if records in the current input buffer are replaced by records that come off the selection tree. That leaves 85,000 characters (850 records) for a selection tree, so one pass over our example data will form about 60 initial runs. (See Eq. 5.4.6–3.) This pass takes only about 50 seconds, if we assume that the internal processing time is fast enough to keep up with the input/output rate (one record moving to the output buffer every 500 microseconds). If the input to be sorted appeared on a MIXT tape, instead of on a drum, this pass would be slower, governed by the tape speed.

With two drums and full-track reading/writing, it is not hard to see that the total transmission time for P-way merging is minimized if we let P be as large as possible. Unfortunately we can't simply do a 60-way merge on all of the initial runs, since there isn't room for 60 buffers in memory. (A buffer of less than 5000 characters would introduce unwanted latency time.) If we do P-way merges, passing all the data from one drum to the other so that reading and writing are overlapped, the number of merge passes is $\lceil \log_P 60 \rceil$, so we may complete the job in two passes if $8 \le P \le 59$. The smallest such P reduces the amount of internal computing, so we choose $P = 8$; if 65 initial runs had been formed we would take $P = 9$. If 82 or more initial runs had been formed, we could take $P = 10$, but since there is room for only 18 input buffers and 2 output buffers there would be a possibility of hangup during the merge (see Algorithm 5.4.6F); it may be better in such a case to do two partial passes over a small portion of the data, reducing the number of initial runs to 81 or less.

Under our assumptions, both of the merging passes will take about 50 seconds, so the entire sort in this ideal situation will be completed in just $2\frac{1}{2}$ minutes (plus a few seconds for bookkeeping, initialization, etc.). This is six times faster than the best six-tape sort considered in Section 5.4.6; the reasons for this speedup are the improved external/internal transmission rate ($3\frac{1}{2}$ times faster), the higher order of merge (we can't do an eight-way tape merge unless we have nine or more tapes), and the fact that the output was left on disk (no final rewind, etc., was necessary). If the initial input and sorted output were required to be on MIXT tapes, with the drums used for merging only, the corresponding sorting time would have been about 8.2 minutes.

If only one drum were available instead of two, the input-output time

would take twice as long, since reading and writing must be done separately. (In fact, the input-output operations would take *three times* as long, since we would be overwriting the initial input data; in such a case it is prudent to follow each write by a "read-back check" operation, lest some of the input data be irretrievably lost, if the hardware does not provide automatic verification of written information.) But some of this excess time can be recovered because we can use partial pass methods which process some data records more often than others. The two-drum case requires all data to be processed an even number or an odd number of times, but the one-drum case can use more general merge patterns.

We observed in Section 5.4.4 that merge patterns can be represented by trees, and that the transmission time corresponding to a merge pattern is proportional to the external path length of its tree. Only certain trees ("T-lifo" or "strongly T-fifo") could be used as efficient tape merging patterns, because some runs get buried in the middle of a tape as the merging proceeds. But *on disks or drums, all trees define usable merge patterns* as long as the degrees of their internal nodes are not too large (i.e., compatible with the available internal memory size).

Therefore we can minimize transmission time by choosing a tree with minimum external path length, such as a complete P-ary tree where P is as large as possible. By Eq. 5.4.4–9, the external path length of such a tree is equal to

$$qS - \lfloor (P^q - S)/(P - 1) \rfloor, \qquad q = \lceil \log_P S \rceil, \tag{1}$$

if there are S external nodes (leaves).

It is particularly easy to design an algorithm which merges according to the complete P-ary tree pattern. See, for example, Fig. 91, which shows the case $P = 3$, $S = 6$. First we add "dummy runs," if necessary, to make $S \equiv 1$ (modulo $P - 1$); then we combine runs according to a "first-in-first-out" discipline, at every stage merging the P "oldest" runs at the front of the queue into a single run which is placed at the rear.

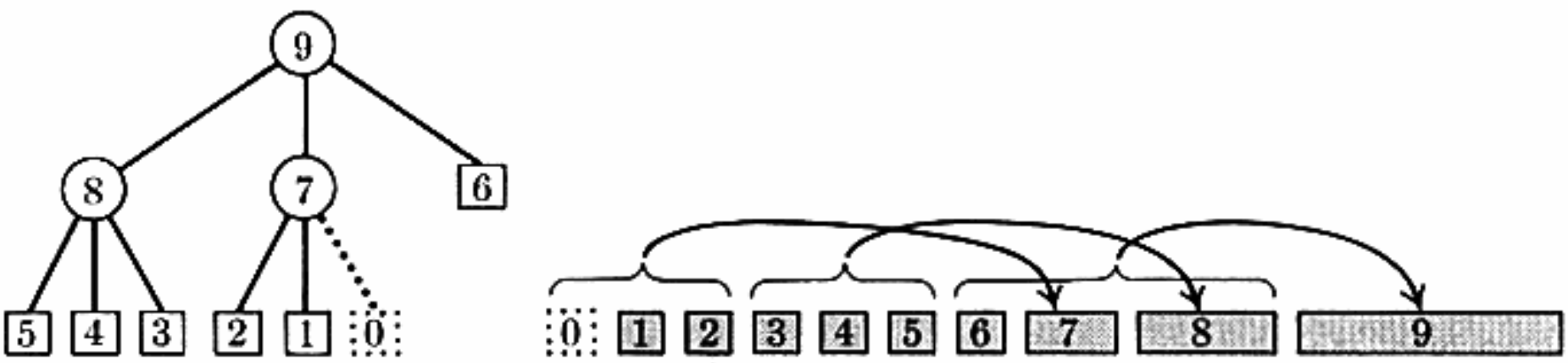


Fig. 91. Complete ternary tree with six leaves, and the corresponding merge pattern.

The complete P-ary tree gives an optimum pattern if all of the initial runs are the same length, but we can often do better if some runs are longer than others. An optimum pattern for this general situation can be constructed without difficulty by using Huffman's method (exercise 2.3.4.5–10), which may be stated in merging language as follows: "First add $(1 - S)$ mod $(P - 1)$

dummy runs of length 0. Then repeatedly merge together the P *shortest* existing runs until only one run is left." When all initial runs have the same length this method reduces to the fifo discipline described above.

In our 100,000-record example we can do nine-way merging, since 18 input buffers and two output buffers will fit in memory and Algorithm 5.4.6F will overlap all compute time. The complete 9-ary tree with 60 leaves corresponds to a merging pattern with $1\frac{29}{30}$ passes, if all initial runs have the same length. The total sorting time with one drum, using "read-back check" after every write, therefore comes to about 7.4 minutes. A higher value of P may reduce this running time slightly; but the situation is very unclear because "reading hangup" may occur, since the buffers might become too full or too empty.

The influence of seek time. The above discussion shows that it is relatively easy to construct "optimum" merging patterns for drums, because seek time and latency time can be essentially nonexistent. But when disks are used it often takes longer to seek out information than to read it, so the seek time has a considerable influence on the sorting strategy. Decreasing the order of merge, P, makes it possible to use larger buffers, so fewer seeks are required; this often compensates for the extra transmission time demanded by the smaller value of P.

Seek time depends on the distance traveled by the access arm, and we could try to arrange things so that this distance is minimized. For example, it may be wise to sort the records within cylinders first. However, large-scale merging requires a good deal of jumping around between cylinders (cf. exercise 2). Furthermore, the multiprogramming capability of modern operating systems means that a user rarely has much actual control over the position of disk access arms; fancy seek-minimization schemes usually work only on weekends! We are often justified, therefore, in assuming that each disk command involves a "random" seek.

Our goal is to discover a tree (i.e., a merge pattern) which achieves the best balance between seek time and transmission time; for this purpose we need some way to estimate the goodness of any particular tree with respect to a particular hardware configuration. Consider, for example, the tree in Fig. 92; we want to estimate how long it will take to carry out the corresponding merge, so that we can compare this tree to other trees.

In the following discussion we shall make some simple assumptions about disk merging, in order to illustrate some of the general ideas. Let us suppose that (1) it takes $72.5 + 0.005n$ milliseconds to read or write n characters; (2) 100,000 characters of internal memory are available for working storage; (3) an average of 0.004 milliseconds of computation time are required to transmit each character from input to output; (4) there is to be *no overlap* between reading, writing, or computing; and (5) the buffer size used on output need not be the same as the buffer size used to read the data on the following pass. An analysis of the sorting problem under these simple assumptions will give us some insights when we turn to more complicated situations.

If we do a P-way merge, we can divide the internal working storage into $P + 1$ buffer areas, P for input and one for output, with $B = 100000/(P + 1)$ characters per buffer. Suppose the files being merged contain a total of L characters; then we will do approximately L/B output operations and about the same number of input operations, so the total merging time under our assumptions will be approximately

$$2\left(72.5\,\frac{L}{B} + 0.005L\right) + 0.004L = (0.00145P + 0.01545)L \qquad (2)$$

milliseconds.

In other words, a P-way merge of L characters takes about $(\alpha P + \beta)L$ units of time, for some constants α and β depending on the seek time, latency time, compute time, and memory size. This formula leads to an interesting way to construct good merge patterns for disks. Consider Fig. 92, for example, and assume that all initial runs (represented by square "leaf" nodes) have length L_0. Then the merges at nodes 9 and 10 each take $(2\alpha + \beta)(2L_0)$ units of time, the merge at node 11 takes $(3\alpha + \beta)(4L_0)$, and the final merge at node 12 takes $(4\alpha + \beta)(8L_0)$. The total merging time therefore comes to $(52\alpha + 16\beta)L_0$ units. The coefficient "16" here is well-known to us, it is simply the external path length of the tree. The coefficient "52" of α is, however, a new concept, which we may call the *degree path length* of the tree; it is the sum, taken over all leaf nodes, of the internal-node degrees on the path from the leaf to the root. For example, in Fig. 92 the degree path length is $(2 + 4) + (2 + 4) + (3 + 4) + (2 + 3 + 4) + (2 + 3 + 4) + (3 + 4) + (4) + (4) = 52$.

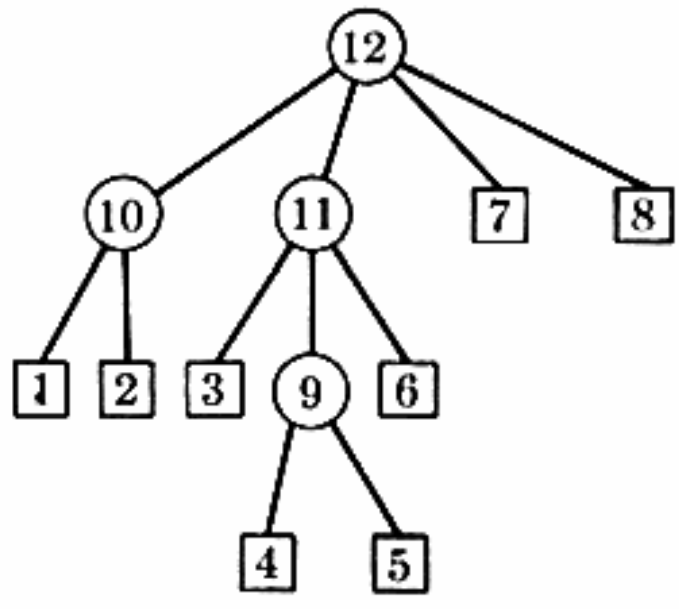


Fig. 92. A tree whose external path length is 16 and whose degree path length is 52.

If $\mathfrak{I}$ is any tree, let $D(\mathfrak{I})$, $E(\mathfrak{I})$ denote its degree path length and its external path length, respectively. Our analysis may be summarized as follows:

Theorem H. *If the time required to do a P-way merge on L characters has the form $(\alpha P + \beta)L$, and if there are S equal-length runs to be merged, the best merge pattern corresponds to a tree $\mathfrak{I}$ for which $\alpha D(\mathfrak{I}) + \beta E(\mathfrak{I})$ is a minimum, over all trees having S leaves.* ∎

(This theorem was implicitly contained in an unpublished paper which George U. Hubbard presented at the ACM National Conference in 1963.)

Let α and β be fixed constants; we shall say a tree is *optimal* if it has the minimum value of $\alpha D(\mathfrak{I}) + \beta E(\mathfrak{I})$ over all trees, $\mathfrak{I}$, with the same number of leaves. It is not difficult to see that *all subtrees of an optimal tree are optimal*, and therefore we can construct optimal trees with n leaves by piecing together optimal trees with $< n$ leaves.

Theorem K. *Let the sequence of numbers $A_m(n)$ be defined for $1 \le m \le n$ by the rules*

$$A_1(1) = 0; \tag{3}$$

$$A_m(n) = \min_{1 \le k \le n/m} \left(A_1(k) + A_{m-1}(n-k)\right), \quad \text{for} \quad 2 \le m \le n; \tag{4}$$

$$A_1(n) = \min_{2 \le m \le n} \left(\alpha m n + \beta n + A_m(n)\right), \quad \text{for} \quad n \ge 2. \tag{5}$$

Then $A_1(n)$ is the minimum value of $\alpha D(\mathfrak{I}) + \beta E(\mathfrak{I})$, over all trees $\mathfrak{I}$ with n leaves.

Proof. Equation (4) implies that $A_m(n)$ is the minimum value of $A_1(n_1) + \cdots + A_1(n_m)$ taken over all positive integers $n_1, \ldots, n_m$ such that $n_1 + \cdots + n_m = n$. The result now follows by induction on n. ∎

The recurrence relations (3), (4), (5) can also be used to construct the optimal trees themselves: Let $k_m(n)$ be a value for which the minimum occurs in the definition of $A_m(n)$. Then we can construct an optimal tree with n leaves by joining $m = k_1(n)$ subtrees at the root; the subtrees are optimal trees with $k_m(n)$, $k_{m-1}(n - k_m(n))$, $k_{m-2}(n - k_m(n) - k_{m-1}(n - k_m(n)))$, . . . leaves, respectively.

For example, Table 1 illustrates this construction when $\alpha = \beta = 1$. A compact specification of the corresponding optimal trees appears at the right of the table; the entry "4:9:9" when $n = 22$ means, for example, that an optimal tree $\mathfrak{I}_{22}$ with 22 leaves may be obtained by combining $\mathfrak{I}_4$, $\mathfrak{I}_9$, and $\mathfrak{I}_9$ (see Fig. 93). Optimal trees are not unique; for instance, 5:8:9 would be just as good as 4:9:9.

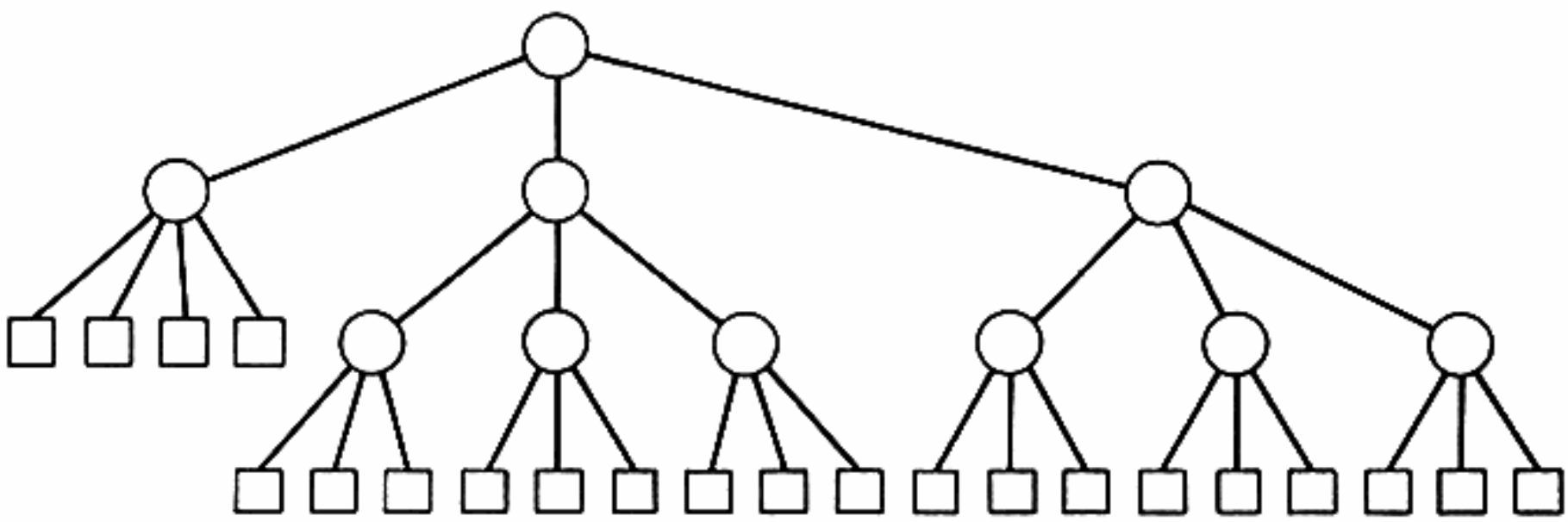


Fig. 93. An optimum way to merge 22 initial runs of equal length, when $\alpha = \beta$ in Theorem H. This pattern minimizes the seek time, under the assumptions leading to Eq. (2) in the text.

Table 1

OPTIMAL TREE CHARACTERISTICS $A_m(n)$, $k_m(n)$ WHEN $\alpha = \beta = 1$

	$m=1$	$m=2$	$m=3$	$m=4$	$m=5$	$m=6$	$m=7$	$m=8$	$m=9$	$m=10$	$m=11$	$m=12$	Tree specification	
$n=1$	0, 0												—	$n=1$
$n=2$	6, 2	0, 1											1:1	$n=2$
$n=3$	12, 3	6, 1	0, 1										1:1:1	$n=3$
$n=4$	20, 4	12, 1	6, 1	0, 1									1:1:1:1	$n=4$
$n=5$	30, 5	18, 2	12, 1	6, 1	0, 1								1:1:1:1:1	$n=5$
$n=6$	42, 2	24, 3	18, 1	12, 1	6, 1	0, 1							3:3	$n=6$
$n=7$	52, 3	32, 3	24, 1	18, 1	12, 1	6, 1	0, 1						1:3:3	$n=7$
$n=8$	62, 3	40, 4	30, 2	24, 1	18, 1	12, 1	6, 1	0, 1					2:3:3	$n=8$
$n=9$	72, 3	50, 4	36, 3	30, 1	24, 1	18, 1	12, 1	6, 1	0, 1				3:3:3	$n=9$
$n=10$	84, 3	60, 5	44, 3	36, 1	30, 1	24, 1	18, 1	12, 1	6, 1	0, 1			3:3:4	$n=10$
$n=11$	96, 3	72, 4	52, 3	42, 2	36, 1	30, 1	24, 1	18, 1	12, 1	6, 1	0, 1		3:4:4	$n=11$
$n=12$	108, 3	82, 4	60, 4	48, 3	42, 1	36, 1	30, 1	24, 1	18, 1	12, 1	6, 1	0, 1	4:4:4	$n=12$
$n=13$	121, 4	92, 4	70, 4	56, 3	48, 1	42, 1	36, 1	30, 1	24, 1	18, 1	12, 1	6, 1	3:3:3:4	$n=13$
$n=14$	134, 4	102, 5	80, 4	64, 3	54, 2	48, 1	42, 1	36, 1	30, 1	24, 1	18, 1	12, 1	3:3:4:4	$n=14$
$n=15$	147, 4	114, 5	90, 5	72, 3	60, 3	54, 1	48, 1	42, 1	36, 1	30, 1	24, 1	18, 1	3:4:4:4	$n=15$
$n=16$	160, 4	124, 7	102, 4	80, 4	68, 3	60, 1	54, 1	48, 1	42, 1	36, 1	30, 1	24, 1	4:4:4:4	$n=16$
$n=17$	175, 4	134, 8	112, 4	90, 4	76, 3	66, 2	60, 1	54, 1	48, 1	42, 1	36, 1	30, 1	4:4:4:5	$n=17$
$n=18$	190, 4	144, 9	122, 4	100, 4	84, 3	72, 3	66, 1	60, 1	54, 1	48, 1	42, 1	36, 1	4:4:5:5	$n=18$
$n=19$	205, 4	156, 9	132, 5	110, 4	92, 3	80, 3	72, 1	66, 1	60, 1	54, 1	48, 1	42, 1	4:5:5:5	$n=19$
$n=20$	220, 4	168, 9	144, 4	120, 5	100, 4	88, 3	78, 2	72, 1	66, 1	60, 1	54, 1	48, 1	5:5:5:5	$n=20$
$n=21$	236, 5	180, 9	154, 4	132, 4	110, 4	96, 3	84, 3	78, 1	72, 1	66, 1	60, 1	54, 1	4:4:4:4:5	$n=21$
$n=22$	252, 3	192, 10	164, 4	142, 4	120, 4	104, 3	92, 3	84, 1	78, 1	72, 1	66, 1	60, 1	4:9:9	$n=22$
$n=23$	266, 3	204, 11	174, 5	152, 4	130, 4	112, 3	100, 3	90, 2	84, 1	78, 1	72, 1	66, 1	5:9:9	$n=23$
$n=24$	282, 3	216, 12	186, 5	162, 5	140, 4	120, 4	108, 3	96, 3	90, 1	84, 1	78, 1	72, 1	5:9:10	$n=24$
$n=25$	296, 3	229, 12	196, 7	174, 4	150, 5	130, 4	116, 3	104, 3	96, 1	90, 1	84, 1	78, 1	7:9:9	$n=25$

Our derivation of (2) shows that the relation $\alpha \leq \beta$ will hold whenever $P + 1$ equal buffer areas are used. The limiting case $\alpha = \beta$, shown in Table 1 and Fig. 93, occurs when the seek time itself is to be minimized without regard to transmission time.

Returning to our original application, we still haven't considered how to get the initial runs in the first place; without read/write/compute overlap, replacement selection loses some of its advantages. Perhaps we should fill the entire internal memory, sort it, and output the results; such input and output operations can each be done with one seek. Or perhaps we are better off using, say, 20 percent of the memory as a combination input/output buffer, and doing replacement selection. This requires five times as many seeks (an extra 60 seconds or so!), but it reduces the number of initial runs from 100 to 64; the reduction would be more dramatic if the input file were pretty much in order already.

If we decide not to use replacement selection, the optimum tree for $S = 100$, $\alpha = 0.00145$, $\beta = 0.01545$ [cf. (2)] turns out to be rather prosaic: It is simply a 10-way merge, completed in two passes over the data. Allowing 30 seconds for internal sorting (100 quicksorts, say), the initial distribution pass takes about $2\frac{1}{2}$ minutes, and the merge passes each take almost 5 minutes, for a total of 12.4 minutes. If we decide to use replacement selection, the optimal tree for $S = 64$ turns out to be equally uninteresting (two 8-way merge passes); the initial distribution pass takes about $3\frac{1}{2}$ minutes, the merge passes each take about $4\frac{1}{2}$ minutes, and the estimated total time comes to 12.6 minutes. Remember that both of these methods give up virtually all read/write/compute overlap in order to have larger buffers (reducing seek time). None of these estimated

times includes the time which might be necessary for "read-back check" operations.

In practice the final merge pass tends to be quite different from the others; for example, the output is often expanded and/or written onto tape. In such cases the tree pattern should be chosen using a different optimality criterion at the root.

***Another way to allocate buffers.** David E. Ferguson [*CACM* **14** (1971), 476–478] has pointed out that seek time can be reduced if we don't make all buffers the same size. The same idea occurred at about the same time to several other people [S. J. Waters, *Comp. J.* **14** (1971), 109–112; Ewing S. Walker, Software Age **4** (August-September, 1970), 16–17].

Suppose we are doing a four-way merge on runs of equal length L_0, with M characters of memory. If we divide the memory into equal buffers of size $B = M/5$, we need about L_0/B seeks on each input file and $4L_0/B$ seeks for the output, totalling $8L_0/B = 40L_0/M$ seeks. But if we use four input buffers of size $M/6$ and one output buffer of size $M/3$, we need only about $4 \times (6L_0/M) + 4 \times (3L_0/M) = 36L_0/M$ seeks! The transmission time is the same in both cases, so we haven't lost anything by the change.

In general, suppose that we want to merge sorted files of lengths $L_1, \ldots, L_P$ into a sorted file of length $L_{P+1} = L_1 + \cdots + L_P$, and assume that a buffer of size B_k is being used for the kth file. Thus

$$B_1 + \cdots + B_P + B_{P+1} = M, \tag{6}$$

where M is the total size of available internal memory. The number of seeks will be approximately

$$\frac{L_1}{B_1} + \cdots + \frac{L_P}{B_P} + \frac{L_{P+1}}{B_{P+1}}. \tag{7}$$

Let's try to minimize this quantity, subject to condition (6), assuming for convenience that the B_k's don't have to be integers. If we increase B_j by δ and decrease B_k by the same small amount, the number of seeks changes by

$$\frac{L_j}{B_j + \delta} - \frac{L_j}{B_j} + \frac{L_k}{B_k - \delta} - \frac{L_k}{B_k} = \left(\frac{L_k}{B_k(B_k - \delta)} - \frac{L_j}{B_j(B_j + \delta)}\right)\delta,$$

so the allocation can be improved if $L_j/B_j^2 \neq L_k/B_k^2$. Therefore we get the minimum number of seeks only if

$$\frac{L_1}{B_2^2} = \cdots = \frac{L_P}{B_P^2} = \frac{L_{P+1}}{B_{P+1}^2}. \tag{8}$$

Since a minimum does exist it must occur when

$$B_k = \sqrt{L_k}\, M/(\sqrt{L_1} + \cdots + \sqrt{L_{P+1}}), \qquad 1 \le k \le P + 1, \tag{9}$$

for these are the only values of $B_1, \ldots, B_{P+1}$ which simultaneously satisfy (6) and (8). Putting these values into (7) gives a fairly simple formula for the total number of seeks,

$$(\sqrt{L_1} + \cdots + \sqrt{L_{P+1}})^2/M, \tag{10}$$

which may be compared with the number $(P+1)(L_1 + \cdots + L_{P+1})/M$ obtained if all buffers are equal in length. By exercise 1.2.3–31, the improvement is $\sum_{1 \le j < k \le P+1} (\sqrt{L_j} - \sqrt{L_k})^2/M$. Unfortunately formula (10) does not lend itself to an easy determination of optimum merge patterns as in Theorem K (cf. exercise 14).

Further considerations. If two disk units are being used, one for reading and one for writing, we can overlap some of the write time at the expense of an extra output buffer. If the "forecasting" technique is used (i.e., looking at the last keys in each input buffer in order to determine which buffer will empty first), we can overlap some of the read time. Unfortunately a simple synchronization procedure like that of Algorithm 5.4.6F does not provide continuous input/output, since the variable behavior of seek time means that input and output operations won't finish at the same time. The exact amount of overlap is hard to estimate, and it may be best to simulate the merging process in order to find constants α and β which approximate the running time as in Theorem H.

If we use "floating" input buffers (not dedicated to a particular file), they must all be the same size, so the above buffer allocation technique cannot be used except that the output buffers may be chosen larger than those for input. If $2P$ input buffers are used instead of P, we overlap a good deal of the input time yet double the seek time, so we may be gaining nothing; perhaps $P+1$ or $P+2$ input buffers would be best.

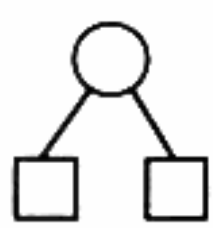
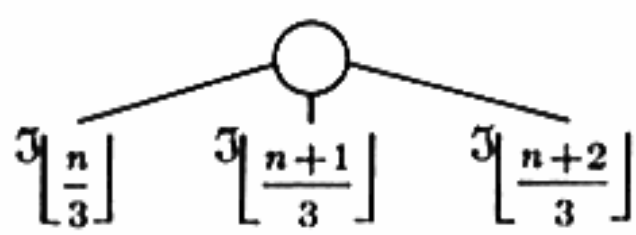
It should be clear to the reader by now that no clear cut strategy for optimum disk sorting has been worked out; the number of available options greatly exceeds the number of strategies that have been theoretically analyzed. Although the theory developed in this section provides some useful insights, a good deal of experimentation still needs to be done.

***A closer look at optimal trees.** It is interesting to examine the extreme case $\beta = 0$ in Theorems H and K, even though practical situations usually lead to parameters with $0 \le \alpha \le \beta$. What tree with n leaves has the smallest possible degree path length? Curiously it turns out that three-way merging is best in this case.

Theorem L. *The degree path length of a tree with n leaves is never less than*

$$f(n) = \begin{cases} 3qn + 2(n - 3^q), & \text{if } 2 \cdot 3^{q-1} \le n \le 3^q; \\ 3qn + 4(n - 3^q), & \text{if } 3^q \le n \le 2 \cdot 3^q. \end{cases} \tag{11}$$

Ternary trees $\mathfrak{I}_n$ defined by the rules

$$\mathfrak{I}_1 = \square, \qquad \mathfrak{I}_2 = \text{  }, \qquad \mathfrak{I}_n = \text{  } \tag{12}$$

have the minimum degree path length.

Proof. It is important to observe that $f(n)$ is a *convex function*, i.e. that

$$f(n+1) - f(n) \geq f(n) - f(n-1) \qquad \text{for all} \qquad n \geq 2. \tag{13}$$

The relevance of this property is due to the following lemma.

Lemma C. *A function $g(n)$ defined on the positive integers satisfies*

$$\min_{1\leq k\leq n} \big(g(k) + g(n-k)\big) = g(\lfloor n/2 \rfloor) + g(\lceil n/2 \rceil), \qquad n \geq 2, \tag{14}$$

if and only if it is convex.

Proof. If $g(n+1) - g(n) < g(n) - g(n-1)$ for some $n \geq 2$, we have $g(n+1) + g(n-1) < g(n) + g(n)$, contradicting (14). Conversely, if (13) holds for g, and $1 \leq k < n-k$, we have $g(k+1) + g(n-k-1) \leq g(k) + g(n-k)$ by convexity. ∎

The latter part of Lemma C's proof can be extended for any $m \geq 2$ to show that

$$\min_{\substack{n_1+\cdots+n_m=n \\ n_1,\cdots,n_m\geq 1}} \big(g(n_1) + \cdots + g(n_m)\big)$$
$$= g(\lfloor n/m \rfloor) + g(\lfloor (n+1)/m \rfloor) + \cdots + g(\lfloor (n+m-1)/m \rfloor), \tag{15}$$

whenever g is convex. Let

$$f_m(n) = f(\lfloor n/m \rfloor) + f(\lfloor (n+1)/m \rfloor) + \cdots + f(\lfloor (n+m-1)/m \rfloor); \tag{16}$$

the proof of Theorem L is completed by proving that $f_3(n) + 3n = f(n)$ and $f_m(n) + mn \geq f(n)$ for all $m \geq 2$. (See exercise 11.) ∎

It would be very nice if optimal trees could always be characterized neatly as in Theorem L. But the results we have seen for $\alpha = \beta$ in Table 1 show that the function $A_1(n)$ is not always convex. In fact, Table 1 is sufficient to disprove most simple conjectures about optimal trees! We can, however, salvage part of Theorem L in the general case; M. Schlumberger and J. Vuillemin have shown that *large* orders of merge can always be avoided:

Theorem M. *Given α and β as in Theorem H, there exists an optimal tree in which the degree of every node is at most*

$$d(\alpha, \beta) = \left\lceil \min_{k\geq 1} \left(k + \left(1 + \frac{1}{k}\right)\left(1 + \frac{\beta}{\alpha}\right)\right)\right\rceil. \tag{17}$$

Proof. Let $n_1, \ldots, n_m$ be positive integers such that $n_1 + \cdots + n_m = n$, $A(n_1) + \cdots + A(n_m) = A_m(n)$, and $n_1 \leq \cdots \leq n_m$, and assume that $m \geq d(\alpha, \beta) + 1$. Let k be the value which minimizes (17); we shall show that

$$\alpha n(m-k) + \beta n + A_{m-k}(n) \leq \alpha nm + \beta n + A_m(n), \tag{18}$$

hence the minimum value in (4) is always achieved for some $m \le d(\alpha, \beta)$.

By definition, since $m \ge k+2$, we must have

$$\begin{aligned} A_{m-k}(n) &\le A_1(n_1 + \cdots + n_{k+1}) + A_1(n_{k+2}) + \cdots + A_1(n_m) \\ &\le \alpha(n_1 + \cdots + n_{k+1})(k+1) + \beta(n_1 + \cdots + n_{k+1}) \\ &\quad + A_1(n_1) + \cdots + A_1(n_m) \\ &= \big(\alpha(k+1) + \beta\big)(n_1 + \cdots + n_{k+1}) + A_m(n) \\ &\le \big(\alpha(k+1) + \beta\big)(k+1)n/m + A_m(n), \end{aligned}$$

and (18) now follows easily. (Careful inspection of this proof shows that (17) is "best possible," in the sense that some optimal trees must have nodes of degree $d(\alpha, \beta)$; see exercise 13.) ∎

The construction in Theorem K requires $O(N^2)$ memory cells and $O(N^2 \log N)$ steps to evaluate $A_m(n)$ for $1 \le m \le n \le N$; Theorem M shows that only $O(N)$ cells and $O(N^2)$ steps are needed. Schlumberger and Vuillemin have discovered several more very interesting properties of optimal trees ["Optimal disk merge patterns," to appear].

The use of chaining. M. A. Goetz [*CACM* **6** (1963), 245–248] has suggested an interesting way to avoid seek time on output, by linking individual tracks together. His idea requires a fairly fancy set of disk storage management routines, but it applies to many problems besides sorting, and it may therefore be a very worthwhile technique for general-purpose use.

The concept is simple: Instead of allocating tracks sequentially within cylinders of the disk, we link them together and maintain lists of available space, one for each cylinder. When it is time to output a track of information, we write it on the current cylinder (wherever the access arm happens to be), unless that cylinder is full. In this way the seek time usually disappears.

The catch is that we can't store a link-to-next-track within the track itself, since the necessary information isn't known at the right time. (We could store a link-to-previous-track and read the file backward on the next pass, if that were suitable.) A separate table of link addresses for the tracks of each file can be maintained, perhaps partly on disk itself. The available space lists can be compactly represented by using bit tables, with 1000 bits specifying the availability or unavailability of 1000 tracks.

Will keysorting help? When records are long and keys are short, it is very tempting to create a new file consisting simply of the keys together with a serial number specifying their original file location. After sorting this key file, we can replace the keys by the successive numbers 1, 2, . . .; the new file can then be sorted by original file location and we will have a convenient specification of how to unshuffle the records for the final rearrangement. Schematically, the process has the following form:

a) Original file	$(K_1, I_1)(K_2, I_2) \ldots (K_N, I_N)$	long
b) Key file	$(K_1, 1)(K_2, 2) \ldots (K_N, N)$	short
c) Sorted (b)	$(K_{p_1}, p_1)(K_{p_2}, p_2) \ldots (K_{p_N}, p_N)$	short
d) Edited (c)	$(1, p_1)(2, p_2) \ldots (N, p_N)$	short
e) Sorted (d)	$(q_1, 1)(q_2, 2) \ldots (q_N, N)$	short
f) Edited (a)	$(q_1, I_1)(q_2, I_2) \ldots (q_N, I_N)$	long

Here $p_j = k$ if and only if $q_k = j$; the two sorting processes in (c), (e) are comparatively fast (perhaps even internal sorts), since the records aren't very long. In stage (f) we have reduced the problem to sorting a file whose keys are simply the numbers $\{1, 2, \ldots, N\}$; each record now specifies exactly where it is to be moved.

The external rearrangement problem which remains after stage (f) seems trivial, at first glance; but in fact it is rather difficult, and no really good algorithms (significantly better than sorting) have yet been found. We could obviously do the rearrangement in N steps, moving one record at a time; for large enough N this is better than the $N \log N$ of a sorting method. But N is never that large, and N seeks are unthinkable!

A radix sorting method can be used efficiently on the edited records, since their keys have a perfectly uniform distribution. On many high-speed computers with high-speed peripheral devices, the processing time for an eight-way distribution is much faster than the processing time for an eight-way merge; hence a distribution sort may be the best procedure. (See Section 5.4.7, and see also exercise 17.)

But it seems wasteful to do a key sort only to follow it by a full sort! One reason the external rearrangement problem is unexpectedly difficult has been discovered by R. W. Floyd, who found a nontrivial lower bound on the number of seeks required to rearrange records on a disk file.

It is convenient to describe Floyd's result in terms of the elevator problem of Section 5.4.8; this time we want to find an elevator schedule which minimizes the number of *stops*, instead of minimizing the distance traveled. (Minimizing the number of stops is not precisely equivalent to finding the minimum-seek rearrangement algorithm, since a stop combines input to the elevator with output from the elevator; but the stop-minimization criterion is close enough to indicate the basic ideas.)

We shall make use of the "discrete entropy" function

$$F(n) = \sum_{1<k\le n} (\lceil \log_2 k \rceil + 1) = B(n) + n - 1 = n\lceil \log_2 n \rceil - 2^{\lceil \log_2 n \rceil} + n, \tag{19}$$

where $B(n)$ is the "binary insertion" function, Eq. 5.3.1–3. By Eq. 5.3.1–33, $F(n)$ is the minimum external path length of a binary tree with n leaves, and

$$n \log_2 n \le F(n) \le n \log_2 n + 0.0861n. \tag{20}$$

Since $F(n)$ is convex and satisfies $F(n) = n + F(\lfloor n/2 \rfloor) + F(\lceil n/2 \rceil)$, we know by Lemma C above that

$$F(n) \le F(k) + F(n - k) + n, \qquad \text{for} \qquad 0 \le k \le n. \tag{21}$$

This relation is also evident from the external path length characterization of F; it is the crucial fact used in what follows.

As in Section 5.4.8 we shall assume that the elevator holds b people, each floor holds c people, and there are n floors. Let s_{ij} be the number of people currently on floor i whose destination is floor j. The *togetherness rating* of any configuration of people in the building is defined to be the sum $\sum_{1 \le i,j \le n} F(s_{ij})$.

For example, assume that $b = c = n = 6$ and that the 36 people are initially scattered among the floors as follows:

```
␣␣␣␣␣␣
123456  123456  123456  123456  123456  123456
```
(22)

The elevator is empty, sitting on floor 1; "␣" denotes a vacant position. Each floor contains one person with each possible destination, so all s_{ij} are 1 and the togetherness rating is zero. If the elevator now transports six people to floor 2, we have the configuration

```
        123456
␣␣␣␣␣␣  123456  123456  123456  123456  123456
```
(23)

and the togetherness rating becomes $6F(0) + 24F(1) + 6F(2) = 12$. Suppose the elevator carries 1, 1, 2, 3, 3, 4 to floor 3:

```
              112334
␣␣␣␣␣␣  245566  123456  123456  123456  123456
```
(24)

The togetherness rating has jumped to $4F(2) + 2F(3) = 18$. When everyone has finally been transported to his or her destination, the togetherness rating will be $6F(6) = 96$.

Floyd observed that the togetherness rating can never increase by more than $b + c$ at each stop, since a set of s equal-destination people joining with a similar set of size s' improves the rating by $F(s + s') - F(s) - F(s') \le s + s'$. Therefore we have the following result.

Theorem F. *Let t be the togetherness rating of an initial configuration of people, in terms of the definitions above. The elevator must make at least*

$$\lceil (nF(c) - t)/(b + c) \rceil$$

stops in order to bring them all to their destinations. ▮

Translating this result into disk terminology, let there be N records, with B per block, and suppose the internal memory can hold M records at a time.

Every disk read brings one block into memory, every disk write stores one block, and s_{ij} is the number of records in block i that belong in block j. If $N \geq B^2$, there are initial configurations in which all the s_{ij} are ≤ 1, so at least $(N/B)F(B)/M \approx (N \log_2 B)/M$ block-reading operations are necessary to rearrange the records. (The factor $\log_2 B$ makes this lower bound nontrivial when B is large, but when $B = 1$ the lower bound can obviously be improved.)

EXERCISES

1. [*M22*] The text explains a method by which the average latency time required to read a fraction x of a track is reduced from $\frac{1}{2}$ to $\frac{1}{2}(1 - x^2)$ revolutions. This is the minimum possible value, when there is one access arm. What is the corresponding minimum average latency time if there are *two* access arms, 180° apart, assuming that only one arm can transmit data at any one time?

2. [*M30*] (A. G. Konheim.) The purpose of this problem is to investigate how far the access arm of a disk must move while merging files that are allocated "orthogonally" to the cylinders. Suppose there are P files, each containing L blocks of records, and assume that the first block of each file appears on cylinder 1, the second on cylinder 2, etc. The relative order of the last keys in each block governs the access arm motion during the merge, hence we may represent the situation in the following mathematically tractable way: Consider a set of PL ordered pairs

$$\begin{array}{cccc} (a_{11}, 1) & (a_{21}, 1) & \ldots & (a_{p1}, 1) \\ (a_{12}, 2) & (a_{22}, 2) & \ldots & (a_{p2}, 2) \\ \vdots & \vdots & & \vdots \\ (a_{1L}, L) & (a_{2L}, L) & \ldots & (a_{pL}, L) \end{array}$$

where the set $\{a_{ij} \mid 1 \le i \le P, 1 \le j \le L\}$ consists of the numbers $\{1, 2, \ldots, PL\}$ in some order, and where $a_{ij} < a_{i,j+1}$ for $1 \le j < L$. (Rows represent cylinders, columns represent input files.) Sort the pairs on their first components and let the resulting sequence be $(1, j_1)\ (2, j_2) \ldots (PL, j_{PL})$. Show that, if each of the $(PL)!/L!^P$ choices of the a_{ij} is equally likely, the average value of

$$|j_2 - j_1| + |j_3 - j_2| + \cdots + |j_{PL} - j_{PL-1}|$$

is

$$(L - 1)\left(1 + (P - 1)2^{2L-2} \Big/ \binom{2L}{L}\right).$$

[*Hint:* See exercise 5.2.1–14.] Note that as $L \to \infty$ this value is asymptotically $\frac{1}{4}(P - 1)L\sqrt{\pi L}$.

3. [*M15*] Suppose the internal memory is limited so that 10-way merging is not feasible. How can recurrence relations (3), (4), (5) be modified so that $A_1(n)$ is the minimum value of $\alpha D(\mathfrak{T}) + \beta E(\mathfrak{T})$, over all n-leaved trees $\mathfrak{T}$ having no internal nodes of degree greater than 9?

▸ **4.** [*M21*] (a) Derive a formula corresponding to (2), for the running time of an L-character P-way merge, when a modified form of the square root buffer allocation is used: All P of the input buffers should have equal length, but the output buffer size should be chosen so as to minimize seek time.

(b) Show that the construction in Theorem K can be modified in order to obtain a merge pattern which is optimal according to your formula from part (a).

5. [*M20*] When two disks are being used, so that reading on one is overlapped with writing on the other, we cannot use merge patterns like that of Fig. 93 since some leaves are at even levels and some are at odd levels. Show how to modify the construction of Theorem K in order to produce trees that are optimal subject to the constraint that all leaves appear on even levels or all on odd levels.

▸ **6.** [*22*] Find a tree which is optimum in the sense of exercise 5, when $n = 23$ and $\alpha = \beta = 1$. (You may wish to use a computer.)

▸ **7.** [*M24*] When the initial runs are not all the same length, the best merge pattern (in the sense of Theorem H) minimizes $\alpha D(\mathfrak{I}) + \beta E(\mathfrak{I})$, where $D(\mathfrak{I})$ and $E(\mathfrak{I})$ now represent *weighted* path lengths: Weights $w_1, \ldots, w_n$ (corresponding to the lengths of the initial runs) are attached to each leaf of the tree, and the degree sums and path lengths are multiplied by the appropriate weights. For example, if $\mathfrak{I}$ is the tree of Fig. 92, we would have $D(\mathfrak{I}) = 6w_1 + 6w_2 + 7w_3 + 9w_4 + 9w_5 + 7w_3 + 4w_7 + 4w_8$, $E(\mathfrak{I}) = 2w_1 + 2w_2 + 2w_3 + 3w_4 + 3w_5 + 2w_6 + w_7 + w_8$.

Prove that there is always an optimal pattern in which the *shortest* k runs are merged first, for some k.

8. [*49*] Is there an algorithm which finds optimal trees for given α, β and weights $w_1, \ldots, w_n$, in the sense of exercise 7, taking only $O(n^c)$ steps for some c?

9. [*M49*] (Schlumberger and Vuillemin.) Do there exist n-leaf optimal trees in the sense of Theorem H, such that all leaves appear on at most two adjacent levels, for all n, α, and β? When α and β are fixed, is there a number m such that $A_1(n) = \alpha mn + \beta n + A_m(n)$ for all sufficiently large n? (Both of these conjectures have been verified when $\alpha/\beta \geq 4$.)

10. [*HM49*] Given α and β, what is the asymptotic value of $A_1(n)$ as $n \to \infty$?

11. [*M29*] In the notation of (11) and (16), prove that $f_m(n) + mn \geq f(n)$ for all $m, n \geq 2$, and determine all m and n for which equality holds.

12. [*25*] Prove that, for all $n > 0$, there is a tree with n leaves and minimum degree path length (11), with all leaves at the same level.

13. [*M24*] Show that for $2 \leq n \leq d(\alpha, \beta)$, the unique best merge pattern in the sense of Theorem H is an n-way merge. (Cf. (17).)

14. [*40*] Using the square root method of buffer allocation, the seek time for the merge pattern in Fig. 92 would be proportional to $(\sqrt{2}+\sqrt{4}+\sqrt{1}+\sqrt{1}+\sqrt{8})^2 + (\sqrt{1}+\sqrt{1}+\sqrt{2})^2 + (\sqrt{1}+\sqrt{2}+\sqrt{1}+\sqrt{4})^2 + (\sqrt{1}+\sqrt{1}+\sqrt{2})^2$; this is the sum, over each internal node, of $(\sqrt{n_1}+\cdots+\sqrt{n_m}+\sqrt{n_1+\cdots+n_m})^2$, where that node's respective subtrees have $(n_1, \ldots, n_m)$ leaves. Write a computer program which generates minimum-seek time trees having 1, 2, 3, . . . leaves, using this formula to estimate the seek time.

15. [*M22*] Show that Theorem F can be improved slightly if the elevator is initially

empty and if $nF(c) \neq t$: At least $\lceil (nF(c) + b - t)/(b + c) \rceil$ stops are necessary in such a case.

16. [*23*] (R. W. Floyd.) Find an elevator schedule which transports all the people of (22) to their destinations in 12 stops or less. (Configuration (23) shows the situation after one, not two, stops.)

17. [*25*] (B. T. Bennett and A. C. McKellar, 1971.) Consider the following approach to keysorting, illustrated on an example file with 10 keys:

a) Original file: (50, I_0)(08, I_1)(51, I_2)(06, I_3)(90, I_4)(17, I_5)(89, I_6)(27, I_7)(65, I_8)(42, I_9)
b) Key file: (50, 0)(08, 1)(51, 2)(06, 3)(90, 4)(17, 5)(89, 6)(27, 7)(65, 8)(42, 9)
c) Sorted (b): (06, 3)(08, 1)(17, 5)(27, 7)(42, 9)(50, 0)(51, 2)(65, 8)(89, 6)(90, 4)
d) Bin assignments: (3, 3)(3, 9)(2, 1)(2, 5)(2, 7)(2, 8)(1, 0)(1, 2)(1, 4)(1, 6)
e) Sorted (d): (1, 0)(2, 1)(1, 2)(3, 3)(1, 4)(2, 5)(1, 6)(2, 7)(2, 8)(3, 9)
f) (a) distributed into bins using (e):
   Bin 1: (50, I_0)(51, I_2)(90, I_4)(89, I_6)
   Bin 2: (08, I_1)(17, I_5)(27, I_7)(65, I_8)
   Bin 3: (06, I_3)(42, I_9)
g) The result of replacement selection, reading first bin 3, then bin 2, then bin 1: (06, I_3)(08, I_1)(17, I_5)(27, I_7)(42, I_9)(50, I_0)(51, I_2)(65, I_8)(89, I_6)(90, I_4)

The assignment of bin numbers in step (d) is made by doing *replacement selection* on (c), *from right to left*, in *decreasing* order of the second component. The bin number is the run number. The above example uses replacement selection with only two elements in the selection tree; the same size tree should be used for replacement selection in both (d) and (g). Note that the bin contents are not necessarily in sorted order!

Prove that this method will sort, i.e., that the replacement selection in (g) will produce only one run. (This technique reduces the number of bins needed in a conventional keysort by distribution, especially if the input is largely in order already.)

▸ **18.** [*25*] Some hardware/software systems provide programmers with a *virtual memory:* Programs can be written as if there were a very large internal memory, able to contain all of the data. This memory is divided into *pages*, only a few of which are in the actual internal memory at any one time; the others are on disks or drums. The programmer need not concern himself with such details, since the system takes care of everything; new pages are automatically brought into memory when needed.

It would seem that the advent of virtual memory technology makes external sorting methods obsolete, since the job can simply be done using the techniques developed for internal sorting. Discuss this situation; in what ways might a hand-tailored external sorting method be better than the application of a general-purpose paging technique to an internal sorting method?

5.5. SUMMARY, HISTORY, AND BIBLIOGRAPHY

Now that we have nearly reached the end of this enormously long chapter, we had better "sort out" the most important facts we have studied.

An algorithm for sorting is a procedure which rearranges a file of records so that the keys are in ascending order. This orderly arrangement is useful because it brings equal-key records together, it allows efficient processing of multiple files sorted on the same key, it leads to efficient retrieval algorithms, and it makes computer output look more authentic.

Internal sorting is used when all of the records fit in the computer's high speed internal memory. We have studied more than two dozen algorithms for internal sorting, in various degrees of detail; and perhaps we would be happier if we didn't know so many different approaches to the problem! It was fun to learn all the techniques, but now we must face the horrible prospect of actually deciding which method ought to be used in a given situation.

It would be nice if only one or two of the sorting methods would dominate all of the others, regardless of the application or the computer being used. But in fact, each method has its own peculiar virtues. For example, the bubble sort (Algorithm 5.2.2B) has no apparent redeeming features, since there is always a better way to do what it does; but even this technique, suitably generalized, turns out to be useful for two-tape sorting (cf. Section 5.4.8). Thus we find that nearly all of the algorithms deserve to be remembered, since there are some applications in which they turn out to be best.

The following brief survey gives the highlights of the most significant algorithms we have encountered for internal sorting. As usual, N stands for the number of records in the given file.

1. *Distribution counting*, Algorithm 5.2D, is very useful when the keys have a small range. It is stable (doesn't affect the order of records with equal keys), but requires memory space for counters and for $2N$ records. A modification which saves N of these record spaces at the cost of stability appears in exercise 5.2–13.

2. *Straight insertion*, Algorithm 5.2.1S, is the simplest method to program, requires no extra space, and is quite efficient for small N (say $N \leq 25$). For large N it is unbearably slow unless the input is nearly in order.

3. *Diminishing increment sort*, Algorithm 5.2.1D (Shell's method), is also quite easy to program, and uses minimum memory space; and it is reasonably efficient for moderately large N (say $N \leq 1000$).

4. *List insertion*, Algorithm 5.2.1L, uses the same basic idea as straight insertion, so it is suitable only for small N. Like the other list sorting methods described below, it saves the cost of moving long records by manipulating links; this is particularly advantageous when the records have variable length or are part of other data structures.

5. *Address calculation* techniques are efficient when the keys have a known

(usually uniform) distribution; the principal variants of this approach are *multiple list insertion* (Algorithm 5.2.1M), and MacLaren's combined radix-insertion method (discussed at the close of Section 5.2.5). The latter can be done with only $O(\sqrt{N})$ cells of additional memory.

6. *Merge exchange*, Algorithm 5.2.2M (Batcher's method) and its cousin the *bitonic sort* (exercise 5.3.4–10) are useful when a large number of comparisons can be made simultaneously.

7. *Partition exchange*, Algorithm 5.2.2Q (Hoare's method, commonly known as *quicksort*) is probably the most useful general-purpose technique for internal sorting, because it requires very little memory space and its average running time on most computers beats that of its competitors. It can run *very* slowly in its worst case, however, so a careful choice of the partitioning elements should be made whenever nonrandom data are likely. Choosing the median of three elements, as suggested in exercise 5.2.2–55, makes the worst-case behavior extremely unlikely and also gives a slight improvement to the average running time.

8. *Straight selection*, Algorithm 5.2.3S, is a simple method especially suitable when special hardware is available to find the smallest element of a list at high speed.

9. *Heapsort*, Algorithm 5.2.3H, requires minimum memory and is guaranteed to run pretty fast; its average time and its maximum time are both roughly twice the average running time of quicksort.

10. *List merging*, Algorithm 5.2.4L, is a list sort which, like heapsort, is guaranteed to be rather fast even in its worst case; moreover, it is stable with respect to equal keys.

11. *Radix sorting*, using Algorithm 5.2.5R, is a list sort especially appropriate for keys which are either rather short or which have an unusual lexicographic collating sequence. Alternatively, the distribution counting method (point 1 above) could be used instead of links; such a procedure requires $2N$ record spaces, plus a table of counters, but the simple form of its inner loop makes it especially good for ultra-fast, "number-crunching" computers which have look-ahead control. *Caution:* Radix sorting should not be used for small N!

12. *Merge insertion*, see Section 5.3.1, is especially suitable for very small N, in a "straight-line-coded" routine; for example, it would be the appropriate method in an application which requires the sorting of numerous five- or six-record groups.

13. Hybrid methods, combining one or more of the above techniques, are also possible. For example, merge insertion could be used for sorting short subfiles that arise in quicksort.

14. Finally, an unnamed method appearing in the answer to exercise 5.2.1–3 seems to require the shortest possible sorting program. But its average running time, proportional to N^3, makes it the slowest sorting routine in this book!

Table 1

A COMPARISON OF INTERNAL SORTING METHODS USING THE MIX COMPUTER

Method	Reference	Stable?	Length of MIX code	Space	Running time: Average	Running time: Maximum	Running time: $N = 16$	Running time: $N = 1000$	Notes
Comparison counting	Ex. 5.2–5	Yes	22	$N(1+\epsilon)$	$4N^2+10N$	$5.5N^2$	1065	3992427	c
Distribution counting	Ex. 5.2–9	Yes	26	$2N+1000\epsilon$	$22N+10000$	$22N$	10352	32000	a
Straight insertion	Ex. 5.2.1–33	Yes	12	N	$2N^2+9N$	$4N^2$	494	1985566	
Diminishing increments	Prog. 5.2.1D	No	21	N	$15N^{1.25}$	$cN^{1.5}$	529	131002	d, h
List insertion	Ex. 5.2.1–33	Yes	19	$N(1+\epsilon)$	$1.25N^2+13.25N$	$2.5N^2$	433	1248615	b, c
Multiple list insertion	Prog. 5.2.1M	No	18	$N+\epsilon(N+100)$	$.0175N^2+18N$	$3.5N^2$	645	35246	b, c, f, i
Merge exchange	Ex. 5.2.2–12	No	36	N	$3.7N(\log_2 N)^2$	$cN\ln^2 N$	1284	379104	h
Partition exchange	Prog. 5.2.2Q	No	63	$N+2\epsilon\log_2 N$	$12.67N\ln N-1.92N$	$\geq 2N^2$	518	81946	
Median-of-3 quicksort	Ex. 5.2.2–55	No	91	$N+2\epsilon\log_2 N$	$11.66N\ln N+1.89N$	$\geq N^2$	567	81485	e
Radix exchange	Prog. 5.2.2R	No	45	$N+68\epsilon$	$14.43N\ln N+23.9N$	$272N$	1135	137614	g, i, j
Straight selection	Prog. 5.2.3S	Yes	15	N	$2.5N^2+3N\ln N$	$3.25N^2$	853	2525287	j
Heapsort	Prog. 5.2.3H	No	30	N	$23.08N\ln N+0.2N$	$26N\ln N$	1068	159714	h, j
List merge	Prog. 5.2.4L	Yes	44	$N(1+\epsilon)$	$14.43N\ln N+4.92N$	$14.4N\ln N$	761	104716	b, c, j
Radix list sort	Prog. 5.2.5R	Yes	36	$N+\epsilon(N+200)$	$32N+4838$	$32N$	4250	36838	b, c

a: three-digit keys only.
b: six-digit (i.e., three-byte) keys only.
c: Output not rearranged; final sequence is specified implicitly by links or counters.
d: Increments chosen as in 5.2.1–8; a slightly better sequence appears in exercise 5.2.1–29.
e: $M = 10$, using SRB; for the version with DIV, add $\frac{13}{7}N$ to the average running time.
f: $M = 100$ (the byte size).
g: $M = 34$, since $2^{34} > 10^{10} > 2^{33}$.
h: Average time is based on empirical estimate, since the theory is incomplete.
i: Average time based on the assumption of uniformly distributed keys.
j: Further refinements, mentioned in the text and exercises accompanying this program, would reduce the running time.

Table 1 summarizes the speed and space characteristics of many of these methods, when programmed for MIX. It is important to realize that the figures in this table are only rough indications of the relative sorting times; they apply to one computer only, and the assumptions made about input data are not completely consistent for all programs. Comparative tables such as this have been given by many authors, with no two people reaching the same conclusions! On the other hand, the timings do give at least an indication of the kind of speed to be expected from each algorithm, when sorting one-word records, since MIX is a fairly typical computer.

The "space" column in Table 1 gives some information about the amount of auxiliary memory used by each algorithm, in units of record length. Here "ϵ" denotes the fraction of a record needed for one link field; thus, for example, $N(1 + \epsilon)$ means that the method requires space for N records plus N link fields.

The asymptotic average and maximum times appearing in Table 1 give only the leading terms which dominate for large N, assuming random input; c denotes an unspecified constant. These formulas can often be misleading, so actual total running times have also been listed, for sample runs of the program on two particular sequences of input data. The case $N = 16$ refers to the sixteen keys which appear in so many of the examples of Section 5.2; and the case $N = 1000$ refers to the sequence $K_1, K_2, \ldots, K_{1000}$ defined by

$$K_0 = 0; \qquad K_{n+1} = (3141592621K_n + 2113148651) \bmod 10^{10}.$$

A reasonably "high-quality" MIX program has been used to represent each algorithm in the table, often incorporating improvements that have been suggested in the exercises. The byte size for these runs was 100.

External sorting techniques are different from internal sorting, because they must use comparatively primitive data structures, and because there is a great emphasis on minimizing their input/output time. Section 5.4.6 summarizes the interesting methods that have been developed for tape merging, and Section 5.4.9 discusses the use of disks and drums.

Of course, sorting isn't the whole story. While studying all of these sorting techniques, we have learned a good deal about how to handle data structures, how to deal with external memories, and how to analyze algorithms; and perhaps we have even learned a little about how to discover new algorithms.

Early developments. A search for the origin of today's sorting techniques takes us back to the nineteenth century, when the first machines for sorting were invented. The United States conducts a census of all its citizens, every ten years, and by 1880 the problem of processing the voluminous census data was becoming very acute; in fact, the total number of single (as opposed to married) people was never tabulated that year, although the necessary information had been gathered. Herman Hollerith, a 20-year-old employee of the Census Bureau, devised an ingenious electric tabulating machine to meet the need for better statistics-gathering, and about 100 of his machines were successfully used to tabulate the 1890 census rolls.

Figure 94 shows Hollerith's original battery-driven apparatus; of chief interest to us is the "sorting box" at the right, which has been opened to show half of the 26 inner compartments. The operator would insert a $6\frac{5}{8}'' \times 3\frac{1}{4}''$ punched card into the "press" and lower the handle; this caused spring-actuated pins in the upper plate to make contact with pools of mercury in the lower plate, wherever a hole was punched in the card. The corresponding completed circuits would cause associated dials on the panel to advance by one unit; and furthermore, one of the 26 lids of the sorting box would pop open. At this point the operator would reopen the press, put the card into the open compartment, and close the lid. One man reportedly ran 19071 cards through this machine in a single $6\frac{1}{2}$-hour working day, an average of about 49 cards per minute! (A typical operator would work at about one-third this speed.)

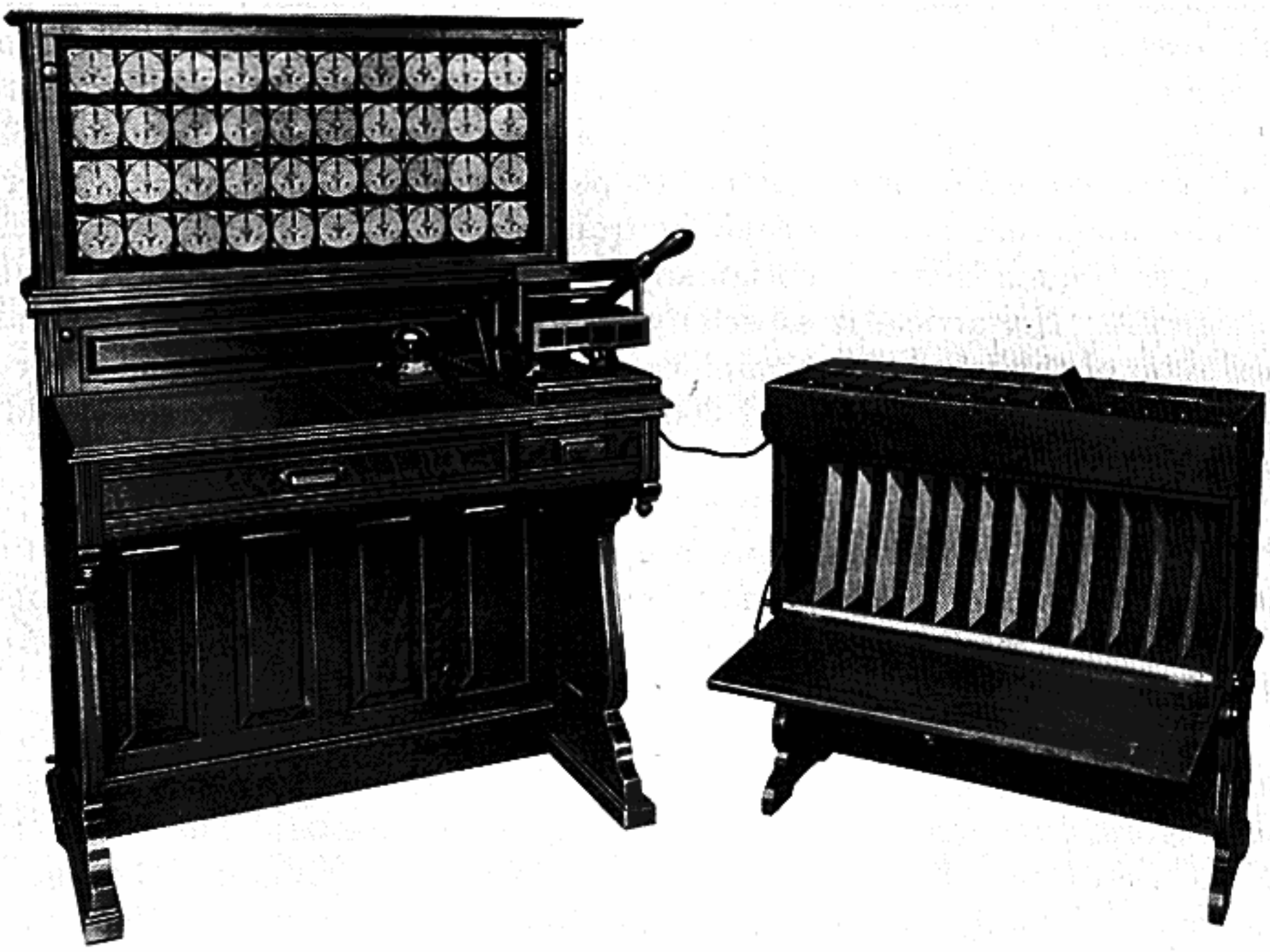


Fig. 94. Hollerith's original tabulating and sorting machine. (Photo courtesy of IBM Archives.)

Population continued its inexorable growth pattern, and the original tabulator-sorters were not fast enough to handle the 1900 census; so Hollerith devised another machine to stave off another data processing crisis. His new device (patented in 1901 and 1904) had an automatic card feed, and in fact it looked essentially like modern card sorters. The story of Hollerith's early machines has been told in interesting detail by Leon E. Truesdell, *The Development of Punch Card Tabulation* (Washington: U.S. Bureau of the Census, 1965); see also the contemporary accounts in *Columbia College School of Mines Quarterly* **10** (1889), 238–255; *J. Franklin Inst.* **129** (1890), 300–306; *The Electrical Engi-*

neer **12** (Nov. 11, 1891), 521–530; *J. Amer. Statistical Assn.* **2** (1891), 330–341, **4** (1895), 365; *J. Royal Statistical Soc.* **55** (1892), 326–327; *Allgemeines Statistisches Archiv* **2** (1892), 78–126; *J. Soc. Statistique de Paris* **33** (1892), 87–96; *U.S. Patents 395781* (1889), *685608* (1901), *777209* (1904). Hollerith and another former Census Bureau employee, James Powers, went on to found rival companies which eventually became part of IBM and Remington Rand corporations, respectively.

Hollerith's sorting machine is, of course, the basis for radix sorting methods now used in digital computers. His patent mentions that two-column numerical items are to be sorted "separately for each column," but he didn't say whether or not the units or the tens columns should be considered first. The nonobvious trick of using the units column first was presumably discovered by some anonymous machine operator and passed on to others (cf. Section 5.2.5); it appears in the earliest extant IBM sorter manual (1936). The first known mention of this right-to-left technique is an incidental remark which appears in an article by L. J. Comrie, *Trans. of the Office Machinery Users' Assoc.* (London, 1930), 25–37. Incidentally, Comrie was the first person to make the important observation that tabulating machines could be fruitfully employed in scientific calculations, even though they were originally designed for statistical and accounting applications. His article is especially interesting because it gives a detailed description of the tabulating equipment available in England in 1930. Sorting machines at that time processed 360 to 400 cards per minute, and could be rented for £9 per month.

The idea of merging goes back to another card-walloping machine, the *collator*, which was a much later invention (1938). With its two feeding stations, it could merge two sorted decks of cards into one, in only one pass; the technique for doing this was clearly explained in the first IBM collator manual (April, 1939). [Cf. James W. Bryce, *U.S. Patent 2189024* (1940).]

Then computers arrived on the scene, and sorting was intimately involved in this development; in fact, there is evidence that a sorting routine was the first program ever written for a stored program computer. The designers of EDVAC were especially interested in sorting, because it epitomized the potential non-numerical applications of computers; they realized that a satisfactory order code should not only be capable of expressing programs for the solution of difference equations, it must also have enough flexibility to handle the combinatorial "decision-making" aspects of algorithms. John von Neumann therefore prepared programs for internal merge sorting in 1945, in order to test the adequacy of some instruction codes he was proposing for the EDVAC computer; the existence of efficient special-purpose sorting machines provided a natural standard by which the merits of his proposed computer organization could be evaluated. Details of this interesting development have been described in an article by D. E. Knuth, *Computing Surveys* **2** (1970), 247–260; see also von Neumann's *Collected Works* **5** (New York: Macmillan, 1963), 196–214, for the final "polished" form of his original sorting programs.

The limited internal memory size planned for early computers made it

natural to think of external sorting as well as internal sorting, and a "Progress Report on the EDVAC" prepared by J. P. Eckert and J. W. Mauchly of the Moore School of Electrical Engineering (September 30, 1945) pointed out that a computer augmented with magnetic wire or tape devices could simulate the operations of card equipment, achieving a faster sorting speed. This progress report described balanced two-way radix sorting, and balanced two-way merging (called "collating"), using four magnetic wire or tape units, reading or writing "at least 5000 pulses per second."

John Mauchly lectured on "Sorting and Collating" at the special session on computing presented at the Moore School in 1946, and the notes of his lecture constitute the first published discussion of computer sorting [*Theory and techniques for the design of electronic digital computers*, ed. by G. W. Patterson, **3** (1946), 22.1–22.20]. Mauchly began his presentation with an interesting remark: "To ask that a single machine combine the abilities to compute and to sort might seem like asking that a single device be able to perform both as a can opener and a fountain pen." Then he observed that machines capable of carrying out sophisticated mathematical procedures must also have the ability to sort and classify data, and he showed that sorting may even be useful in connection with numerical calculations. He described straight insertion and binary insertion, observing that the former method uses about $N^2/4$ comparisons on the average, while the latter never needs more than about $N \log_2 N$. Yet binary insertion requires a rather complex data structure, and he went on to show that two-way merging achieves the same low number of comparisons using only sequential accessing of lists. The last half of his lecture notes were devoted to a discussion of partial-pass radix sorting methods which simulate digital card sorting on four tapes, using less than four passes per digit (cf. Section 5.4.7).

Shortly afterwards, Eckert and Mauchly started a company which produced some of the earliest electronic computers, the BINAC (for military applications) and the UNIVAC (for commercial applications). Again the U.S. Census Bureau played a part in this development, receiving the first UNIVAC. At this time it was not at all clear that computers would be economically profitable; computing machines could sort faster than card equipment, but they cost more. Therefore the UNIVAC programmers, led by Frances E. Holberton, put considerable effort into the design of high-speed external sorting routines, and their preliminary programs also influenced the hardware design. According to their estimates, 100 million 10-word records could be sorted on UNIVAC in 9000 hours (i.e., 375 days).

UNIVAC I, officially dedicated in July, 1951, had an internal memory of 1000 12-character (72-bit) words. It was designed to read and write 60-word blocks on tapes, at a rate of 500 words per second; reading could be either forward or backward, and simultaneous reading/writing/computing was possible. In 1948, Mrs. Holberton devised an interesting way to do two-way merging with perfect overlap of reading, writing, and computing, using six input buffers: Let there be one "current buffer" and two "auxiliary buffers" for each input file; it is possible to merge in such a way that, whenever it is time

to output one block, the two current input buffers contain a total of exactly one block's worth of unprocessed records. Therefore exactly one input buffer becomes empty while each output block is being formed, and we can arrange to have three of the four auxiliary buffers full at all times while we are reading into the other. This method is slightly faster than the forecasting method of Algorithm 5.4.6F, since it is not necessary to inspect the result of one input before initiating the next. [Cf. "Collation Methods for the UNIVAC System," (Eckert-Mauchly Computer Corp., 1950), 2 vols.]

The culmination of this work was a sort generator program, which was the first major "software" routine ever developed for automatic programming. The user would specify the record size, the positions of up to five keys in partial fields of each record, and the "sentinel" keys which mark file's end, and the sort generator would produce a copyrighted sorting program for one-reel files. The first pass of this program was an internal sort of 60-word blocks, using comparison counting (Algorithm 5.2C); then came a number of balanced two-way merge passes, reading backwards and avoiding tape interlock as described above. [Cf. "Master Generating Routine for 2-way Sorting" (Eckert-Mauchly Div. of Remington Rand, 1952); the first draft of this report was entitled "Master Prefabrication Routine for 2-way Collation"! See also F. E. Holberton, *Symposium on Automatic Programming* (Office of Naval Research, 1954), 34–39.]

By 1952, many approaches to internal sorting were well known in the programming folklore, but comparatively little theory had been developed. Daniel Goldenberg ["Time analyses of various methods of sorting data," Digital Computer Laboratory memo M-1680 (Mass. Inst. of Tech., October 17, 1952)] coded five different methods for the Whirlwind computer, and made best-case and worst-case analyses of each program. When sorting one hundred 15-bit words on an 8-bit key, he found that the fastest method was to use a 256-word table, storing each record into a unique position corresponding to its key, then compressing the table. But this technique had an obvious disadvantage, since it would eliminate a record whenever a subsequent one had the same key. The other four methods he analyzed were ranked as follows: straight two-way merging beat radix-2 sorting beat straight selection beat bubble sort.

These results were extended by Harold H. Seward in his 1954 Master's thesis ["Information sorting in the application of electronic digital computers to business operations," Digital Computer Lab. report R-232 (Mass. Inst. of Tech., May 24, 1954; 60 pp.)]. Seward introduced the ideas of distribution counting and replacement selection; he showed that the first run in a random permutation has an average length of $e - 1$; and he analyzed external sorting as well as internal sorting, on various types of bulk memories as well as tapes.

An even more noteworthy thesis—a Ph.D. thesis in fact—was written by Howard B. Demuth in 1956 ["Electronic Data Sorting" (Stanford University, October, 1956), 92 pp.]. This work helped to lay the foundations of computational complexity theory. It considered three abstract models of the sorting

problem, using cyclic, linear, and random-access memories; and optimal or near-optimal methods were developed for each model. (Cf. exercise 5.3.4–6.) Although no practical consequences flowed immediately from Demuth's thesis, it established important ideas about how to link theory with practice.

Thus the history of sorting has been closely associated with many "firsts" in computing: The first data-processing machines, the first stored programs, the first software, the first buffering methods, the first work on algorithmic analysis and computational complexity.

None of the computer-related documents mentioned so far actually appeared in the "open literature"; in fact, most of the early history of computing appears in comparatively inaccessible reports, because comparatively few people were involved with computers at the time. Literature about sorting finally broke into print in 1955–1956, in the form of three major survey articles.

The first paper was prepared by J. C. Hosken [*Proc. Eastern Joint Computer Conference* **8** (1955), 39–55]. He began with an astute observation: "To lower costs per unit of output, people usually increase the size of their operations. But under these conditions, the unit cost of sorting, instead of falling, rises." Hosken surveyed all the available special-purpose equipment then being marketed, as well as the methods of sorting on computers. His bibliography of 54 items is mostly based on manufacturer's brochures.

The comprehensive paper "Sorting on Electronic Computer Systems" by E. H. Friend [*JACM* **3** (1956), 134–168] was a major milestone in the development of sorting. Although numerous techniques have been developed since 1956, this paper is still remarkably up-to-date in many respects. Friend gave careful descriptions of quite a few internal and external sorting algorithms, and he paid special attention to buffering techniques and the characteristics of magnetic tape units. He introduced some new methods (e.g., tree selection, amphisbaenic sorting, and forecasting), and developed some of the mathematical properties of the older methods.

The third survey of sorting to appear about this time was prepared by D. W. Davies [*Proc. Inst. Elect. Engineers* **103B**, Supplement 1 (1956), 87–93]. In the following years several other notable surveys were published, by D. A. Bell [*Comp. J.* **1** (1958), 71–77]; A. S. Douglas [*Comp. J.* **2** (1959), 1–9]; D. D. McCracken, H. Weiss, and T. Lee [*Programming Business Computers* (New York: Wiley, 1959), Chapter 15, pp. 298–332]; I. Flores [*JACM* **8** (1961), 41–80]; K. E. Iverson [*A Programming Language* (New York: Wiley, 1962), Chapter 6, 176–245]; C. C. Gotlieb [*CACM* **6** (1963), 194–201]; T. N. Hibbard [*CACM* **6** (1963), 206–213]; M. A. Goetz [*Digital Computer User's Handbook*, ed. by M. Klerer and G. A. Korn (New York: McGraw-Hill, 1967), Chapter 1.10, pp. 1.292–1.320]. A symposium on sorting was sponsored by ACM in November, 1962; most of the papers presented at that symposium were published in the May, 1963, issue of *CACM*, and they constitute a good representation of the state of the art at that time. C. C. Gotlieb's survey of contemporary sort generators, T. N. Hibbard's survey of minimal storage internal sorting, and

G. U. Hubbard's early exploration of disk file sorting are particularly noteworthy articles in this collection.

New sorting methods were being discovered throughout this period: Address calculation (1956), merge insertion (1959), radix exchange (1959), cascade merge (1959), Shell's diminishing increment sort (1959), polyphase merge (1960), tree insertion (1960), oscillating sort (1962), Hoare's quicksort (1962), Williams's heapsort (1964), Batcher's merge exchange (1964). The history of each individual algorithm has been traced in the particular section of this chapter where that method is described. The late 1960's saw an intensive development of the corresponding theory.

A complete bibliography of all papers on sorting examined by the author as this chapter was being written appears in *Computing Reviews* **13** (1972), 283–289.

EXERCISES

1. [*05*] Summarize the content of this chapter by stating a generalization of Theorem 5.4.6A.

2. [*20*] Based on the information in Table 1, what is the best list-sorting method for six-digit keys, for use on the MIX computer?

3. [*47*] (*Stable sorting in minimum storage.*) A sorting algorithm is said to require *minimum storage* if it uses only $O((\log N)^2)$ bits of memory space for its variables besides the space needed to store the N records. The algorithm must be general in the sense that it works for all N, not just for a particular value of N, assuming that a sufficient amount of random access memory has been made available whenever the algorithm is actually called upon to sort.

Many of the sorting methods we have studied violate this minimum-storage requirement; in particular, the use of N link fields is forbidden. Quicksort (Algorithm 5.2.2Q) satisfies the minimum-storage requirement, but its worst case running time is proportional to N^2. Heapsort (Algorithm 5.2.3H) is the only $O(N \log N)$ algorithm we have studied which uses minimum storage, although another such algorithm could be formulated using the idea of exercise 5.2.4–18.

The fastest general algorithm we have considered which sorts keys in a *stable* manner is the list merge sort (Algorithm 5.2.4L), but it does not use minimum storage. In fact, the only stable minimum-storage sorting algorithms we have seen are $O(N^2)$ methods (straight insertion, bubble sorting, and a variant of straight selection).

Is there a stable minimum-storage sorting algorithm which requires less than $O(N^2)$ units of time in its worst case, and/or on the average?

I shall have accomplished my purpose if I have sorted and put in logical order the gist of the great volume of material which has been generated about sorting over the past few years.

—J. C. HOSKEN (1955)

CHAPTER SIX

SEARCHING

Let's look at the record.

—AL SMITH (1928)

This chapter might have been given the more pretentious title, "Storage and Retrieval of Information"; on the other hand, it might simply have been called "Table Look-Up." We are concerned with the process of collecting information in a computer's memory, and with the subsequent recovery of that information as quickly as possible. Sometimes we are confronted with more data than we can really use, and it may be wisest to forget and to destroy most of it; but at other times it is important to retain and organize the given facts in such a way that fast retrieval is possible.

Most of this chapter is devoted to the study of a very simple search problem: how to find the data that has been stored with a given identification. For example, in a numerical application we might want to find $f(x)$, given x and a table of the values of f; in a nonnumerical application, we might want to find the English translation of a given Russian word.

In general, we shall suppose that a set of N records has been stored, and the problem is to locate the appropriate one. As in the case of sorting, we assume that each record includes a special field called its *key*, perhaps because many people spend so much time every day searching for their keys. We generally require the N keys to be distinct, so that each key uniquely identifies its record. The collection of all records is called a *table* or a *file*, where the word "table" is usually used to indicate a small file, and "file" is usually used to indicate a large table. A large file or a group of files is frequently called a *data base*.

Algorithms for searching are presented with a so-called *argument*, K, and the problem is to find which record has K as its key. After the search is complete, two possibilities can arise: Either the search was *successful*, having located the unique record containing K, or it was *unsuccessful*, having determined that K is nowhere to be found. After an unsuccessful search it is sometimes desirable to enter a new record, containing K, into the table; a method which does this is called a "search and insertion" algorithm. Some hardware devices known as "associative memories" solve the search problem automatically, in a way that resembles the functioning of a human brain; but we shall study techniques for searching on a conventional general-purpose digital computer.

Although the goal of searching is to find the information stored in the record associated with K, the algorithms in this chapter generally ignore everything but the keys themselves. In practice we can find the associated data once we have located K; for example, if K appears in location TABLE $+ i$, the associated data (or a pointer to it) might be in location TABLE $+ i + 1$, or in DATA $+ i$, etc. It is therefore convenient to gloss over the details of what should be done after K has been successfully found.

Searching is the most time-consuming part of many programs, and the substitution of a good search method for a bad one often leads to a substantial increase in speed. In fact it is often possible to arrange the data or the data structure so that searching is eliminated entirely, i.e., so that we always know just where to find the information we need. Linked memory is a common way to achieve this; for example, a doubly-linked list makes it unnecessary to search for the predecessor or successor of a given item. Another way to avoid searching occurs if we are allowed to choose the keys freely, since we might as well let them be the numbers $\{1, 2, \ldots, N\}$; then the record containing K can simply be placed in location TABLE $+ K$. Both of these techniques were used to eliminate searching from the topological sorting algorithm discussed in Section 2.2.3. However, there are many cases when a search is necessary (for example, if the objects in the topological sorting algorithm had been given symbolic names instead of numbers), so it is important to have efficient algorithms for searching.

Search methods might be classified in several ways. We might divide them into internal vs. external searching, just as we divided the sorting algorithms of Chapter 5 into internal vs. external sorting. Or we might divide search methods into static vs. dynamic searching, where "static" means that the contents of the table are essentially unchanging (so that it is important to minimize the search time without regard for the time required to set up the table), and "dynamic" means that the table is subject to frequent insertions (and perhaps also deletions). A third possible scheme is to classify search methods according to whether they are based on comparisons between keys or on digital properties of the keys, analogous to the distinction between sorting by comparison and sorting by distribution. Finally we might divide searching into those methods which use the actual keys and those which work with transformed keys.

The organization of this chapter is essentially a combination of the latter two modes of classification. Section 6.1 considers "brute force" sequential methods of search, then Section 6.2 discusses the improvements which can be made based on comparisons between keys, using alphabetic or numeric order to govern the decisions. Section 6.3 treats digital searching, and Section 6.4 discusses an important class of methods called hashing techniques, based on arithmetic transformations of the actual keys. Each of these sections treats both internal and external searching, in both the static and the dynamic case;

and each section points out the relative advantages and disadvantages of the various algorithms.

There is a certain amount of interaction between searching and sorting. For example, consider the following problem:

> Given two sets of numbers, $A = \{a_1, a_2, \ldots, a_m\}$ and $B = \{b_1, b_2, \ldots, b_n\}$, determine whether or not $A \subseteq B$.

Three solutions suggest themselves, namely

1. Compare each a_i sequentially with the b_j's until finding a match.
2. Enter the b_j's in a table, then search for each of the a_i.
3. Sort the a's and b's, then make one sequential pass through both files, checking the appropriate condition.

Each of these solutions is attractive for a different range of values of m and n. Solution 1 will take roughly c_1mn units of time, for some constant c_1, and solution 3 will take about $c_2(m \log_2 m + n \log_2 n)$ units, for some (larger) constant c_2. With a suitable hashing method, solution 2 will take roughly $c_3m + c_4n$ units of time, for some (still larger) constants c_3 and c_4. It follows that solution 1 is good for very small m and n, but solution 3 soon becomes better as m and n grow larger. Eventually solution 2 becomes preferable, until n exceeds the internal memory size; then solution 3 is usually again superior until n gets much larger still. Thus we have a situation where sorting is sometimes a good substitute for searching, and searching is sometimes a good substitute for sorting.

More complicated search problems can often be reduced to the simpler case considered here. For example, suppose that the keys are words which might be slightly misspelled; we might want to find the correct record in spite of this error. If we make two copies of the file, one in which the keys are in normal alphabetic order and another in which they are ordered from right to left (as if the words were spelled backwards), a misspelled search argument will probably agree up to half or more of its length with an entry in one of these two files. The search methods of Sections 6.2 and 6.3 can therefore be adapted to find the key that was probably intended.

A related problem has received considerable attention in connection with airline reservation systems, and in other applications involving people's names when there is a good chance that the name will be misspelled due to poor handwriting or voice transmission. The goal is to transform the argument into some code that tends to bring together all variants of the same name. The following "Soundex" method, which was originally developed by Margaret K. Odell and Robert C. Russell [cf. *U.S. Patents 1261167* (1918), *1435663* (1922)], has often been used for encoding surnames:

1. Retain the first letter of the name, and drop all occurrences of a, e, h, i, o, u, w, y in other positions.
2. Assign the following numbers to the remaining letters after the first:

   b, f, p, v → 1
   c, g, j, k, q, s, x, z → 2
   d, t → 3
   l → 4
   m, n → 5
   r → 6

3. If two or more letters with the same code were adjacent in the original name (before step 1), omit all but the first.
4. Convert to the form "letter, digit, digit, digit" by adding trailing zeros (if there are less than three digits), or by dropping rightmost digits (if there are more than three).

For example, the names Euler, Gauss, Hilbert, Knuth, Lloyd, and Łukasiewicz have the respective codes E460, G200, H416, K530, L300, L222. Of course this system will bring together names that are somewhat different, as well as names that are similar; the same six codes would be obtained for Ellery, Ghosh, Heilbronn, Kant, Ladd, and Lissajous. And on the other hand a few related names like Rogers and Rodgers, or Sinclair and St. Clair, or Tchebysheff and Chebyshev, remain separate. But by and large the Soundex code greatly increases the chance of finding a name in one of its disguises. [For further information, cf. C. P. Bourne and D. F. Ford, *JACM* **8** (1961), 538–552; Leon Davidson, *CACM* **5** (1962), 169–171; *Federal Population Censuses* 1790–1890 (Washington, D.C.: National Archives, 1971), 90.]

When using a scheme like Soundex, we need not give up the assumption that all keys are distinct; we can make lists of all records with equivalent codes, treating each list as a unit.

Large data bases tend to make the retrieval process more complex, since people often want to consider many different fields of each record as potential keys, with the ability to locate items when only part of the key information is specified. For example, given a large file about stage performers, a producer might wish to find all unemployed actresses between 25 and 30 with dancing talent and a French accent; given a large file of baseball statistics, a sports-writer may wish to determine the total number of runs scored by the Cincinnati Redlegs in 1964, during the seventh inning of night games, against lefthanded pitchers. Given a large file of data about anything, people like to ask arbitrarily complicated questions. Indeed, we might consider an entire library as a data base, and a searcher may want to find everything that has been published about information retrieval. An introduction to the techniques for such *multi-attribute retrieval* problems appears below in Section 6.5.

Before entering into a detailed study of searching, it may be helpful to put things in historical perspective. During the pre-computer era, many books of logarithm tables, trigonometry tables, etc., were compiled, so that mathematical calculations could be replaced by searching. Eventually these tables were transferred to punched cards, and used for scientific problems in connection

with collators, sorters, and duplicating punch machines. But when stored-program computers were introduced, it soon became apparent that it was now cheaper to recompute $\log x$ or $\cos x$ each time, instead of looking up the answer in a table.

Although the problem of sorting received considerable attention already in the earliest days of computers, comparatively little was done about algorithms for searching. With small internal memories, and with nothing but sequential media like tapes for storing large files, searching was either trivially easy or almost impossible.

But the development of larger and larger random-access memories during the 1950's eventually led to the recognition that searching was an interesting problem in its own right. After years of complaining about the limited amounts of space in the early machines, programmers were suddenly confronted with larger amounts of memory than they knew how to use efficiently.

The first surveys of the searching problem were published by A. I. Dumey, *Computers & Automation* **5**, 12 (December 1956), 6–9; W. W. Peterson, *IBM J. Research & Development* **1** (1957), 130–146; A. D. Booth, *Information and Control* **1** (1958), 159–164; A. S. Douglas, *Comp. J.* **2** (1959), 1–9. More extensive treatments were given later by Kenneth E. Iverson, *A Programming Language* (New York: Wiley, 1962), 133–158, and by Werner Buchholz, *IBM Systems J.* **2** (1963), 86–111.

During the early 1960's, a number of interesting new search procedures based on tree structures were introduced, as we shall see; and research about searching is still actively continuing at the present time.

6.1. SEQUENTIAL SEARCHING

"Begin at the beginning, and go on till you find the right key: then stop." This sequential procedure is the obvious way to search, and it makes a useful starting point for our discussion of searching because many of the more intricate algorithms are based on it. We shall see that sequential searching involves some very interesting ideas, in spite of its simplicity.

The algorithm might be formulated more precisely as follows:

Algorithm S (*Sequential search*). Given a table of records $R_1, R_2, \ldots, R_N$, whose respective keys are $K_1, K_2, \ldots, K_N$, this algorithm searches for a given argument K. We assume that $N \geq 1$.

S1. [Initialize.] Set $i \leftarrow 1$.

S2. [Compare.] If $K = K_i$, the algorithm terminates successfully.

S3. [Advance.] Increase i by 1.

S4. [End of file?] If $i \leq N$, go back to S2. Otherwise the algorithm terminates unsuccessfully. ▮

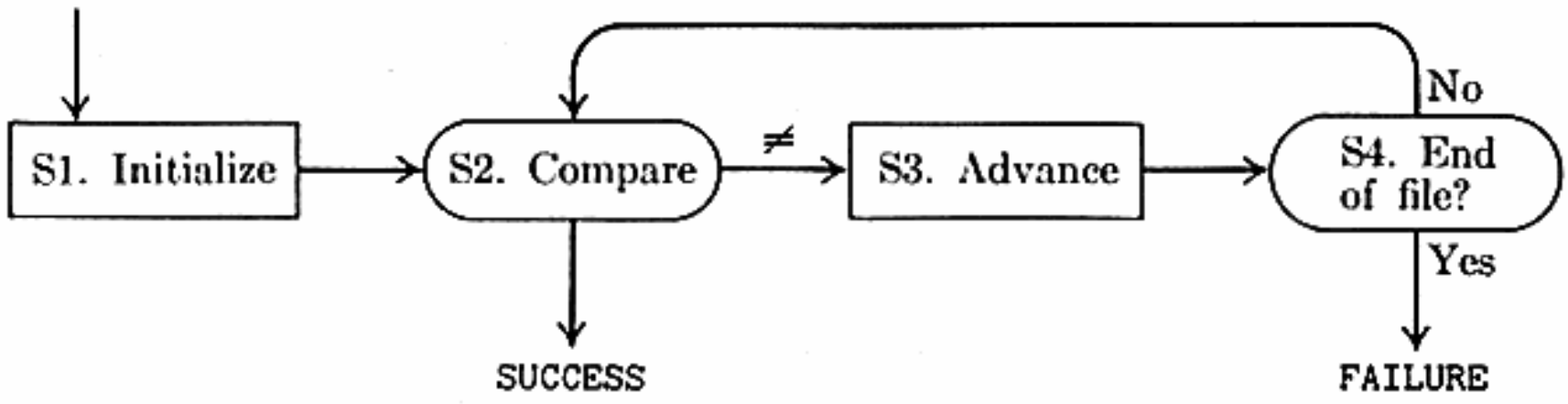


Fig. 1. Sequential search.

Note that this algorithm can terminate in two different ways, *successfully* (having located the desired key) or *unsuccessfully* (having established that the given argument is not present in the table). The same will be true of most other algorithms in this chapter.

A MIX program can be written down immediately:

Program S (*Sequential search*). Assume that K_i appears in location KEY + i, and that the remainder of record R_i appears in location INFO + i. The following program uses rA $\equiv K$, rI1 $\equiv i - N$.

```
01    START   LDA   K          1        S1. Initialize.
02            ENT1  1-N        1        i ← 1.
03    2H      CMPA  KEY+N,1    C        S2. Compare.
04            JE    SUCCESS    C        Exit if K = K_i.
05            INC1  1          C - S    S3. Advance.
06            J1NP  2B         C - S    S4. End of file?
07    FAILURE EQU   *          1 - S    Exit if not in table.
```

At location SUCCESS, the instruction "LDA INFO+N,1" will now bring the desired information into rA. ∎

The analysis of this program is straightforward; it shows that the running time of Algorithm S depends on two things,

$$\begin{aligned} C &= \text{the number of key comparisons;} \\ S &= 1 \text{ if successful, } 0 \text{ if unsuccessful.} \end{aligned} \tag{1}$$

Program S takes $5C - 2S + 3$ units of time. If the search successfully finds $K = K_i$, we have $C = i$, $S = 1$; hence the total time is $(5i + 1)u$. On the other hand if the search is unsuccessful, we have $C = N$, $S = 0$, for a total time of $(5N + 3)u$. If every input key occurs with equal probability, the average value of C in a successful search will be

$$\frac{1 + 2 + \cdots + N}{N} = \frac{N + 1}{2}\,; \tag{2}$$

the standard deviation is, of course, rather large, about $0.289N$ (see exercise 1).

The above algorithm is surely familiar to all programmers. But too few people know that it is *not* always the right way to do a sequential search! A straightforward change makes the algorithm faster, unless the list of records is quite short:

Algorithm Q (*Quick sequential search*). This algorithm is the same as Algorithm S, except that it assumes the presence of a "dummy" record R_{N+1} at the end of the file.

Q1. [Initialize.] Set $i \leftarrow 1$, and set $K_{N+1} \leftarrow K$.

Q2. [Compare.] If $K = K_i$, go to Q4.

Q3. [Advance.] Increase i by 1 and return to Q2.

Q4. [End of file?] If $i \leq N$, the algorithm terminates successfully; otherwise it terminates unsuccessfully ($i = N + 1$). ∎

Program Q (*Quick sequential search*). rA $\equiv K$, rI1 $\equiv i - N$.

```
01    BEGIN    LDA    K           1          Q1. Initialize.
02             STA    KEY+N+1     1          K_{N+1} ← K.
03             ENT1   -N          1          i ← 0.
04             INC1   1           C+1-S      Q3. Advance.
05             CMPA   KEY+N,1     C+1-S      Q2. Compare.
06             JNE    *-2         C+1-S      To Q3 if K_i ≠ K.
07             J1NP   SUCCESS     1          Q4. End of file?
08    FAILURE  EQU    *           1-S        Exit if not in table. ∎
```

In terms of the quantities C and S in the analysis of Program S, the running time has decreased to $(4C - 4S + 10)u$; this is an improvement whenever $C \geq 6$ in a successful search, and it is an improvement whenever $N \geq 8$ in an unsuccessful search.

The transition from Algorithm S to Algorithm Q makes use of an important "speed-up" principle: When an inner loop of a program tests two or more conditions, an attempt should be made to reduce it to just one condition.

Another technique will make Program Q *still* faster.

Program Q′ (*Quicker sequential search*). rA $\equiv K$, rI1 $\equiv i - N$.

```
01    BEGIN    LDA    K           1                  Q1. Initialize.
02             STA    KEY+N+1     1                  K_{N+1} ← K.
03             ENT1   -1-N        1                  i ← -1.
04    3H       INC1   2           ⌊(C-S+2)/2⌋        Q3. Advance. (twice)
05             CMPA   KEY+N,1     ⌊(C-S+2)/2⌋        Q2. Compare.
06             JE     4F          ⌊(C-S+2)/2⌋        To Q4 if K = K_i.
07             CMPA   KEY+N+1,1   ⌊(C-S+1)/2⌋        Q2. Compare. (next)
08             JNE    3B          ⌊(C-S+1)/2⌋        To Q3 if K ≠ K_{i+1}.
09             INC1   1           (C-S) mod 2        Advance i.
10    4H       J1NP   SUCCESS     1                  Q4. End of file?
11    FAILURE  EQU    *           1-S                Exit if not in table. ∎
```

The inner loop has been duplicated; this avoids about half of the "$i \leftarrow i + 1$" instructions, so it reduces the running time to

$$3.5C - 3.5S + 10 + \frac{(C - S) \bmod 2}{2}$$

units. We have saved 30 percent of the running time of Program S, when large tables are being searched; many existing programs can be improved in this way.

A slight variation of the algorithm is appropriate if we know that the keys are in increasing order:

Algorithm T (*Sequential search in ordered table*). Given a table of records $R_1, R_2, \ldots, R_N$ whose keys are in increasing order $K_1 < K_2 < \cdots < K_N$, this algorithm searches for a given argument K. For convenience and speed, the algorithm assumes that there is a dummy record R_{N+1} whose key value is $K_{N+1} = \infty > K$.

T1. [Initialize.] Set $i \leftarrow 1$.

T2. [Compare.] If $K \leq K_i$, go to T4.

T3. [Advance.] Increase i by 1 and return to T2.

T4. [Equality?] If $K = K_i$, the algorithm terminates successfully. Otherwise it terminates unsuccessfully. ▮

If all input keys are equally likely, this algorithm takes essentially the same average time as Algorithm Q, for a successful search. But unsuccessful searches are performed about twice as fast, since the absence of a record can be established more quickly.

Each of the above algorithms uses subscripts to denote the table entries. It is convenient to describe the methods in terms of these subscripts, but the same search procedures can be used for tables which have a *linked* representation, since the data is being traversed sequentially. (See exercises 2, 3, and 4.)

Frequency of access. So far we have been assuming that every argument occurs as often as every other. This is not always a realistic assumption; in a general situation, key K_i will occur with probability p_i, where $p_1 + p_2 + \cdots + p_N = 1$. The time required to do a successful search is essentially proportional to the number of comparisons, C, which now has the average value

$$\overline{C}_N = p_1 + 2p_2 + \cdots + Np_N. \tag{3}$$

If we have the option of putting the records into the table in any desired order, this quantity $\overline{C}_N$ is smallest when

$$p_1 \geq p_2 \geq \cdots \geq p_N, \tag{4}$$

i.e., when the most frequently used records appear near the beginning.

Let's look at several probability distributions, in order to see how much of a saving is possible when the records are arranged in the "optimal" manner specified in (4). If $p_1 = p_2 = \cdots = p_N = 1/N$, formula (3) reduces to $\overline{C}_N = (N+1)/2$; we have already derived this in Eq. (2). Suppose, on the other hand, that

$$p_1 = \frac{1}{2}, \quad p_2 = \frac{1}{4}, \quad \cdots, \quad p_{N-1} = \frac{1}{2^{N-1}}, \quad p_N = \frac{1}{2^{N-1}}. \tag{5}$$

Then by exercise 7, $\overline{C}_N = 2 - 2^{1-N}$; the average number of comparisons is *less than two*, for this distribution, if the records appear in the proper order within the table.

Another probability distribution that suggests itself is

$$p_1 = Nc, \qquad p_2 = (N-1)c, \qquad \ldots, \qquad p_N = c,$$

where

$$c = 2/N(N+1). \tag{6}$$

This "wedge-shaped" distribution is not as dramatic a departure from uniformity as (5). In this case we find

$$\overline{C}_N = c \sum_{1 \le k \le N} k \cdot (N+1-k) = \frac{N+2}{3}; \tag{7}$$

the optimum arrangement saves about one-third of the search time which would have been obtained if the records had appeared in random order.

Of course the probability distributions in (5) and (6) are rather artificial, and they may never be a very good approximation to reality. A more typical distribution is "Zipf's law,"

$$p_1 = c/1, \quad p_2 = c/2, \quad \ldots, \quad p_N = c/N, \quad \text{where } c = 1/H_N. \tag{8}$$

This distribution was formulated by G. K. Zipf, who observed that the nth most common word in natural language text seems to occur with a frequency inversely proportional to n. [*Human Behavior and the Principle of Least Effort, an Introduction to Human Ecology* (Reading, Mass.: Addison-Wesley, 1949).] He observed the same phenomenon in census tables, when metropolitan areas are ranked in order of decreasing population. If Zipf's law governs the frequency of the keys in a table, we have immediately

$$\overline{C}_N = N/H_N; \tag{9}$$

searching such a file is about $\frac{1}{2} \ln N$ times as fast as searching the same file with randomly-ordered records. [Cf. A. D. Booth et al., *Mechanical Resolution of Linguistic Problems* (New York: Academic Press, 1958), 79.]

Another approximation to realistic distributions is the "80-20" rule of thumb that has been commonly observed in commercial applications [cf. W. P. Heising, *IBM Systems J.* **2** (1963), 114–115]. This rule states that 80 percent of the transactions deal with the most active 20 percent of a file; and the same applies to this 20 percent, so that 64 percent of the transactions deal with the most active 4 percent, etc. In other words,

$$\frac{p_1 + p_2 + \cdots + p_{.20n}}{p_1 + p_2 + p_3 + \cdots + p_n} \approx .80, \qquad \text{for all } n. \tag{10}$$

One distribution which satisfies this rule exactly whenever n is a multiple of 5 is

$$p_1 = c, \quad p_2 = (2^\theta - 1)c, \quad p_3 = (3^\theta - 2^\theta)c, \quad \ldots, \quad p_N = (N^\theta - (N-1)^\theta)c, \tag{11}$$

where

$$c = 1/N^\theta, \qquad \theta = \frac{\log .80}{\log .20} = 0.1386, \tag{12}$$

since $p_1 + p_2 + \cdots + p_n = cn^\theta$ for all n in this case. It is not especially easy to work with the probabilities in (11); we have, however, $n^\theta - (n-1)^\theta = \theta n^{\theta-1}(1 + O(1/n))$, so there is a simpler distribution which approximately fulfills the 80-20 rule, namely

$$p_1 = c/1^{1-\theta}, \quad p_2 = c/2^{1-\theta}, \quad \ldots, \quad p_N = c/N^{1-\theta}, \quad \text{where } c = 1/H_N^{(1-\theta)}. \tag{13}$$

Here $\theta = \log .80/\log .20$ as before, and $H_N^{(s)}$ is the Nth harmonic number of order s, namely $1^{-s} + 2^{-s} + \cdots + N^{-s}$. Note that this probability distribution is very similar to that of Zipf's law (8); as θ varies from 1 to 0, the probabilities vary from a uniform distribution to a Zipfian one. (Indeed, Zipf found that $\theta \approx \frac{1}{2}$ in the distribution of personal income.). Applying (3) to (13) yields

$$C_N = H_N^{(-\theta)}/H_N^{(1-\theta)} = \frac{\theta N}{\theta + 1} + O(N^{1-\theta}) \approx 0.122N \tag{14}$$

as the mean number of comparisons for the 80-20 law (see exercise 8).

A study of word frequencies carried out by E. S. Schwartz [see the interesting graph on p. 422 of *JACM* **10** (1963)] suggests that a more appropriate substitute for Zipf's law is

$$p_1 = c/1^{1+\theta}, \; p_2 = c/2^{1+\theta}, \; \ldots, \; p_N = c/N^{1+\theta}, \; \text{where } c = 1/H_N^{(1+\theta)}, \tag{15}$$

for a small *positive* value of θ. [Cf. with (13); the sign of θ has been reversed.] In this case

$$C_N = H_N^{(\theta)}/H_N^{(1+\theta)} = N^{1-\theta}/(1-\theta)\zeta(1+\theta) + O(N^{1-2\theta}), \tag{16}$$

which is substantially smaller than (9) as $N \to \infty$.

A "self-organizing" file. The above calculations with probabilities are very nice, but in most cases we don't know the probabilities are. We could keep a count in each record of how often it has been accessed, reallocating the records on the basis of these counts; the formulas derived above suggest that this procedure would often lead to a worthwhile savings. But we probably don't want to devote so much memory space to the count fields, since we can make better use of that memory (e.g. by using nonsequential search techniques which are explained later in this chapter).

A simple scheme, which has been in use for many years although its origin is unknown, can be used to keep the records in a pretty good order without

auxiliary count fields: Whenever a record has been successfully located, it is moved to the beginning of the table. This procedure is readily implemented if the table is a linked linear list, especially because the record being moved to the beginning often has to be substantially updated anyway.

The idea behind this "self-organizing" technique is that the oft-used items will tend to be located fairly near the beginning of the table, when we need them. If we assume that the N keys occur with respective probabilities $\{p_1, p_2, \ldots, p_N\}$, with each search being completely *independent* of previous searches, it can be shown that the average number of comparisons needed to find an item in such a self-organizing file tends to the limiting value

$$\overline{C}_N = 1 + 2 \sum_{1 \le i < j \le N} \frac{p_i p_j}{p_i + p_j} = \frac{1}{2} + \sum_{i,j} \frac{p_i p_j}{p_i + p_j}. \tag{17}$$

(See exercise 11.) For example, if $p_i = 1/N$ for $1 \le i \le N$, the self-organizing table is always in completely random order, and this formula reduces to the familiar expression $(N + 1)/2$ derived above.

Let us see how well the self-organizing procedure works when the key probabilities obey Zipf's law (8). We have

$$\begin{aligned}
\overline{C}_N &= \frac{1}{2} + \sum_{1 \le i,j \le N} \frac{(c/i)(c/j)}{c/i + c/j} = \frac{1}{2} + c \sum_{1 \le i,j \le N} \frac{1}{i + j} \\
&= \frac{1}{2} + c \sum_{1 \le i \le N} (H_{N+i} - H_i) = \frac{1}{2} + c \sum_{1 \le i \le 2N} H_i - 2c \sum_{1 \le i \le N} H_i \\
&= \tfrac{1}{2} + c((2N + 1)H_{2N} - 2N - 2(N + 1)H_N + 2N) \\
&= \tfrac{1}{2} + c(N \ln 4 - \ln N + O(1)) \\
&\approx 2N/\log_2 N,
\end{aligned} \tag{18}$$

by Eqs. 1.2.7–8, 3. This is substantially better than $\frac{1}{2}N$, when N is reasonably large, and it is only about $\ln 4 \approx 1.386$ times as many comparisons as would be obtained in the optimum arrangement (cf. 9).

Computational experiments involving actual compiler symbol tables indicate that the self-organizing method works even better than the above formulas predict, because successive searches are not independent (small groups of keys tend to occur in bunches).

A somewhat similar self-organizing scheme has been studied by G. Schay, Jr. and F. W. Dauer, *SIAM J. Appl. Math.* **15** (1967), 874–888.

Tape searching with unequal-length records. Now let's give the problem still another twist: Suppose the table we are searching is stored on tape, and the individual records have varying lengths. For example, in an old-fashioned operating system, the "system library tape" was such a file; standard system programs such as compilers, assemblers, loading routines, report generators, etc. were the "records" on this tape, and most user jobs would start by searching

down the tape until the appropriate routine had been input. This setup makes our previous analysis of Algorithm S inapplicable, since step S3 takes a variable amount of time each time we reach it. The number of comparisons is therefore not the only criterion of interest.

Let L_i be the length of record R_i, and let p_i be the probability that this record will be sought. The running time of the search method will now be approximately proportional to

$$p_1L_1 + p_2(L_1 + L_2) + \cdots + p_N(L_1 + L_2 + L_3 + \cdots + L_N). \tag{19}$$

When $L_1 = L_2 = \cdots = L_N = 1$, this reduces to (3), the case already studied.

It seems logical to put the most frequently-needed records at the beginning of the tape; but this is sometimes a bad idea! For example, assume that the tape contains just two programs, A and B; A is needed twice as often as B, but it is four times as long. Thus, $N = 2$, $p_A = \frac{2}{3}$, $L_A = 4$, $p_B = \frac{1}{3}$, $L_B = 1$. If we place A first on tape, according to the "logical" principle stated above, the average running time is $\frac{2}{3} \cdot 4 + \frac{1}{3} \cdot 5 = \frac{13}{3}$; but if we use an "illogical" idea, placing B first, the average running time is reduced to $\frac{1}{3} \cdot 1 + \frac{2}{3} \cdot 5 = \frac{11}{3}$.

The optimum arrangement of programs on a library tape may be determined as follows.

Theorem S. *Let L_i and p_i be as defined above. The arrangement of records in the table is optimal if and only if*

$$p_1/L_1 \geq p_2/L_2 \geq \cdots \geq p_N/L_N. \tag{20}$$

In other words, the minimum value of

$$p_{a_1}L_{a_1} + p_{a_2}(L_{a_1} + L_{a_2}) + \cdots + p_{a_N}(L_{a_1} + \cdots + L_{a_N}),$$

over all permutations $a_1\, a_2 \ldots a_N$ of $\{1, 2, \ldots, N\}$, is equal to (19) if and only if (20) holds.

Proof. Suppose that R_i and R_{i+1} are interchanged on the tape; the cost (19) changes from

$$\cdots + p_i(L_1 + \cdots + L_{i-1} + L_i) + p_{i+1}(L_1 + \cdots + L_{i+1}) + \cdots$$

to

$$\cdots + p_{i+1}(L_1 + \cdots + L_{i-1} + L_{i+1}) + p_i(L_1 + \cdots + L_{i+1}) + \cdots,$$

a net change of $p_iL_{i+1} - p_{i+1}L_i$. Therefore if $p_i/L_i < p_{i+1}/L_{i+1}$, such an interchange will improve the running time, and the given arrangement is not optimal. It follows that (20) holds in any optimal arrangement.

Conversely, assume that (20) holds; we need to prove that the arrangement is optimal. The argument just given shows that the arrangement is "locally optimal" in the sense that adjacent interchanges make no improvement; but

there may conceivably be a long, complicated sequence of interchanges which leads to a better "global optimum." We shall consider two proofs, one which uses computer science and one which uses a mathematical trick.

First proof. Assume that (20) holds. We know that any permutation of the records can be "sorted" into the order $R_1 R_2 \ldots R_N$ by using a sequence of interchanges of adjacent records. Each of these interchanges replaces $\ldots R_j R_i \ldots$ by $\ldots R_i R_j \ldots$ for some $i < j$, so it decreases the search time by the nonnegative amount $p_i L_j - p_j L_i$. Therefore the order $R_1 R_2 \ldots R_N$ must have minimum search time.

Second proof. Replace each probability p_i by

$$p_i(\epsilon) = p_i + \epsilon^i - (\epsilon^1 + \epsilon^2 + \cdots + \epsilon^N)/N, \tag{21}$$

where ϵ is an extremely small positive number. When ϵ is sufficiently small, we will never have $x_1 p_1(\epsilon) + \cdots + x_N p_N(\epsilon) = y_1 p_1(\epsilon) + \cdots + y_N p_N(\epsilon)$ unless $x_1 = y_1, \ldots, x_N = y_N$; and in particular, equality will not hold in (20). Consider now the $N!$ permutations of the records; at least one of these is optimum, and we know that it satisfies (20); but only one permutation satisfies (20) because there are no equalities. Therefore (20) uniquely characterizes the optimum arrangement of records in the table for the probabilities $p_i(\epsilon)$, whenever ϵ is sufficiently small. By continuity, the same arrangement must also be optimum when ϵ is set equal to zero. (This "tie-breaking" type of proof is often useful in connection with combinatorial optimization.) ∎

Theorem S is due to W. E. Smith, *Naval Research Logistics Quarterly* **3** (1956), 59–66. The exercises below contain further results about optimum file arrangements.

File compression. Sequential searching on tape and other external memory devices goes faster if we can pack the data so that it takes up less space; therefore it is a good idea to consider alternative ways to represent a file. We need not always store the keys explicitly.

For example, suppose that we want to have a table of all the prime numbers less than one million, for use in factoring 12-digit numbers. (Cf. Section 4.5.4.) There are 78498 such primes; so if we use 20 bits for each one, the file will be 1,569,960 bits long. This is obviously wasteful, since we could have a million-bit table, with each bit telling us whether or not the corresponding number is prime. Since all primes (except 2) are odd, the file could in fact be shortened to 500,000 bits.

Another way to cut down the size of this file is to list the sizes of *gaps* between primes, instead of the primes themselves. According to Table 1, the difference $(p_{k+1} - p_k)/2$ is less than 64 for all primes less than 1,357,201, so we can represent all primes between 3 and 1000000 by simply storing 78496 six-bit values of (gap size)/2; this makes the file about 471,000 bits long. Further compression can be achieved by using a variable-length binary code for the gaps (cf. Section 6.2.2).

Table 1

RECORD-BREAKING GAPS BETWEEN CONSECUTIVE PRIME NUMBERS

Gap $(p_{k+1} - p_k)$	p_k	Gap $(p_{k+1} - p_k)$	p_k
1	2	52	19609
2	3	72	31397
4	7	86	155921
6	23	96	360653
8	89	112	370261
14	113	114	492113
18	523	118	1349533
20	887	132	1357201
22	1129	148	2010733
34	1327	154	4652353
36	9551	180	17051707
44	15683	210	20831323

This table lists $p_{k+1} - p_k$ whenever it exceeds $p_{j+1} - p_j$ for all $j < k$. For further information, see "Statistics on the first six million prime numbers" by F. Gruenberger and G. Armerding, RAND Corp. report P-2460 (October, 1961).

EXERCISES

1. [*M20*] When all the search keys are equally probable, what is the standard deviation of the number of comparisons made in a successful sequential search through a table of N records?

2. [*16*] Restate the steps of Algorithm S, using linked-memory notation instead of subscript notation. (If `P` points to a record in the table, assume that `KEY(P)` is the key, `INFO(P)` is the associated information, and that `LINK(P)` is a pointer to the next record. Assume also that `FIRST` points to the first record, and that the last record points to Λ.)

3. [*16*] Write a `MIX` program for the algorithm of exercise 2. What is the running time of your program, in terms of the quantities C and S in (1)?

▸ **4.** [*17*] Does the idea of Algorithm Q carry over from subscript notation to linked-memory notation? (Cf. exercise 2.)

5. [*20*] Program Q′ is, of course, noticeably faster than Program Q, when C is large. But are there any small values of C and S for which Program Q′ actually takes more time than Program Q?

▸ **6.** [*20*] Add three more instructions to Program Q′, reducing its running time to about $(3.33C + \text{constant})u$.

7. [*M20*] Evaluate the average number of comparisons, (3), using the "binary" probability distribution (5).

8. [*HM22*] Find an asymptotic series for $H_n^{(x)}$ as $n \to \infty$, when $x \neq 1$.

▸ **9.** [*M23*] The text observes that the probability distributions given by (11) and (13) are roughly equivalent, and that the mean number of comparisons using (13) is $\theta N/(\theta+1) + O(N^{1-\theta})$. Is the mean number of comparisons equal to $\theta N/(\theta+1) + O(N^{1-\theta})$ also when the probabilities of (11) are used?

10. [*M20*] The best arrangement of records in a sequential table is specified by (4); what is the *worst* arrangement? Show that the average number of comparisons in the worst arrangement has a simple relation to the average number of comparisons in the best arrangement.

11. [*M30*] The purpose of this exercise is to analyze the behavior of the text's self-organizing file. First we need to define some rather complicated notation: Let $f_m(x_1, x_2, \ldots, x_m)$ be the infinite sum of all distinct ordered products $x_{i_1}x_{i_2}\ldots x_{i_k}$ such that $1 \le i_1, \ldots, i_k \le m$, where each of $x_1, x_2, \ldots, x_m$ appears in every term. For example,

$$f_2(x, y) = \sum_{j,k\ge 0} (x^{1+j}y(x+y)^k + y^{1+j}x(x+y)^k) = \frac{xy}{1-x-y}\left(\frac{1}{1-x}+\frac{1}{1-y}\right).$$

Given a set X of n variables $\{x_1, \ldots, x_n\}$, let

$$p_{nm} = \sum_{1\le j_1<\cdots<j_m\le n} f_m(x_{j_1}, \ldots, x_{j_m}); \quad Q_{nm} = \sum_{1\le j_1<\cdots<j_m\le n} \frac{1}{1-x_{j_1}-\cdots-x_{j_m}}.$$

For example, $P_{32} = f_2(x_1, x_2) + f_2(x_1, x_3) + f_2(x_2, x_3)$ and $Q_{32} = 1/(1-x_1-x_2) + 1/(1-x_1-x_3) + 1/(1-x_2-x_3)$. By convention we set $P_{n0} = Q_{n0} = 1$.

a) Assume that the text's self-organizing file has been servicing requests for item R_i with probability p_i. After the system has been running a long time, show that R_i will be the mth item from the front with limiting probability $p_i P_{N-1,m-1}$, where $X = \{p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_N\}$.

b) By summing the result of (a) for $m = 1, 2, \ldots$, we obtain the identity

$$P_{nn} + P_{n,n-1} + \cdots + P_{n0} = Q_{nn}.$$

Prove that, consequently,

$$P_{nm} + \binom{n-m+1}{1} P_{n,m-1} + \cdots + \binom{n-m+m}{m} P_{n0} = Q_{nm};$$

$$Q_{nm} - \binom{n-m+1}{1} Q_{n,m-1} + \cdots + (-1)^m \binom{n-m+m}{m} Q_{n0} = P_{nm}.$$

c) Compute the limiting average distance $d_i = \sum_{m\ge 1} m p_i P_{N-1,m-1}$ of R_i from the front of the list; then evaluate $\bar{C}_N = \sum_{1\le i\le n} p_i d_i$.

12. [*M23*] Use (17) to evaluate the average number of comparisons needed to search the self-organizing file when the search keys have the binary probability distribution (5).

13. [*M27*] Use (17) to evaluate $\bar{C}_N$ for the probability distribution (6).

14. [*M21*] Given two sequences $\langle x_1, x_2, \ldots, x_n\rangle$ and $\langle y_1, y_2, \ldots, y_n\rangle$ of real numbers, what permutation $a_1\, a_2 \ldots a_n$ of the subscripts will make $\sum x_i y_{a_i}$ a maximum? a minimum?

▸**15.** [*M22*] The text shows how to arrange programs optimally on a "system library tape," when only one program is being sought. But another set of assumptions is more appropriate for a *subroutine* library tape, from which we may wish to load various subroutines called for in a user's program.

For this case let us suppose that subroutine i is desired with probability P_i, independently of whether or not other subroutines are desired. Then, for example, the probability that no subroutines at all are needed is $(1 - P_1)(1 - P_2) \ldots (1 - P_N)$; and the probability that the search will end just after loading the ith subroutine is $P_i(1 - P_{i+1}) \ldots (1 - P_N)$. If L_i is the length of subroutine i, the average search time will therefore be essentially proportional to

$$L_1P_1(1 - P_2) \ldots (1 - P_N) + (L_1 + L_2)P_2(1 - P_3) \ldots (1 - P_N) + \cdots + (L_1 + \cdots + L_N)P_N.$$

What is the optimum arrangement of subroutines on the tape, under these assumptions?

16. [*M22*] (H. Riesel.) A programmer wants to test whether or not n given conditions are all simultaneously true. (For example, he may want to test whether both $x > 0$ and $y < z^2$, and it is not immediately clear which condition should be tested first.) Suppose that it costs T_i units of time to test condition i, and that the condition will be true with probability p_i, independent of the outcomes of all the other conditions. In which order should he make the tests?

17. [*M23*] (W. E. Smith.) Suppose you have to do n jobs; the ith job takes T_i units of time, and it has a *deadline* D_i. In other words, the ith job is supposed to be finished after at most D_i units of time have elapsed. What schedule for processing the jobs will minimize the *maximum tardiness*, i.e.,

$$\max (T_{a_1} - D_{a_1}, T_{a_1} + T_{a_2} - D_{a_2}, \ldots, T_{a_1} + T_{a_2} + \cdots + T_{a_n} - D_{a_n})?$$

18. [*M30*] (*Catenated search.*) Suppose N records are located in a linear array $R_1 \ldots R_N$, with probability p_i that record R_i will be sought. A search process is called "catenated" if each search begins where the last one left off. If consecutive searches are independent, the average time required will be $\sum_{1 \le i,j \le N} p_i p_j d(i, j)$, where $d(i, j)$ represents the amount of time to do a search that starts at position i and ends at position j. This model can be applied, for example, to disk file seek time, if $d(i, j)$ is the time needed to travel from cylinder i to cylinder j.

The object of this exercise is to characterize the optimum placement of records for catenated searches, whenever $d(i, j)$ is an increasing function of $|i - j|$, i.e., whenever we have $d(i, j) = d_{|i-j|}$ for $d_1 < d_2 < \cdots < d_{N-1}$. (The value of d_0 is irrelevant.) Prove that in this case the records are optimally placed, among all $N!$ permutations, if and only if either $p_1 \le p_N \le p_2 \le p_{N-1} \le \cdots \le p_{\lfloor N/2 \rfloor + 1}$ or $p_N \le p_1 \le p_{N-1} \le p_2 \le \cdots \le p_{\lfloor N/2 \rfloor}$. (Thus, an "organ pipe" arrangement of probabilities is best, as shown in Fig. 2.) *Hint:* Consider any arrangement where the respective probabilities are $q_1\, q_2 \ldots q_k\, s\, r_k \ldots r_2\, r_1\, t_1 \ldots t_m$, for some $m \ge 0$ and $k > 0$; $N = 2k + m + 1$. Show that the rearrangement $q'_1\, q'_2 \ldots q'_k\, s\, r'_k \ldots r'_2\, r'_1\, t_1 \ldots t_m$ is better, where $q'_i = \min(q_i, r_i)$ and $r'_i = \max(q_i, r_i)$, except when $q'_i = q_i$ and $r'_i = r_i$ for all i or when $q'_i = r_i$ and $r'_i = q_i$ and $t_j = 0$ for all i, j. The same holds when s is not present and $N = 2k + m$.

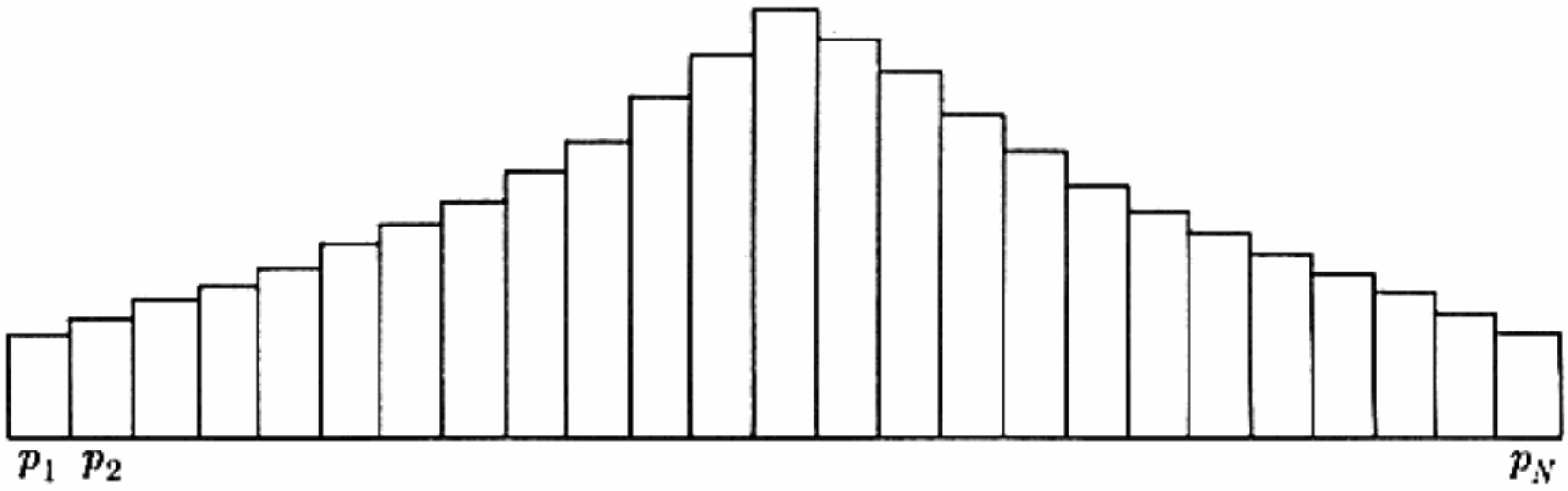


Fig. 2. An "organ-pipe arrangement" of probabilities minimizes the average seek time in a catenated search.

19. [*M20*] Continuing exercise 18, what are the optimal arrangements for catenated searches when the function $d(i, j)$ has the property that $d(i, j) + d(j, i) = c$ for all i, j? [This situation occurs, for example, on tapes without read-backwards capability, when we do not know the appropriate direction to search; for $i < j$ we have, say, $d(i, j) = a + b(L_{i+1} + \cdots + L_j)$ and $d(j, i) = a + b(L_{j+1} + \cdots + L_N) + r + b(L_1 + \cdots + L_i)$, where r is the rewind time.]

20. [*M28*] Continuing exercise 18, what are the optimal arrangements for catenated searches when the function $d(i, j) = \min(d_{|i-j|}, d_{n-|i-j|})$, for $d_1 < d_2 < \cdots$? [This situation occurs, for example, in a two-way linked circular list, or in a two-way shift-register storage device.]

21. [*M28*] Consider an n-dimensional cube whose vertices have coordinates $(d_n, \ldots, d_1)$ with $d_i = 0$ or 1; two vertices are called *adjacent* if they differ in exactly one coordinate. Suppose that a set of 2^n numbers $x_0 \le x_1 \le \cdots \le x_{2^n-1}$ is to be assigned to the 2^n vertices in such a way that $\sum|x_i - x_j|$ is minimized, where the sum is over all i and j such that x_i and x_j have been assigned to adjacent vertices. Prove that this minimum will be achieved if x_i is assigned to the vertex whose coordinates are the binary representation of i, for all i.

▸**22.** [*20*] Suppose you want to search a large file, not for equality but to find the 1000 records which are *closest* to a given key, in the sense that these 1000 records have the smallest values of $d(K_i, K)$ for some given distance function d. What data structure is most appropriate for such a sequential search?

6.2. SEARCHING BY COMPARISON OF KEYS

In this section we shall discuss search methods which are based on a linear ordering of the keys (e.g., alphabetic order or numeric order). After comparing the given argument K to a key K_i in the table, the search continues in three different ways, depending on whether $K < K_i$, $K = K_i$, or $K > K_i$. The sequential search methods of Section 6.1 were essentially limited to a two-way decision ($K = K_i$ vs. $K \neq K_i$), but if we free ourselves from the restriction of sequential access it becomes possible to make effective use of an order relation.

6.2.1. Searching an Ordered Table

What would you do if someone handed you a large telephone directory and told you to find the name of the man whose number is 795-6841? There is no better way to tackle this problem than to use the sequential methods of Section 6.1. (However, a clever private detective might try dialing the number and finding out who answers; or he might have a friend at the telephone company who has access to a special directory that is sorted by number instead of by name.) The point is that it is much easier to find an entry by the party's name, instead of by number, although the telephone directory contains all the information necessary in both cases. When a large file must be searched, sequential scanning is almost out of the question, but an ordering relation simplifies the job enormously.

With so many sorting methods at our disposal (Chapter 5), we will have little difficulty rearranging a file into order so that it may be searched conveniently. Of course, if we only need to search the table once, it is faster to do a sequential search than to do a complete sort of the file; but if we need to make repeated searches in the same file, we are better off having it in order. Therefore in this section we shall concentrate on methods which are appropriate for searching a table whose keys are in order,

$$K_1 < K_2 < \cdots < K_N,$$

making random accesses to the table entries. After comparing K to K_i in such a table, we either have

- $K < K_i$ [$R_i, R_{i+1}, \ldots, R_N$ are eliminated from consideration];

or

- $K = K_i$ [the search is done];

or

- $K > K_i$ [$R_1, R_2, \ldots, R_i$ are eliminated from consideration].

In each of these three cases, substantial progress has been made, unless i is near one of the ends of the table; this is why the ordering leads to an efficient algorithm.

Binary search. Perhaps the first such method which suggests itself is to start by comparing K to the middle key in the table; the result of this probe tells

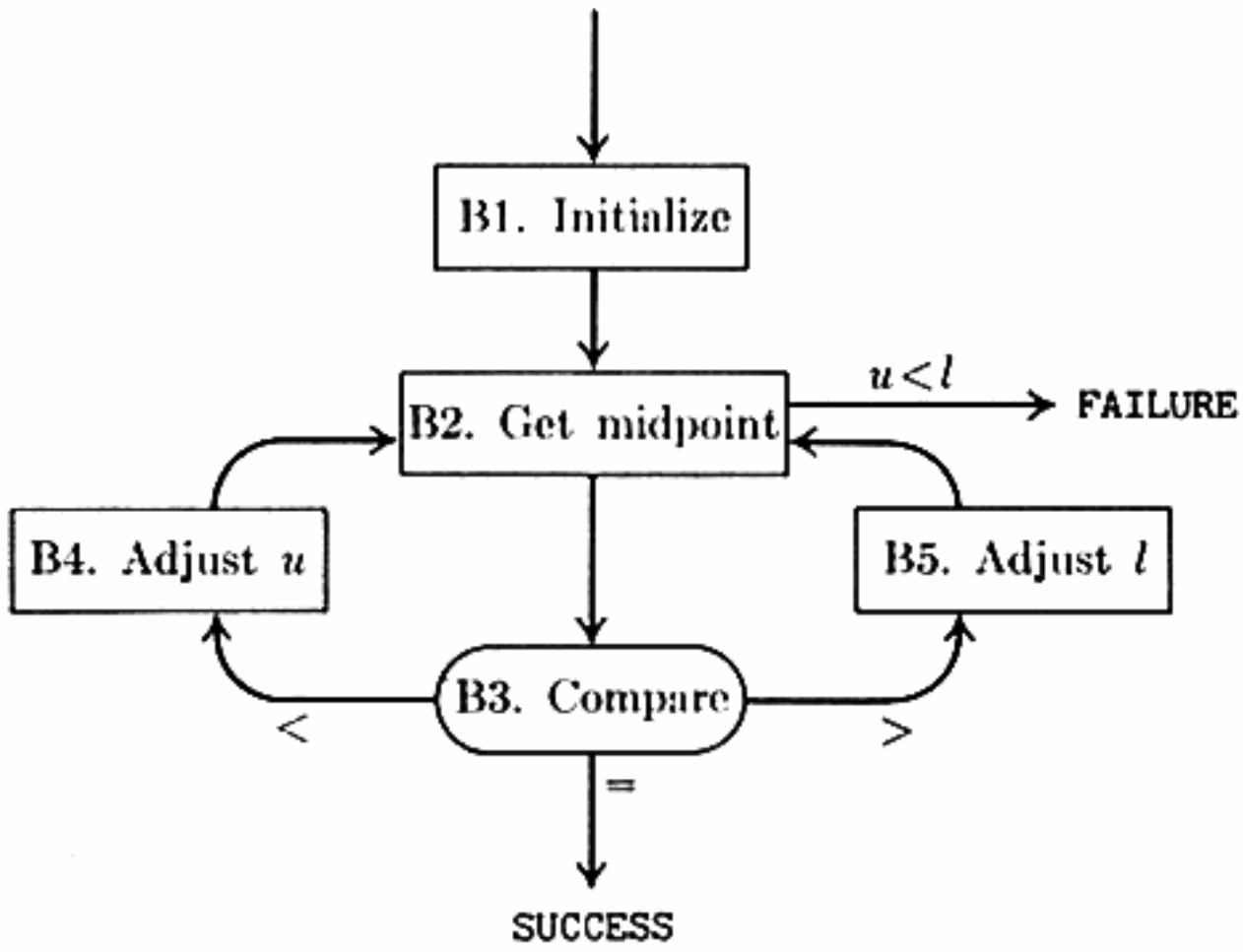


Fig. 3. Binary search.

which half of the table should be searched next, and the same procedure can be used again, comparing K to the middle key of the selected half, etc. After at most about $\log_2 N$ comparisons, we will have found the key or we will have established that it is not present. This procedure is sometimes known as "logarithmic search" or "bisection," but it is most commonly called *binary search*.

Although the basic idea of binary search is comparatively straightforward, the details can be somewhat tricky, and many good programmers have done it wrong the first few times they tried. One of the most popular correct forms of the algorithm makes use of two pointers, l and u, which indicate the current lower and upper limits for the search, as follows:

Algorithm B (*Binary search*). Given a table of records $R_1, R_2, \ldots, R_N$ whose keys are in increasing order $K_1 < K_2 < \cdots < K_N$, this algorithm searches for a given argument K.

B1. [Initialize.] Set $l \leftarrow 1$, $u \leftarrow N$.

B2. [Get midpoint.] (At this point we know that if K is in the table, it satisfies $K_l \le K \le K_u$. A more precise statement of the situation appears in exercise 1 below.) If $u < l$, the algorithm terminates unsuccessfully. Otherwise, set $i \leftarrow \lfloor (l + u)/2 \rfloor$, the approximate midpoint of the relevant table area.

B3. [Compare.] If $K < K_i$, go to B4; if $K > K_i$, go to B5; and if $K = K_i$, the algorithm terminates successfully.

B4. [Adjust u.] Set $u \leftarrow i - 1$ and return to B2.

B5. [Adjust l.] Set $l \leftarrow i + 1$ and return to B2. ∎

Figure 4 illustrates two cases of this binary search algorithm: first to search for the argument 653, which is present in the table, and then to search for 400,

a) Searching for 653 :

[061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908]
061 087 154 170 275 426 503 509 [512 612 653 677 703 765 897 908]
061 087 154 170 275 426 503 509 [512 612 653] 677 703 765 897 908
061 087 154 170 275 426 503 509 512 612 [653] 677 703 765 897 908

b) Searching for 400 :

[061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908]
[061 087 154 170 275 426 503] 509 512 612 653 677 703 765 897 908
061 087 154 170 [275 426 503] 509 512 612 653 677 703 765 897 908
061 087 154 170 [275] 426 503 509 512 612 653 677 703 765 897 908
061 087 154 170 275] [426 503 509 512 612 653 677 703 765 897 908

Fig. 4. Examples of binary search.

which is absent. The brackets indicate l and u, and the underlined key represents K_i. In both examples the search terminates after making four comparisons.

Program B (*Binary search*). As in the programs of Section 6.1, we assume here that K_i is a full-word key appearing in location KEY + i. The following code uses rI1 ≡ l, rI2 ≡ u, rI3 ≡ i.

```
01     START  ENT1  1          1          B1. Initialize. l ← 1.
02            ENT2  N          1          u ← N.
03            JMP   2F         1          To B2.
04     5H     JE    SUCCESS    C1         Jump if K = K_i.
05            ENT1  1,3        C1 − S     B5. Adjust l. l ← i + 1.
06     2H     ENTA  0,1        C + 1 − S  B2. Get midpoint.
07            INCA  0,2        C + 1 − S  rA ← l + u.
08            SRB   1          C + 1 − S  rA ← ⌊rA/2⌋.
09            STA   TEMP       C + 1 − S
10            CMP1  TEMP       C + 1 − S
11            JG    FAILURE    C + 1 − S  Jump if u < l.
12            LD3   TEMP       C          i ← midpoint.
13     3H     LDA   K          C          B3. Compare.
14            CMPA  KEY,3      C
15            JGE   5B         C          Jump if K ≥ K_i.
16            ENT2  -1,3       C2         B4. Adjust u. u ← i − 1.
17            JMP   2B         C2         To B2.
```

This procedure doesn't blend with MIX quite as "smoothly" as the other algorithms we have seen, because MIX does not allow much arithmetic in index registers. The running time is $(18C - 10S + 12)u$, where $C = C1 + C2$ is the number of comparisons made (the number of times step B3 is performed), and $S = 1$ or 0 depending on whether the outcome is successful or not. Note that line 08 of this program is "shift right binary 1," which is legitimate only on binary versions of MIX; for general byte size, this instruction should be replaced by "MUL =1//2+1=", increasing the running time to $(26C - 18S + 20)u$.

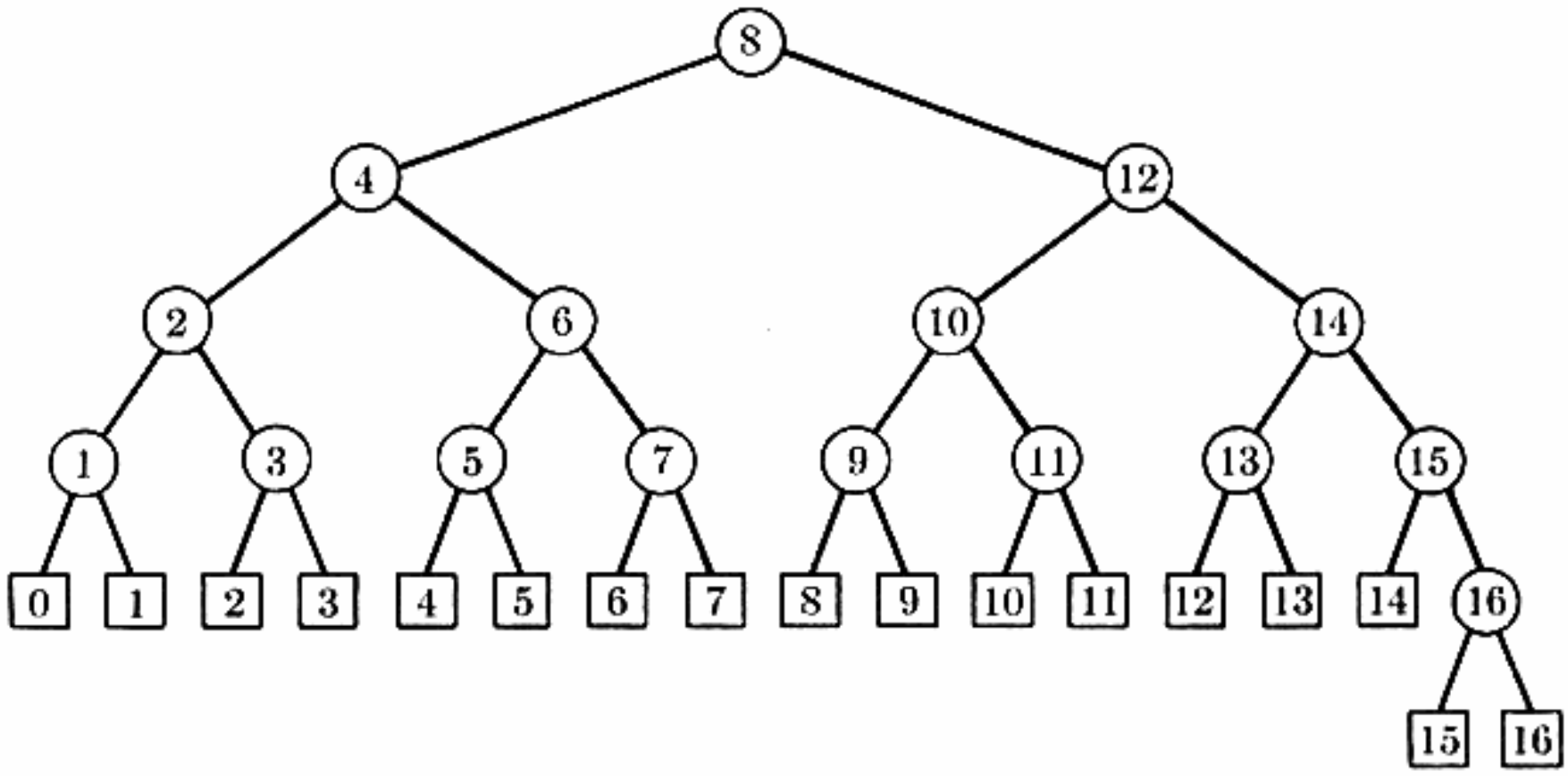


Fig. 5. A binary tree which corresponds to binary search when $N = 16$.

A tree representation. In order to really understand what is happening in Algorithm B, it is best to think of it as a binary decision tree, as shown in Fig. 5 for the case $N = 16$.

When N is 16, the first comparison made by the algorithm is $K:K_8$; this is represented by the root node (8) in the figure. Then if $K < K_8$, the algorithm follows the left subtree, comparing K to K_4; similarly if $K > K_8$, the right subtree is used. An unsuccessful search will lead to one of the "external" square nodes numbered $\boxed{0}$ through $\boxed{N}$; for example, we reach node $\boxed{6}$ if and only if $K_6 < K < K_7$.

The binary tree corresponding to a binary search on N records can be constructed as follows: If $N = 0$, the tree is simply $\boxed{0}$. Otherwise the root node is

$$\left(\lceil N/2 \rceil\right),$$

the left subtree is the corresponding binary tree with $\lceil N/2 \rceil - 1$ nodes, and the right subtree is the corresponding binary tree with $\lfloor N/2 \rfloor$ nodes and with all node numbers increased by $\lceil N/2 \rceil$.

In an analogous fashion, *any* algorithm for searching an ordered table of length N by means of comparisons can be represented as a binary tree in which the nodes are labelled with the numbers 1 to N (unless the algorithm makes redundant comparisons). Conversely, any binary tree corresponds to a valid method for searching an ordered table; we simply label the nodes

$$\boxed{0} \quad (1) \quad \boxed{1} \quad (2) \quad \boxed{2} \quad \ldots \quad \boxed{N-1} \quad (N) \quad \boxed{N} \qquad (1)$$

in symmetric order, from left to right.

If the search argument input to Algorithm B is K_{10}, the algorithm makes the comparisons $K > K_8$, $K < K_{12}$, $K = K_{10}$. This corresponds to the path from the root to (10) in Fig. 5. Similarly, the behavior of Algorithm B on

other keys corresponds to the other paths leading from the root of the tree. The method of constructing the binary trees corresponding to Algorithm B therefore makes it easy to prove the following result by induction on N:

Theorem B. If $2^{k-1} \le N < 2^k$, a successful search using Algorithm B requires (min 1, max k) comparisons. If $N = 2^k - 1$, an unsuccessful search requires k comparisons; and if $2^{k-1} \le N < 2^k - 1$, an unsuccessful search requires either $k - 1$ or k comparisons. ∎

Further analysis of binary search. (Nonmathematical readers, skip to Eq. 4.) The tree representation shows us also how to compute the *average* number of comparisons in a simple way. Let C_N be the average number of comparisons in a successful search, assuming that each of the N keys is an equally likely argument; and let C'_N be the average number of comparisons in an *un*successful search, assuming that each of the $N + 1$ intervals between keys is equally likely. Then we have

$$C_N = 1 + \frac{\text{internal path length of tree}}{N},$$

$$C'_N = \frac{\text{external path length of tree}}{N+1},$$

by the definition of internal and external path length. We have seen in Eq. 2.3.4.5–3 that the external path length is always $2N$ more than the internal path length; hence there is a rather unexpected relationship between C_N and C'_N:

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1. \tag{2}$$

This formula, which is due to Hibbard [*JACM* **9** (1962), 16–17], holds for all search methods which correspond to binary trees, i.e., for all methods which are based on nonredundant comparisons. The variance of C_N can also be expressed in terms of the variance of C'_N (see exercise 25).

From the above formulas we can see that the "best" way to search by comparisons is one whose tree has minimum external path length, over all binary trees with N internal nodes. Fortunately it can be proved that *Algorithm B is optimum* in this sense, for all N; for we have seen (exercise 5.3.1–20) that a binary tree has minimum path length if and only all its external nodes occur on at most two adjacent levels. It follows that the external path length of the tree corresponding to Algorithm B is

$$(N+1)(\lfloor \log_2 N \rfloor + 2) - 2^{\lfloor \log_2 N \rfloor + 1}. \tag{3}$$

(See Eq. 5.3.1–33.) From this formula and (2) we can compute the exact average number of comparisons, assuming that all search arguments are equally probable.

$N =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$C_N =$	1	$1\frac{1}{2}$	$1\frac{2}{3}$	2	$2\frac{1}{5}$	$2\frac{2}{6}$	$2\frac{3}{7}$	$2\frac{5}{8}$	$2\frac{7}{9}$	$2\frac{9}{10}$	3	$3\frac{1}{12}$	$3\frac{2}{13}$	$3\frac{3}{14}$	$3\frac{4}{15}$	$3\frac{6}{16}$
$C'_N =$	1	$1\frac{2}{3}$	2	$2\frac{2}{5}$	$2\frac{4}{6}$	$2\frac{6}{7}$	3	$3\frac{2}{9}$	$3\frac{4}{10}$	$3\frac{6}{11}$	$3\frac{8}{12}$	$3\frac{10}{13}$	$3\frac{12}{14}$	$3\frac{14}{15}$	4	$4\frac{2}{17}$

In general, if $k = \lfloor \log_2 N \rfloor$, we have (cf. Eq. 5.3.1–34)

$$\begin{aligned} C_N &= k + 1 - (2^{k+1} - k - 2)/N = \log_2 N - 1 + \epsilon + (k+2)/N, \\ C'_N &= k + 2 - 2^{k+1}/(N+1) \qquad\quad = \log_2 N + \epsilon', \end{aligned} \tag{4}$$

where $0 \le \epsilon, \epsilon' < 0.0861$.

To summarize: Algorithm B never makes more than $\lfloor \log_2 N \rfloor + 1$ comparisons, and it makes about $\log_2 N - 1$ comparisons in an average successful search. No search method based on comparisons can do better than this. The average running time of Program B is approximately

$$\begin{aligned} &(18 \log_2 N - 15)u \qquad \text{for a successful search,} \\ &(18 \log_2 N + 13)u \qquad \text{for an unsuccessful search,} \end{aligned} \tag{5}$$

if we assume that all outcomes of the search are equally likely.

An important variation. Instead of using three pointers l, i, and u in the search, it is tempting to use only two, the current position i and its rate of change, δ; after each unequal comparison, we could then set $i \leftarrow i \pm \delta$ and $\delta \leftarrow \delta/2$ (approximately). It is possible to do this, but only if extreme care is paid to the details, as in the following algorithm; simpler approaches are doomed to failure!

Algorithm U (*Uniform binary search*). Given a table of records $R_1, R_2, \ldots, R_N$ whose keys are in increasing order $K_1 < K_2 < \cdots < K_N$, this algorithm searches for a given argument K. If N is even, the algorithm will sometimes refer to a dummy key K_0 which should be set to $-\infty$ (or any value less than K). We assume that $N \ge 1$.

U1. [Initialize.] Set $i \leftarrow \lceil N/2 \rceil$, $m \leftarrow \lfloor N/2 \rfloor$.

U2. [Compare.] If $K < K_i$, go to U3; if $K > K_i$, go to U4; and if $K = K_i$, the algorithm terminates successfully.

U3. [Decrease i]. (We have pinpointed the search to an interval which contains either m or $m - 1$ records; i points just to the right of this interval.) If $m = 0$, the algorithm terminates unsuccessfully. Otherwise set $i \leftarrow i - \lceil m/2 \rceil$; then set $m \leftarrow \lfloor m/2 \rfloor$ and return to U2.

U4. [Increase i.] (We have pinpointed the search to an interval which contains either m or $m - 1$ records; i points just to the left of this interval.) If $m = 0$, the algorithm terminates unsuccessfully. Otherwise set $i \leftarrow i + \lceil m/2 \rceil$; then set $m \leftarrow \lfloor m/2 \rfloor$ and return to U2. ∎

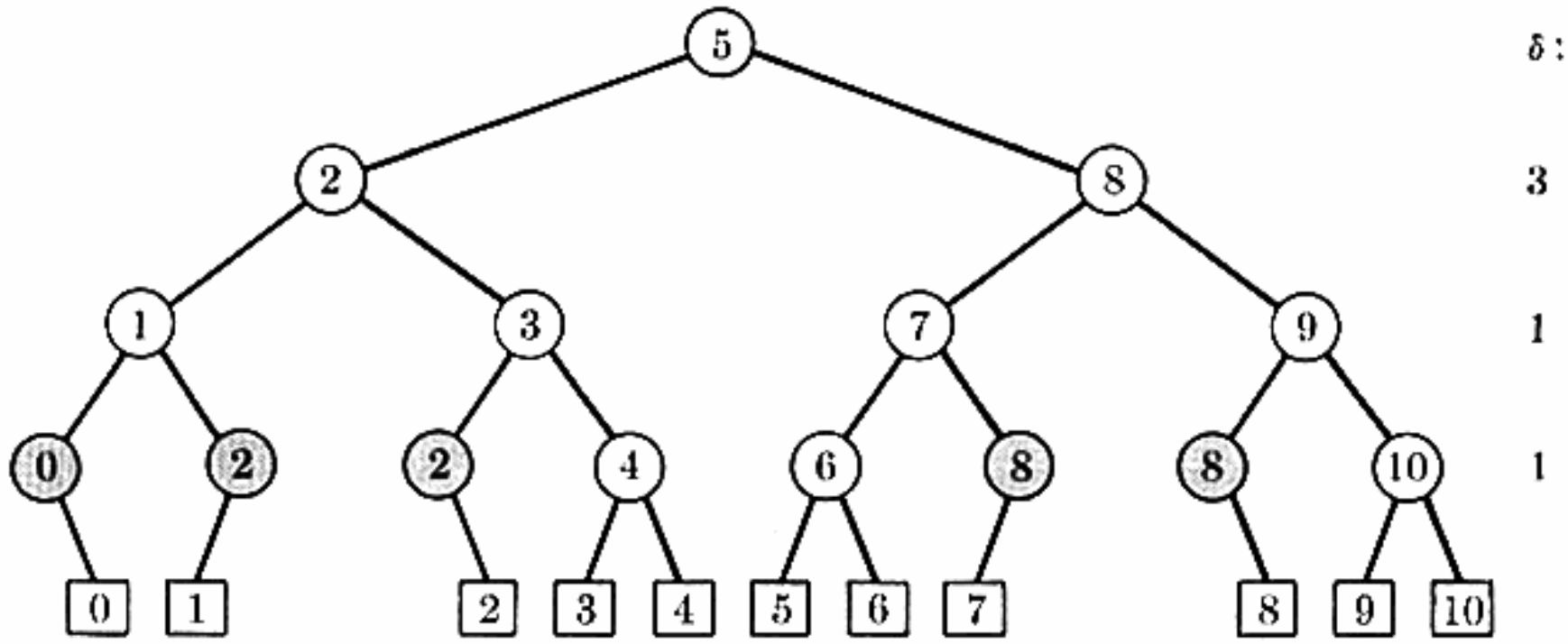


Fig. 6. The binary tree for a "uniform" binary search, when $N = 10$.

Figure 6 shows the corresponding binary tree for the search, when $N = 10$. In an unsuccessful search, the algorithm may make a redundant comparison just before termination; these nodes are shaded in the figure. We may call the search process *uniform* because the difference between the number of a node on level ℓ and the number of its ancestor on level $\ell - 1$ has a constant value δ for all nodes on level ℓ.

The theory underlying Algorithm U can be understood as follows: Suppose that we have an interval of length $n - 1$ to search; a comparison with the middle element (for n even) or with one of the two middle elements (for n odd) leaves us with two intervals of lengths $\lfloor n/2 \rfloor - 1$ and $\lceil n/2 \rceil - 1$. After repeating this process k times, we obtain 2^k intervals, of which the smallest has length $\lfloor n/2^k \rfloor - 1$ and the largest has length $\lceil n/2^k \rceil - 1$. Hence the lengths of two intervals at the same level differ by at most unity; this makes it possible to choose an appropriate "middle" element, without keeping track of the exact lengths.

The principal advantage of Algorithm U is that we need not maintain the value of m at all; we need only refer to a short table of the various δ to use at each level of the tree. Thus the algorithm reduces to the following procedure, which is equally good on binary or decimal computers:

Algorithm C (*Uniform binary search*). This algorithm is just like Algorithm U, but it uses an auxiliary table in place of the calculations involving m. The table entries are

$$\texttt{DELTA}[j] = \left\lfloor \frac{N + 2^{j-1}}{2^j} \right\rfloor = \left(\frac{N}{2^j}\right)\text{rounded}, \quad \text{for} \quad 1 \le j \le \lfloor \log_2 N \rfloor + 2. \tag{6}$$

C1. [Initialize.] Set $i \leftarrow \texttt{DELTA}[1]$, $j \leftarrow 2$.

C2. [Compare.] If $K < K_i$, go to C3; if $K > K_i$, go to C4; and if $K = K_i$, the algorithm terminates successfully.

C3. [Decrease i.] If $\texttt{DELTA}[j] = 0$, the algorithm terminates unsuccessfully. Otherwise, set $i \leftarrow i - \texttt{DELTA}[j]$, $j \leftarrow j + 1$, and go to C2.

C4. [Increase i.] If DELTA[j] $= 0$, the algorithm terminates unsuccessfully. Otherwise, set $i \leftarrow i +$ DELTA[j], $j \leftarrow j + 1$, and go to C2. ▮

Exercise 8 proves that this algorithm refers to the artificial key $K_0 = -\infty$ only when N is even.

Program C (*Uniform binary search*). This program does the same job as Program B, using Algorithm C with rA $\equiv K$, rI1 $\equiv i$, rI2 $\equiv j$, rI3 $\equiv$ DELTA[j].

```
01    START   ENT1  N+1/2      1          C1. Initialize.
02            ENT2  2          1          j ← 2.
03            LDA   K          1
04            JMP   2F         1
05    3H      JE    SUCCESS    C1         Jump if K = K_i.
06            J3Z   FAILURE    C1 − S     Jump if DELTA[j] = 0.
07            DEC1  0,3        C1 − S − A C3. Decrease i.
08    5H      INC2  1          C − 1      j ← j + 1.
09    2H      LD3   DELTA,2    C          C2. Compare.
10            CMPA  KEY,1      C
11            JLE   3B         C          Jump if K ≤ K_i.
12            INC1  0,3        C2         C4. Increase i.
13            J3NZ  5B         C2         Jump if DELTA[j] ≠ 0.
14    FAILURE EQU   *          1 − S      Exit if not in table. ▮
```

In a successful search, this algorithm corresponds to a binary tree with the same internal path length as the tree of Algorithm B, so the average number of comparisons C_N is the same as before. In an unsuccessful search, Algorithm C always makes exactly $\lfloor \log_2 N \rfloor + 1$ comparisons. The total running time of Program C is not quite symmetrical between left and right branches, since $C1$ is weighted more heavily than $C2$, but exercise 9 shows that we have $K < K_i$ roughly as often as $K > K_i$; hence Program C takes approximately

$$\begin{aligned} &(8.5 \log_2 N - 6)u && \text{for a successful search,} \\ &(8.5 \lfloor \log_2 N \rfloor + 12)u && \text{for an unsuccessful search.} \end{aligned} \tag{7}$$

This is more than twice as fast as Program B, without using any special properties of binary computers, even though the running times (5) for Program B assume that MIX has a "shift right binary" instruction.

Another modification of binary search, suggested in 1971 by L. E. Shar, will be still faster on some computers, because it is uniform after the first step, and it requires no table. The first step is to compare K with K_i, where $i = 2^k$, $k = \lfloor \log_2 N \rfloor$. If $K < K_i$, we use a uniform search with the δ's equal to 2^{k-1}, $2^{k-2}, \ldots, 1, 0$. On the other hand, if $K > K_i$ and $N > 2^k$, we reset i to $i' = N + 1 - 2^\ell$, where $\ell = \lfloor \log_2 (N - 2^k) \rfloor + 1$, and pretend that the first comparison was actually $K > K_{i'}$, using a uniform search with the δ's equal to $2^{\ell-1}, 2^{\ell-2}, \ldots, 1, 0$.

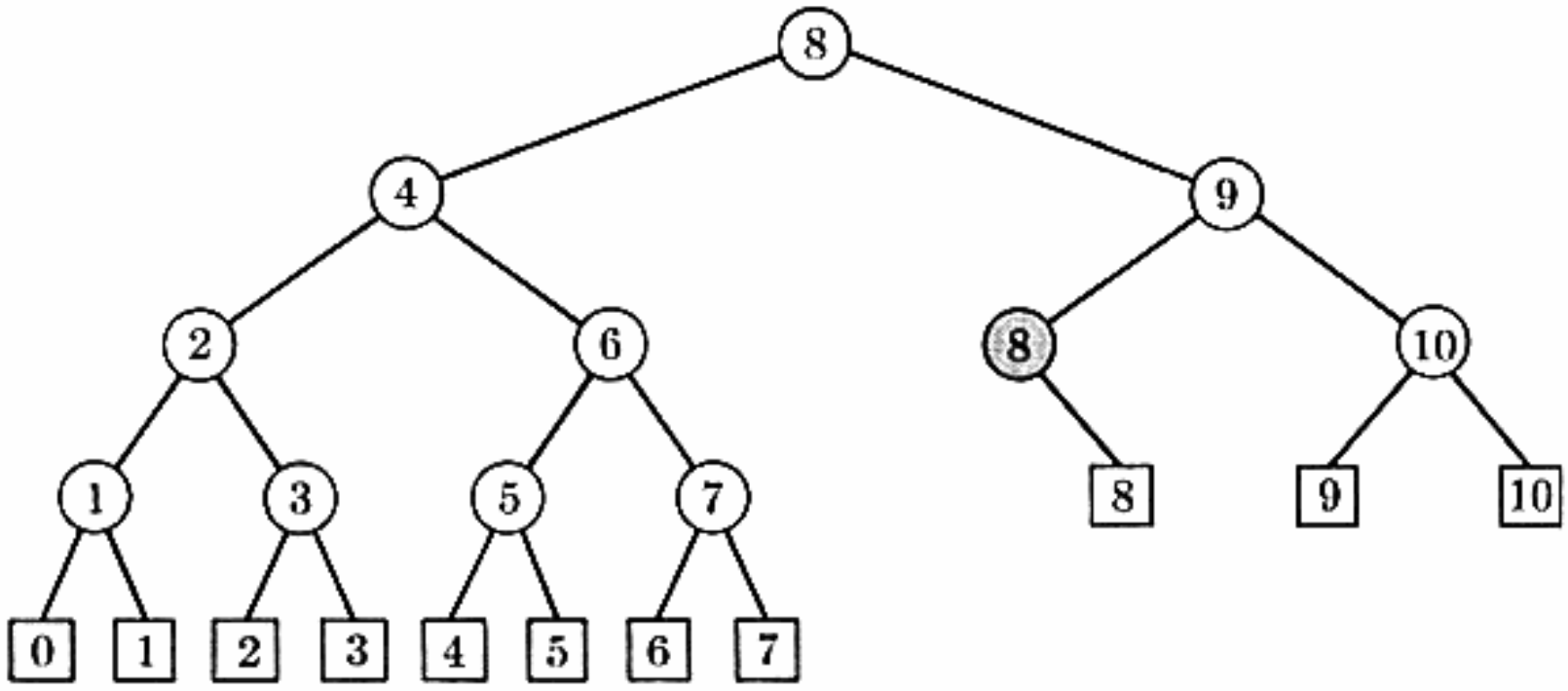


Fig. 7. The binary tree for Shar's almost uniform search, when $N = 10$.

Shar's method is illustrated for $N = 10$ in Fig. 7. Like the previous algorithms, it never makes more than $\lfloor \log_2 N \rfloor + 1$ comparisons; hence it is within one of the best possible average number of comparisons, in spite of the fact that it occasionally goes through several redundant steps in succession (cf. exercise 12).

Still another modification of binary search, which increases the speed of *all* the above methods when N is very large, is discussed in exercise 23. See also exercise 24 for a method that is faster yet!

Fibonaccian search. In the polyphase merge we have seen that the Fibonacci numbers can play a role analogous to the powers of 2. A similar phenomenon occurs in searching, where Fibonacci numbers provide us with an alternative to binary search. The resulting method is preferable on some computers, because it involves only addition and subtraction, not division by 2. The procedure we are about to discuss should be distinguished from an important numerical "Fibonacci search" procedure used to locate the maximum of a unimodal function [cf. *Fibonacci Quarterly* **4** (1966), 265–269]; the similarity of names has led to some confusion.

The Fibonaccian search technique looks very mysterious at first glance, if we simply take the program and try to explain what is happening; it seems to work by magic. But the mystery disappears as soon as the corresponding search tree is displayed. Therefore we shall begin our study of the method by looking at "Fibonacci trees."

Figure 8 shows the Fibonacci tree of order 6. Note that it looks somewhat more like a real-life shrub than the other trees we have been considering, perhaps because many natural processes satisfy a Fibonacci law. In general, the Fibonacci tree of order k has $F_{k+1} - 1$ internal (circular) nodes and F_{k+1} external (square) nodes, and it is constructed as follows:

If $k = 0$ or $k = 1$, the tree is simply $\boxed{0}$.

If $k \geq 2$, the root is $\textcircled{F_k}$; the left subtree is the Fibonacci tree of order $k - 1$; and the right subtree is the Fibonacci tree of order $k - 2$ with all numbers increased by F_k.

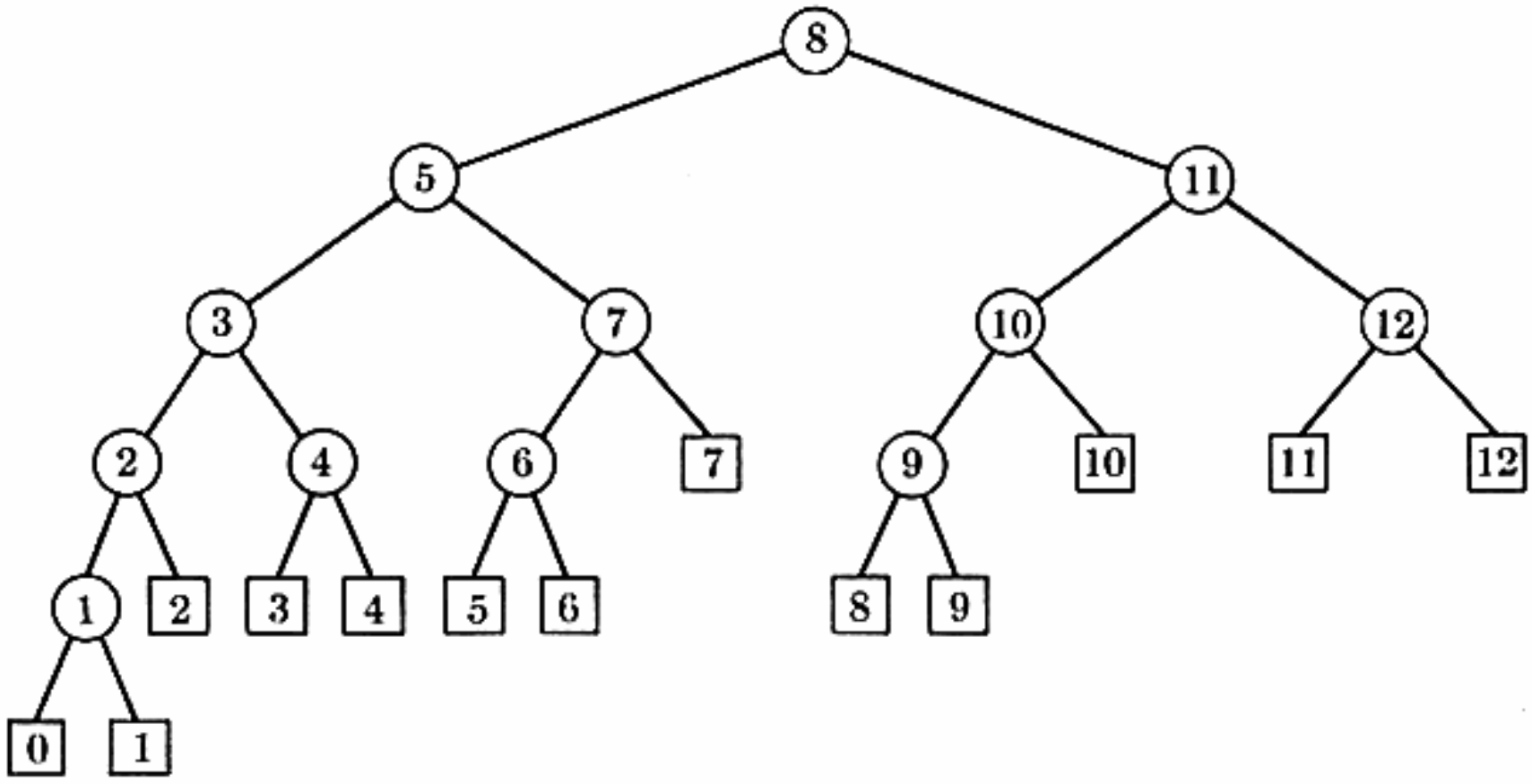


Fig. 8. The Fibonacci tree of order 6.

Note that, except for the external nodes, the numbers on the two sons of each internal node differ from their father's number by the same amount, and this amount is a Fibonacci number. Thus, $5 = 8 - F_4$ and $11 = 8 + F_4$ in Fig. 8. When the difference is F_j, the corresponding Fibonacci difference for the next branch on the left is F_{j-1}, while on the right it skips down to F_{j-2}. For example, $3 = 5 - F_3$ while $10 = 11 - F_2$.

If we combine these observations with an appropriate mechanism for recognizing the external nodes, we arrive at the following method:

Algorithm F (*Fibonaccian search*). Given a table of records $R_1, R_2, \ldots, R_N$ whose keys are in increasing order $K_1 < K_2 < \cdots < K_N$, this algorithm searches for a given argument K.

For convenience in description, this algorithm assumes that $N + 1$ is a perfect Fibonacci number, F_{k+1}. It is not difficult to make the method work for arbitrary N, if a suitable initialization is provided (see exercise 14).

F1. [Initialize.] Set $i \leftarrow F_k$, $p \leftarrow F_{k-1}$, $q \leftarrow F_{k-2}$. (Throughout the algorithm, p and q will be consecutive Fibonacci numbers.)

F2. [Compare.] If $K < K_i$, go to step F3; if $K > K_i$, go to F4; and if $K = K_i$, the algorithm terminates successfully.

F3. [Decrease i.] If $q = 0$, the algorithm terminates unsuccessfully. Otherwise set $i \leftarrow i - q$, and set $(p, q) \leftarrow (q, p - q)$; then return to F2.

F4. [Increase i.] If $p = 1$, the algorithm terminates unsuccessfully. Otherwise set $i \leftarrow i + q$, $p \leftarrow p - q$, then $q \leftarrow q - p$, and return to F2. ∎

The following MIX implementation gains speed by making two copies of the inner loop, one in which p is in rI2 and q in rI3, and one in which the registers are reversed; this simplifies step F3. In fact, the program actually keeps $p - 1$ and $q - 1$ in the registers, instead of p and q, in order to simplify the test "$p = 1$?" in step F4.

Program F (*Fibonaccian search*). rA $\equiv K$, rI1 $\equiv i$, (rI2, rI3) $\equiv p-1$, (rI3, rI2) $\equiv q-1$.

```
                    01  START  LDA   K                          1          F1. Initialize.
                    02         ENT1  F_k                        1          i ← F_k.
                    03         ENT2  F_{k-1}-1                  1          p ← F_{k-1}.
                    04         ENT3  F_{k-2}-1                  1          q ← F_{k-2}.
                    05         JMP   F2A                        1          To step F2.
06  F4A   INC1  1,3          18  F4B   INC1  1,2          C2 − S − A   F4. Increase i.  i ← i + q.
07        DEC2  1,3          19        DEC3  1,2          C2 − S − A   p ← p − q.
08        DEC3  1,2          20        DEC2  1,3          C2 − S − A   q ← q − p.
09  F2A   CMPA  KEY,1        21  F2B   CMPA  KEY,1             C       F2. Compare.
10        JL    F3A          22        JL    F3B              C       To F3 if K < K_i.
11        JE    SUCCESS      23        JE    SUCCESS          C2      Exit if K = K_i.
12        J2NZ  F4A          24        J3NZ  F4B            C2 − S    To F4 if p ≠ 1.
13        JMP   FAILURE      25        JMP   FAILURE          A       Exit if not in table.
14  F3A   DEC1  1,3          26  F3B   DEC1  1,2              C1      F3. Decrease i.  i ← i − q.
15        DEC2  1,3          27        DEC3  1,2              C1      p ← p − q.
16        J3NN  F2B          28        J2NN  F2A              C1      Swap registers if q > 0.
17        JMP   FAILURE      29        JMP   FAILURE      1 − S − A   Exit if not in table.  ∎
```

The running time of this program is analyzed in exercise 18. Figure 8 shows, and the analysis proves, that a left branch is taken somewhat more often than a right branch. Let C, $C1$, and $(C2 - S)$ be the respective number of times steps F2, F3, and F4 are performed. Then we have

$$\begin{aligned} C &= (\text{ave} \quad \phi k/\sqrt{5} + O(1), \quad \max k - 1), \\ C1 &= (\text{ave} \quad k/\sqrt{5} + O(1), \quad \max k - 1), \\ C2 - S &= (\text{ave } \phi^{-1}k/\sqrt{5} + O(1), \quad \max \lfloor k/2 \rfloor). \end{aligned} \tag{8}$$

Thus the left branch is taken about $\phi = 1.618$ times as often as the right branch (a fact which we might have guessed, since each probe divides the remaining interval into two parts, with the left part about ϕ times as large as the right). The total average running time of Program F therefore comes to approximately

$$\begin{aligned} (6\phi k/\sqrt{5} - (2 + 22\phi)/5)\ u &\approx (6.252 \log_2 N - 4.6)u \\ &\quad \text{for a successful search;} \\ (6\phi k/\sqrt{5} + (58/(27\phi))/5)\ u &\approx (6.252 \log_2 N + 5.8)u \\ &\quad \text{for an unsuccessful search.} \end{aligned} \tag{9}$$

This is slightly faster than Program C, although the worst case running time (roughly $8.6 \log_2 N$) is slightly slower.

Interpolation search. Let's forget computers for a moment, and consider how people actually carry out a search. Sometimes everyday life provides us with clues that lead to good algorithms.

Imagine yourself looking up a word in a dictionary. You probably *don't* begin by looking first at the middle page, then looking at the 1/4 or 3/4 point, etc., as in a binary search. It's even less likely that you use a Fibonaccian search!

If the word you want starts with the letter A, you probably begin near the front of the dictionary. In fact, many dictionaries have "thumb-indexes" which

show the starting page for the words beginning with a fixed letter. This thumb-index technique can readily be adapted to computers, and it will speed up the search; such algorithms are explored in Section 6.3.

Yet even after the initial point of search has been found, your actions still are not much like the methods we have discussed. If you notice that the desired word is alphabetically much greater than the words on the page being examined, you will turn over a fairly large chunk of pages before making the next reference. This is quite different from the above algorithms, which make no distinction between "much greater" and "slightly greater."

These considerations suggest an algorithm which might be called "interpolation search": When we know that K lies between K_l and K_u, we can choose the next probe to be about $(K - K_l)/(K_u - K_l)$ of the way between l and u, assuming that the keys are numeric and that they increase in a roughly constant manner throughout the interval.

Unfortunately, computer simulation experiments show that interpolation search does not decrease the number of comparisons enough to compensate for the extra computing time involved, when searching a table stored within a high-speed memory. It has been successful only to a limited extent when applied to *external* searching in peripheral memory devices. (Note that dictionary look-up by hand is essentially an external, not an internal, search.) We shall discuss external searching later.

History and bibliography. The earliest known example of a long list of items that was sorted into order to facilitate searching is the remarkable Babylonian reciprocal table of Inakibit-Anu, dating from about 200 B.C. This clay tablet is apparently the first of a series of three, which contained over 800 multiple-precision sexagesimal numbers and their reciprocals, sorted into lexicographic order. For example, the list included the following sequence of entries:

02 43 50 24	21 58 21 33 45
02 44 01 30	21 56 52 20 44 26 40
02 45 42 03 14 08	21 43 33 12 53 32 50 30 28 07 30
02 45 53 16 48	21 42 05
02 46 04 31 07 30	21 40 36 53 04 38 11 21 28 53 20

The task of sorting 800 entries like this, given the technology available at that time, must have been truly phenomenal. [See D. E. Knuth, *CACM* **15** (1972), 671–677, for further details.]

It is fairly natural to sort numerical values into order, but an order relation between letters or words does not suggest itself so readily. Yet a collating sequence for individual letters was present already in the most ancient alphabets. For example, many of the Biblical psalms have verses which follow a strict alphabetic sequence, the first verse starting with aleph, the second with beth, etc.; this was an aid to memory. Eventually the standard sequence of letters was used by Semitic and Greek peoples to denote numerals; for example, α, β, γ stood for 1, 2, 3, respectively.

But the use of alphabetic order for entire words seems to be a much later invention; it is something we might think is obvious, yet it has to be taught to children, and at some point in history it was necessary to teach it to adults! Several lists from about 300 B.C. have been found on the Aegean Islands, giving the names of people in certain religious cults; these lists have been alphabetized, but only by the first letter, thus representing only the first pass of a left-to-right radix sort. Some Greek papyri from the years 134–135 A.D. contain fragments of ledgers which show the names of taxpayers alphabetized by the first two letters. Apollonius Sophista used alphabetic order on the first two letters, and often on subsequent letters, in his lengthy concordance of Homer's poetry (first century A.D.). A few examples of more perfect alphabetization are known, notably Galen's *Hippocratic Glosses* (c. 200 A.D.), but these were very rare. Thus, words were arranged by their first letter only, in the *Etymologiarum* of St. Isidorus (c. 630 A.D., book x); and the *Corpus Glossary* (c. 725) used only the first two letters of each word. The latter two works were perhaps the largest nonnumerical files of data to be compiled during the Middle Ages.

It is not until Giovanni di Genoa's *Catholicon* (1286) that we find a specific description of true alphabetical order. In his preface, Giovanni explained that

amo	precedes	*bibo*
abeo	precedes	*adeo*
amatus	precedes	*amor*
imprudens	precedes	*impudens*
iusticia	precedes	*iustus*
polisintheton	precedes	*polissenus*

(thereby giving examples of situations in which the ordering is determined by the 1st, 2nd, . . . , 6th letters), "and so in like manner." He remarked that strenuous effort was required to devise these rules. "I beg of you, therefore, good reader, do not scorn this great labor of mine and this order as something worthless."

A detailed study of the development of alphabetic order, up to the time printing was invented, has been made by Lloyd W. Daly, *Collection Latomus* **90** (1967), 100 pp. He found some interesting old manuscripts that were evidently used as worksheets while sorting words by their first letters (see pp. 87–90 of his monograph).

The first dictionary of English, Robert Cawdrey's *Table Alphabeticall* (London, 1604), contains the following instructions:

> Nowe if the word, which thou art desirous to finde, beginne with (a) then looke in the beginning of this Table, but if with (v) looke towards the end. Againe, if thy word beginne with (ca) looke in the beginning of the letter (c) but if with (cu) then looke toward the end of that letter. And so of all the rest. &c.

It is interesting to note that Cawdrey was teaching *himself* how to alphabetize

as he prepared his dictionary; numerous misplaced words appear on the first few pages, but the last part is in nearly perfect alphabetical order!

Binary search was first mentioned by John Mauchly, in what was perhaps the first published discussion of nonnumerical programming methods [*Theory and techniques for the design of electronic digital computers*, ed. by G. W. Patterson, **3** (1946), 22.8–22.9]. The method became "well known" during the 50's, but nobody seems to have worked out the details of what should be done when N does not have the special form $2^n - 1$. [See A. D. Booth, *Nature* **176** (1955), 565; A. I. Dumey, *Computers and Automation* **5** (December, 1956), 7, where binary search is called "Twenty Questions"; Daniel D. McCracken, *Digital Computer Programming* (Wiley, 1957), 201–203; and M. Halpern, *CACM* **1** (February, 1958), 1–3.]

H. Bottenbruch [*JACM* **9** (1962), 214] was apparently the first to publish a binary search algorithm which works for all N. He presented an interesting variation of Algorithm B which avoids a separate test for equality until the very end: Using $i \leftarrow \lceil (l+u)/2 \rceil$ instead of $\lfloor (l+u)/2 \rfloor$ in step B2, he set $l \leftarrow i$ whenever $K \geq K_i$; then $u - l$ decreases at every step. Eventually, when $l = u$, we have $K_l \leq K < K_{l+1}$, and we can test whether or not the search was successful by making one more comparison. (He assumed that $K \geq K_1$ initially.) This idea speeds up the inner loop slightly on many computers, and the same principle can be used with all of the algorithms we have discussed in this section; but the change is desirable only for large N (see exercise 23).

K. E. Iverson [*A Programming Language* (Wiley, 1962), 141] gave the procedure of Algorithm B, but without considering the possibility of an unsuccessful search. D. E. Knuth [*CACM* **6** (1963), 556–558] presented Algorithm B as an example used with an automated flowcharting system. The uniform binary search, Algorithm C, was suggested to the author by A. K. Chandra of Stanford University in 1971.

Fibonaccian searching was invented by David E. Ferguson [*CACM* **3** (1960), 648], but his flowchart and analysis were incorrect. The Fibonacci tree (without labels) had appeared many years earlier, as a curiosity in the first edition of Hugo Steinhaus's popular book *Mathematical Snapshots* (New York: Stechert, 1938), p. 28; he drew it upside down and made it look like a real tree, with right branches twice as long as left branches so that all the leaves occur at the same level.

Interpolation searching was suggested by W. W. Peterson [*IBM J. Res. & Devel.* **1** (1957), 131–132]; he gave a theoretical estimate for the average number of comparisons needed if the keys are randomly selected from a uniform distribution, but the estimate does not seem to agree with actual simulation experiments.

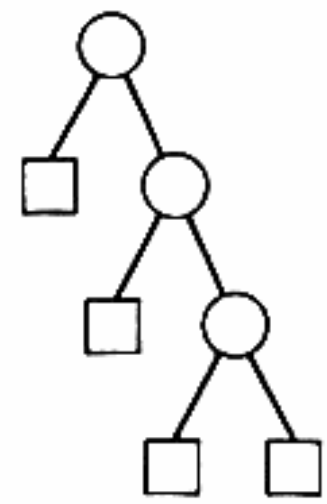
EXERCISES

▶ **1.** [*21*] Prove that if $u < l$ in step B2 of the binary search, we have $u = l - 1$ and

$K_u < K < K_l$. (Assume by convention that $K_0 = -\infty$ and $K_{N+1} = +\infty$, although these artificial keys are never really used by the algorithm so they need not be present in the actual table.)

▸ **2.** [*22*] Would Algorithm B still work properly if we (a) changed step B5 to "$l \leftarrow i$" instead of "$l \leftarrow i + 1$"? (b) changed step B4 to "$u \leftarrow i$" instead of "$u \leftarrow i - 1$"? (c) made both of these changes?

3. [*15*] What searching method corresponds to the tree



?

What is the average number of comparisons made in a successful search? in an unsuccessful search?

4. [*20*] If a search using Program 6.1S (sequential search) takes exactly 638 units of time, how long does it take with Program B (binary search)?

5. [*M24*] For what values of N is Program B actually *slower* than a sequential search (Program 6.1Q′) on the average, assuming that the search is successful?

6. [*28*] (K. E. Iverson.) Exercise 5 suggests that it would be best to have a "hybrid" method, changing from binary search to sequential search when the remaining interval has length less than some judiciously chosen value. Write an efficient MIX program for such a search and determine the best changeover value.

▸ **7.** [*M22*] Would Algorithm U still work properly if we changed step U1 so that (a) both i and m are set equal to $\lfloor N/2 \rfloor$? (b) both i and m are set equal to $\lceil N/2 \rceil$? [*Hint:* Suppose the first step were "Set $i \leftarrow 0$, $m \leftarrow N$ (or $N + 1$), go to U4."]

8. [*M20*] (a) What is the sum $\sum_{0 \le j \le \lfloor \log_2 N \rfloor + 2}$ DELTA[j] of the increments in Algorithm C? (b) What are the minimum and maximum values of i which can occur in step C2?

9. [*M26*] Find exact formulas for the average values of $C1$, $C2$, and A in the frequency analysis of Program C, as a function of N and S.

10. [*20*] Is there any value of $N > 1$ for which Algorithms B and C are exactly equivalent, in the sense that they will both perform the same sequence of comparisons for all search arguments?

11. [*21*] Explain how to write a MIX program for Algorithm C containing approximately $7 \log_2 N$ instructions and having a running time of about $4.5 \log_2 N$ units.

12. [*20*] Draw the binary search tree corresponding to Shar's method when $N = 12$.

13. [*M24*] Tabulate the average number of comparisons made by Shar's method, for $1 \le N \le 16$, considering both successful and unsuccessful searches.

14. [*21*] Explain how to extend Algorithm F so that it will apply for all $N \ge 1$.

15. [*21*] Figure 9 shows the lineal chart of the rabbits in Fibonacci's original rabbit problem (cf. Section 1.2.8). Is there a simple relationship between this and the Fibonacci tree discussed in the text?

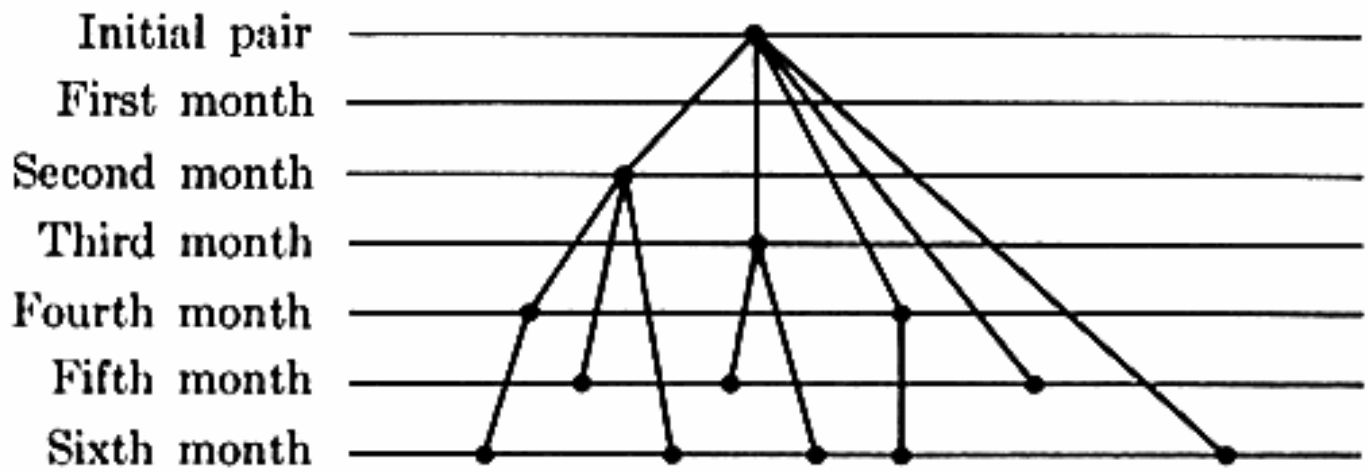


Fig. 9. Pairs of rabbits breeding by Fibonacci's rule.

16. [*M19*] For what values of k does the Fibonacci tree of order k define an optimal search procedure, in the sense that the fewest comparisons are made on the average?

17. [*M21*] From exercise 1.2.8–34 (or exercise 5.4.2–10) we know that every positive integer n has a unique representation as a sum of Fibonacci numbers $n = F_{a_1} + F_{a_2} + \cdots + F_{a_r}$, where $r \geq 1$, $a_j \geq a_{j+1} + 2$ for $1 \leq j < r$, and $a_r \geq 2$. Prove that in the Fibonacci tree of order k, the path from the root to node $\boxed{n}$ has length $k + 1 - r - a_r$.

18. [*M30*] Find exact formulas for the average values of $C1$, $C2$, and A in the frequency analysis of Program F, as a function of k, F_k, F_{k+1}, and S.

19. [*M42*] Carry out a detailed analysis of the average running time of the algorithm suggested in exercise 14.

20. [*M22*] The number of comparisons required in a binary search is approximately $\log_2 N$, and in the Fibonaccian search it is roughly $(\phi/\sqrt{5}) \log_\phi N$. The purpose of this exercise is to show that these formulas are special cases of a more general result.

Let p and q be positive numbers with $p + q = 1$. Consider a search algorithm which, given a table of N numbers in increasing order, starts by comparing the argument with the (pN)th key, and iterates this procedure on the smaller blocks. (The binary search has $p = q = 1/2$; the Fibonacci search has $p = 1/\phi$, $q = 1/\phi^2$.)

If $C(N)$ denotes the average number of comparisons required to search a table of size N, it approximately satisfies the relations

$$C(1) = 0; \qquad C(N) = 1 + pC(pN) + qC(qN) \qquad \text{for} \qquad N > 1.$$

This happens because there is probability p (roughly) that the search reduces to a pN-element search, and probability q that it reduces to a qN-element search, after the first comparison. When N is large, we may ignore the small-order effect caused by the fact that pN and qN aren't exactly integers.

a) Show that $C(N) = \log_b N$ satisfies these relations exactly, for a certain choice of b. For binary and Fibonaccian search, this value of b agrees with the formulas derived earlier.

b) A man argues as follows: "With probability p, the size of the interval being scanned in this algorithm is divided by $1/p$; with probability q, the interval size is divided by $1/q$. Therefore the interval is divided by $p \cdot (1/p) + q \cdot (1/q) = 2$ on the average, so the algorithm is exactly as good as the binary search, regardless of p and q." Is there anything wrong with his argument?

21. [*20*] Draw the binary tree corresponding to interpolation search when $N = 10$.

22. [*M47*] Derive formulas which properly estimate the average number of iterations needed in the interpolation search applied to random data.

▶**23.** [*25*] The binary search algorithm of H. Bottenbruch, mentioned at the close of this section, avoids testing for equality until the very end of the search. (During the algorithm we know that $K_l \le K < K_{u+1}$, and the case of equality is not examined until $l = u$.) Such a trick would make Program B run a little bit faster for large N, since the "JE" instruction could be removed from the inner loop. (However, the idea wouldn't really be practical since $\log_2 N$ is usually small; we would need $N > 2^{36}$ in order to compensate for the extra iteration necessary!)

Show that *every* search algorithm corresponding to a binary tree can be adapted to a search algorithm that uses two-way branching ($<$ vs. $\ge$) at the internal nodes of the tree, in place of the three-way branching ($<$, $=$, or $>$) used in the text's discussion. In particular, show how to modify Algorithm C in this way.

▶**24.** [*23*] The complete binary tree is a convenient way to represent a minimum-path-length tree in consecutive locations. (Cf. Section 2.3.4.5.) Devise an efficient search method based on this representation. [*Hint:* Is it possible to use multiplication by 2 instead of division by 2 in a binary search?]

▶**25.** [*M25*] Suppose that a binary tree has a_k internal nodes and b_k external nodes on level k, for $k = 0, 1, \ldots$. (The root is at level zero.) Thus in Fig. 8 we have $(a_0, a_1, \ldots, a_6) = (1, 2, 4, 4, 1, 0)$ and $(b_0, b_1, \ldots, b_6) = (0, 0, 0, 4, 7, 2)$. (a) Show that there is a simple algebraic relationship which connects the generating functions $A(z) = \sum_k a_k z^k$ and $B(z) = \sum_k b_k z^k$. (b) The probability distribution for a successful search in a binary tree has the generating function $g(z) = zA(z)/N$, and for an unsuccessful search the generating function is $h(z) = B(z)/(N+1)$. (Thus in the text's notation we have $C_N = \text{mean}(g)$, $C'_N = \text{mean}(h)$, and Eq. (2) gives a relation between these quantities.) Find a relation between $\text{var}(g)$ and $\text{var}(h)$.

26. [*22*] Show that the Fibonacci tree is related to polyphase merge sorting on three tapes.

27. [*M30*] (H. S. Stone and John Linn.) Consider a search process which uses k processors simultaneously, and which is based solely on comparisons of keys. Thus at every step of the search, k indices $i_1, \ldots, i_k$ are specified, and we perform k simultaneous comparisons; if $K = K_{i_j}$ for some j, the search terminates successfully, otherwise the search proceeds to the next step based on the 2^k possible outcomes $K < K_{i_j}$ or $K > K_{i_j}$, $1 \le j \le n$.

Prove that such a process must always take at least approximately $\log_{k+1} N$ steps on the average, as $N \to \infty$, assuming that each key of the table is equally likely as a search argument. (Hence the potential increase in speed over 1-processor binary search is only a factor of $\log_2 (k+1)$, not the factor of k we might expect. In this sense it is more efficient to assign each processor to a different, independent search problem, instead of making them cooperate on a single search.)

6.2.2. Binary Tree Searching

In the preceding section, we learned that an implicit binary tree structure makes it easier to understand the behavior of binary search and Fibonaccian search. For a given value of N, the tree corresponding to binary search achieves

the theoretical minimum number of comparisons that are necessary to search a table by means of key comparisons. But the methods of the preceding section are appropriate mainly for fixed-size tables, since the sequential allocation of records makes insertions and deletions rather expensive. If the table is dynamically changing, we might spend more time maintaining it than we save in binary-searching it.

The use of an *explicit* binary tree structure makes it possible to insert and delete records quickly, as well as to search the table efficiently. As a result, we essentially have a method which is useful both for searching and for sorting. This gain in flexibility is achieved by adding two link fields to each record of the table.

Techniques for searching a growing table are often called *symbol table algorithms*, because assemblers and compilers and other system routines generally use such methods to keep track of user-defined symbols. For example, the key of each record within a compiler might be a symbolic identifier denoting a variable in some FORTRAN or ALGOL program, and the rest of that record might contain information about the type of that variable and its storage allocation. Or the key might be a symbol in a MIXAL program, with the rest of the record containing the equivalent of that symbol. The tree search and insertion routines to be described in this section are quite efficient for use as symbol table algorithms, especially in applications where it is desirable to print out a list of the symbols in alphabetic order. Other symbol table algorithms are described in Sections 6.3 and 6.4.

Figure 10 shows a binary search tree containing the names of eleven signs

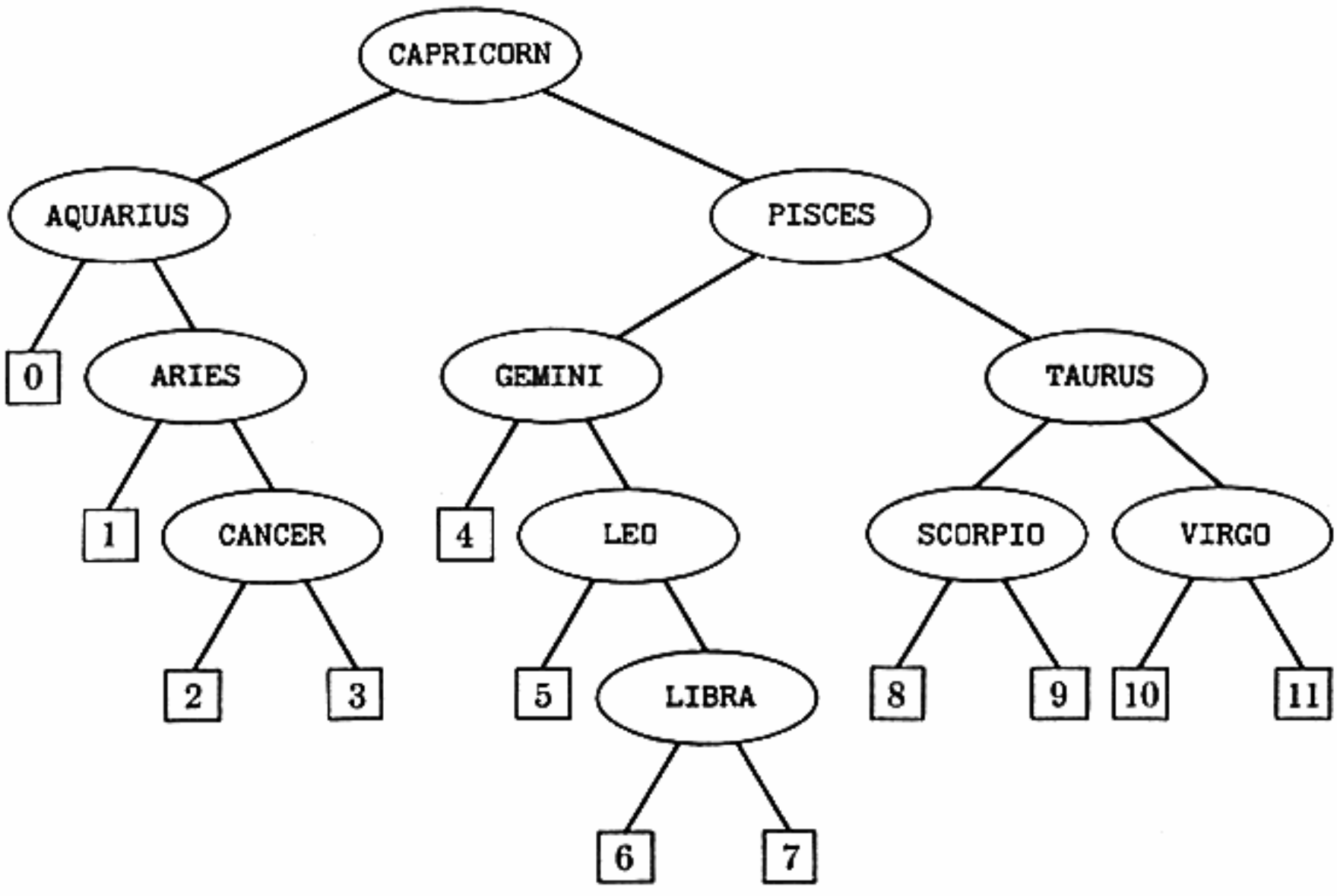


Fig. 10. A binary search tree.

of the zodiac. If we now search for the twelfth name, SAGITTARIUS, starting at the root or apex of the tree, we find it is greater than CAPRICORN, so we move to the right; it is greater than PISCES, so we move right again; it is less than TAURUS, so we move left; and it is less than SCORPIO, so we arrive at external node $\boxed{8}$. The search was unsuccessful; we can now *insert* SAGITTARIUS at the place the search ended, by linking it into the tree in place of the external node $\boxed{8}$. In this way the table can grow without the necessity of moving any of the existing records. Figure 10 was formed by starting with an empty tree and successively inserting the keys CAPRICORN, AQUARIUS, PISCES, ARIES, TAURUS, GEMINI, CANCER, LEO, VIRGO, LIBRA, SCORPIO, in this order.

All of the keys in the left subtree of the root in Fig. 10 are alphabetically less than CAPRICORN, and all keys in the right subtree are alphabetically greater. A similar statement holds for the left and right subtrees of every node. It follows that the keys appear in strict alphabetic sequence from left to right,

AQUARIUS, ARIES, CANCER, CAPRICORN, GEMINI, LEO, . . . , VIRGO

if we traverse the tree in *symmetric order* (cf. Section 2.3.1), since symmetric order is based on traversing the left subtree of each node just before that node, then traversing the right subtree.

The following algorithm spells out the searching and insertion processes in detail.

Algorithm T (*Tree search and insertion*). Given a table of records which form a binary tree as described above, this algorithm searches for a given argument K. If K is not in the table, a new node containing K is inserted into the tree in the appropriate place.

The nodes of the tree are assumed to contain at least the following fields:

KEY(P) = key stored in NODE(P);
LLINK(P) = pointer to left subtree of NODE(P);
RLINK(P) = pointer to right subtree of NODE(P).

Null subtrees (the external nodes in Fig. 10) are represented by the null pointer Λ. The variable ROOT points to the root of the tree. For convenience, we assume that the tree is not empty (i.e., ROOT $\neq \Lambda$).

T1. [Initialize.] Set P $\leftarrow$ ROOT. (The pointer variable P will move down the tree.)

T2. [Compare.] If $K <$ KEY(P), go to T3; if $K >$ KEY(P), go to T4; and if $K =$ KEY(P), the search terminates successfully.

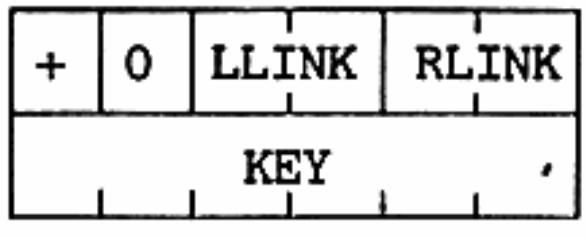
T3. [Move left.] If LLINK(P) $\neq \Lambda$, set P $\leftarrow$ LLINK(P) and go back to T2. Otherwise go to T5.

T4. [Move right.] If RLINK(P) $\neq \Lambda$, set P $\leftarrow$ RLINK(P) and go back to T2.

T5. [Insert into tree.] (The search is unsuccessful; we will now put K into the

tree.) Set Q ⇐ AVAIL, the address of a new node. Set KEY(Q) ← K, LLINK(Q) ← RLINK(Q) ← Λ. (In practice, other fields of the new node should also be initialized.) If K was less than KEY(P), set LLINK(P) ← Q, otherwise set RLINK(P) ← Q. (At this point we could set P ← Q and terminate the algorithm successfully.) ∎

This algorithm lends itself to a convenient machine language implementation. We may assume, for example, that the tree nodes have the form



(1)

followed perhaps by additional words of INFO. Using an AVAIL list for the free storage pool, as in Chapter 2, we can write the following MIX program:

Program T (*Tree search and insertion*). rA ≡ K, rI1 ≡ P, rI2 ≡ Q.

```
01    LLINK   EQU   2:3
02    RLINK   EQU   4:5
03    START   LDA   K                 1          T1. Initialize.
04            LD1   ROOT              1          P ← ROOT.
05            JMP   2F                1
06    4H      LD2   0,1(RLINK)        C2         T4. Move right.  Q ← RLINK(P).
07            J2Z   5F                C2         To T5 if Q = Λ.
08    1H      ENT1  0,2               C − 1      P ← Q.
09    2H      CMPA  1,1               C          T2. Compare.
10            JG    4B                C          To T4 if K > KEY(P).
11            JE    SUCCESS           C1         Exit if K = KEY(P).
12            LD2   0,1(LLINK)        C1 − S     T3. Move left.  Q ← LLINK(P).
13            J2NZ  1B                C1 − S     To T2 if Q ≠ Λ.
14    5H      LD2   AVAIL             1 − S      T5. Insert into tree.
15            J2Z   OVERFLOW          1 − S
16            LDX   0,2(RLINK)        1 − S
17            STX   AVAIL             1 − S      Q ⇐ AVAIL.
18            STA   1,2               1 − S      KEY(Q) ← K.
19            STZ   0,2               1 − S      LLINK(Q) ← RLINK(Q) ← Λ.
20            JL    1F                1 − S      Was K < KEY(P)?
21            ST2   0,1(RLINK)          A        RLINK(P) ← Q.
22            JMP   *+2                 A
23    1H      ST2   0,1(LLINK)        1 − S − A  LLINK(P) ← Q.
24    DONE    EQU   *                 1 − S      Exit after insertion. ∎
```

The first 13 lines of this program do the search; the last 11 lines do the insertion. The running time for the searching phase is $(7C + C1 - 3S + 4)u$, where

C = number of comparisons made;
$C1$ = number of times $K \le$ KEY(P);
S = 1 if the search is successful, 0 otherwise.

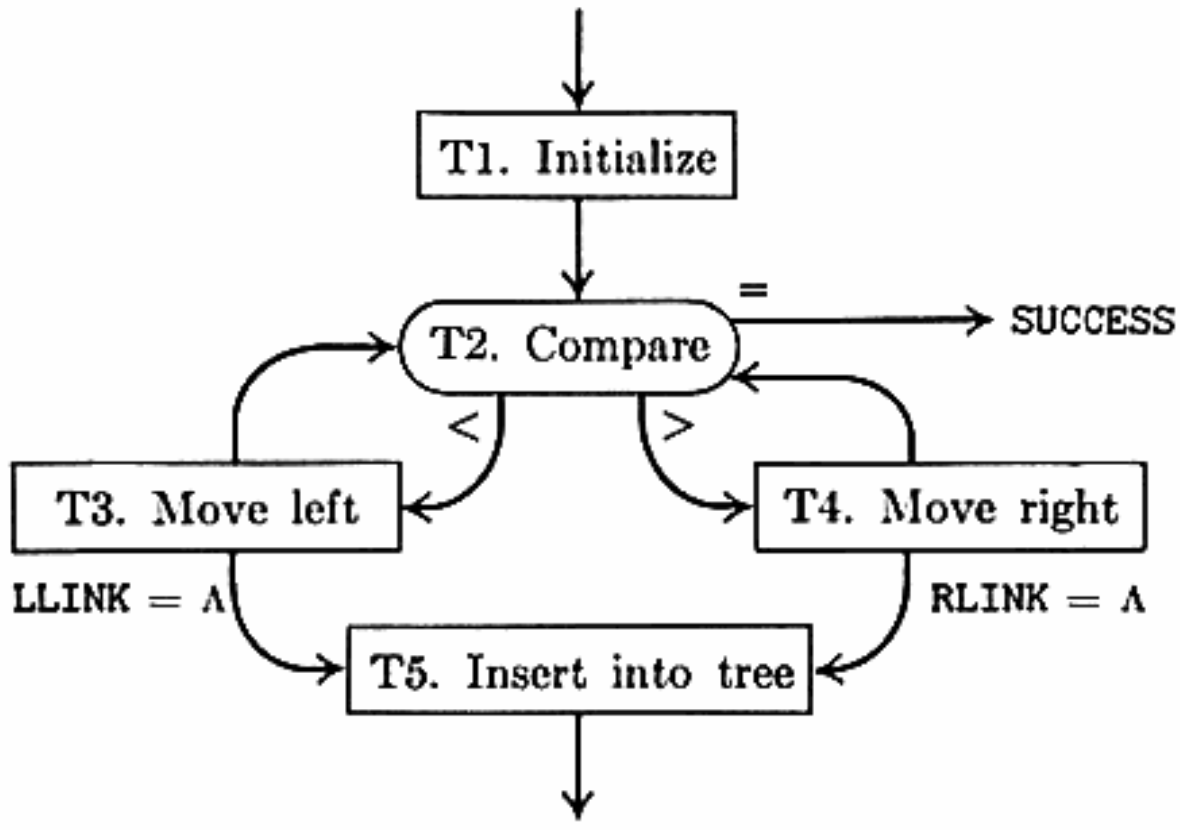


Fig. 11. Tree search and insertion.

On the average we have $C1 = \frac{1}{2}(C + S)$, since $C1 + C2 = C$ and $C1 - S \approx C2$; so the running time is about $(7.5C - 2.5S + 4)u$. This compares favorably with the binary search algorithms which use an implicit tree (cf. Program 6.2.1C). By duplicating the code as in Program 6.2.1F we could eliminate line 08 of Program T, reducing the running time to $(6.5C - 2.5S + 5)u$. If the search is unsuccessful, the insertion phase of the program costs an extra $14u$ or $15u$.

Algorithm T can be conveniently adapted to *variable-length keys* and variable-length records. For example, if we allocate the available space sequentially, in a last-in-first-out manner, we can easily create nodes of varying size; the first word of (1) could indicate the size. Since this is an efficient use of storage, symbol table algorithms based on trees are often especially attractive for use in compilers, assemblers, and loaders.

But what about the worst case? Programmers are often skeptical of Algorithm T when they first see it. If the keys of Fig. 10 had been entered into the tree in alphabetic order AQUARIUS, . . . , VIRGO instead of the calendar order CAPRICORN, . . . , SCORPIO, the algorithm would have built a degenerate tree which essentially specifies a *sequential* search. (All LLINKS would be null.) Similarly, if the keys come in the uncommon order

AQUARIUS, VIRGO, ARIES, TAURUS, CANCER, SCORPIO,

CAPRICORN, PISCES, GEMINI, LIBRA, LEO

we obtain a "zigzag" tree which is just as bad. (Try it!)

On the other hand, the particular tree in Fig. 10 requires only $3\frac{2}{11}$ comparisons, on the average, for a successful search; this is just a little higher than the minimum possible average number of comparisons, 3, achievable in the best possible binary tree.

When we have a fairly well-balanced tree, the search time is roughly proportional to $\log N$, but when we have a degenerate tree, the search time is

roughly proportional to N. Exercise 2.3.4.5–5 proves that the average search time would be roughly proportional to $\sqrt{N}$ if we considered each N-node binary tree to be equally likely. What behavior can we really expect from Algorithm T?

Fortunately, it can be proved that tree search will require only about $2 \ln N \approx 1.386 \log_2 N$ comparisons, if the keys are inserted into the tree in random order; well-balanced trees are common, and degenerate trees are very rare.

There is a surprisingly simple proof of this fact. Let us assume that each of the $N!$ possible orderings of the N keys is an equally likely sequence of insertions for building the tree. The number of comparisons needed to find a key is exactly one more than the number of comparisons that were needed when that key was entered into the tree. Therefore if C_N is the average number of comparisons involved in a successful search and C'_N is the average number in an unsuccessful search, we have

$$C_N = 1 + \frac{C'_0 + C'_1 + \cdots + C'_{N-1}}{N}. \tag{2}$$

But the relation between internal and external path length tells us that

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1; \tag{3}$$

this is Eq. 6.2.1–2. Putting this together with (2) yields

$$(N+1)C'_N = 2N + C'_0 + C'_1 + \cdots + C'_{N-1}. \tag{4}$$

This recurrence is easy to solve. Subtracting the equation

$$NC'_{N-1} = 2(N-1) + C'_0 + C'_1 + \cdots + C'_{N-2},$$

we obtain

$$(N+1)C'_N - NC'_{N-1} = 2 + C'_{N-1},$$

$$C'_N = C'_{N-1} + 2/(N+1).$$

Since $C'_0 = 0$, this means that

$$C'_N = 2H_{N+1} - 2. \tag{5}$$

Applying (3) and simplifying yields the desired result

$$C_N = 2\left(1 + \frac{1}{N}\right) H_N - 3. \tag{6}$$

Exercises 6–8 below give more detailed information; it is possible to compute the exact probability distribution of C_N and C'_N, not merely the average values.

Tree insertion sorting. Algorithm T was developed for searching, but it can also be used as the basis of an internal *sorting* algorithm; in fact, we can view it as a natural generalization of list insertion, Algorithm 5.2.1L. When properly programmed, its average running time will be only a little slower than some of the best algorithms we discussed in Chapter 5. After the tree has been constructed for all keys, a symmetric tree traversal (Algorithm 2.3.1T) will visit the records in sorted order.

A few precautions are necessary, however. Note that something different needs to be done if $K = \texttt{KEY(P)}$ in step T2, since we are sorting instead of searching. One solution is to treat $K = \texttt{KEY(P)}$ exactly as if $K > \texttt{KEY(P)}$; this leads to a stable sorting method. (Note, however, that equal keys will not necessarily be adjacent in the tree, they will only be adjacent in symmetric order.) If many duplicate keys are present, this method will cause the tree to get badly unbalanced, and the sorting will slow down. Another idea is to keep a list, for each node, of all records having the same key; this requires another link field, but it will make the sorting faster when a lot of equal keys occur.

Thus if we are interested only in sorting, not in searching, Algorithm L isn't bad; but there are better ways to sort. On the other hand, if we have an application that combines searching and sorting, the tree method can be warmly recommended.

It is interesting to note that there is a strong relation between the analysis of tree insertion sorting and the analysis of partition exchange ("quicksort"), although the methods are superficially dissimilar. If we successively insert N keys into an initially empty tree, we make the same average number of comparisons between keys as Algorithm 5.2.2Q does, with minor exceptions. For example, in tree insertion every key gets compared with K_1, and then every key less than K_1 gets compared with the first key less than K_1, etc.; in quicksort, every key gets compared to the first partitioning element K, and then every key less than K gets compared to a particular element less than K, etc. The average number of comparisons made in both cases is NC_N. (However, Algorithm 5.2.2Q actually makes a few more comparisons, in order to speed up the inner loop.)

Deletions. Sometimes we want to make the computer forget one of the table entries it knows. It is easy to delete a "leaf" node (one in which both subtrees are empty), or to delete a node in which either LLINK or RLINK $= \Lambda$; but when both LLINK and RLINK are non-null pointers, we have to do something special, since we can't point two ways at once.

For example, consider Fig. 10 again; how can we delete CAPRICORN? One solution is to delete the *next* node, which always has a null LLINK, then reinsert it in place of the node we really wanted to delete. For example, in Fig. 10 we could delete GEMINI, then replace CAPRICORN by GEMINI. This operation preserves the essential left-to-right order of the table entries. The following algorithm gives a detailed description of one general way to do this.

Algorithm D (*Tree deletion*). Let Q be a variable which points to a node of a binary search tree represented as in Algorithm T. This algorithm deletes that node, leaving a binary search tree. (In practice, we will have either Q ≡ ROOT or Q ≡ LLINK(P) or RLINK(P) in some node of the tree. This algorithm resets the value of Q in memory, to reflect the deletion.)

D1. [Is RLINK null?] Set T ← Q. If RLINK(T) = Λ, set Q ← LLINK(T) and go to D4.

D2. [Find successor.] Set R ← RLINK(T). If LLINK(R) = Λ, set LLINK(R) ← LLINK(T), Q ← R, and go to D4.

D3. [Find null LLINK.] Set S ← LLINK(R). Then if LLINK(S) ≠ Λ, set R ← S and repeat this step until LLINK(S) = Λ. (At this point S will be equal to Q$, the symmetric successor of Q.) Finally, set LLINK(S) ← LLINK(T), LLINK(R) ← RLINK(S), RLINK(S) ← RLINK(T), Q ← S.

D4. [Free the node.] Set AVAIL ⇐ T (i.e., return the deleted node to the free storage pool). ▮

The reader may wish to try this algorithm by deleting AQUARIUS, CANCER, and CAPRICORN from Fig. 10; each case is slightly different. An alert reader may have noticed that no special test has been made for the case RLINK(T) ≠ Λ, LLINK(T) = Λ; we will defer the discussion of this case until later, since the algorithm as it stands has some very interesting properties.

Since Algorithm D is quite unsymmetrical between left and right, it stands to reason that a long sequence of random deletions and insertions will make the tree get way out of balance, so that the efficiency estimates we have made will be invalid. But actually the trees do not degenerate at all!

Theorem H (T. N. Hibbard, 1962). *After a random element is deleted from a random tree by Algorithm D, the resulting tree is still random.*

[Nonmathematical readers, please skip to (10).] This statement of the theorem is, of course, very vague. We can summarize the situation more precisely as follows: Let $\mathfrak{I}$ be a tree of n elements, and let $P(\mathfrak{I})$ be the probability that $\mathfrak{I}$ occurs if its keys are inserted in random order by Algorithm T. Some trees are more probable than others. Let $Q(\mathfrak{I})$ be the probability that $\mathfrak{I}$ will occur if $n + 1$ elements are inserted in random order by Algorithm T and then one of these elements is chosen at random and deleted by Algorithm D. In calculating $P(\mathfrak{I})$, we assume that the $n!$ permutations of the keys are equally likely; in calculating $Q(\mathfrak{I})$, we assume that the $(n + 1) \cdot (n + 1)!$ permutations of keys and selections of the key to delete are equally likely. The theorem states that $P(\mathfrak{I}) = Q(\mathfrak{I})$ for all $\mathfrak{I}$.

Proof. We are faced with the fact that permutations are equally probable, not trees, and therefore we shall prove the result by considering *permutations* as the random objects. We shall define a deletion from a permutation, and then we will prove that "a random element deleted from a random permutation leaves a random permutation."

Let $a_1a_2 \ldots a_{n+1}$ be a permutation of $\{1, 2, \ldots, n+1\}$; we want to define the operation of deleting a_i, so as to obtain a permutation $b_1b_2 \ldots b_n$ of $\{1, 2, \ldots, n\}$. This operation should correspond to Algorithms T and D, so that if we start with the tree constructed from the sequence of insertions $a_1, a_2, \ldots, a_{n+1}$ and delete a_i, renumbering the keys from 1 to n, we obtain the tree constructed from $b_1b_2 \ldots b_n$.

Fortunately it is not hard to define such a deletion operation. There are two cases:

Case 1: $a_i = n + 1$, or $a_i + 1 = a_j$ for some $j < i$. (This is essentially the condition "RLINK(a_i) = Λ.") Remove a_i from the sequence, and subtract unity from each element greater than a_i.

Case 2: $a_i + 1 = a_j$ for some $j > i$. Replace a_i by a_j, remove a_j from its original place, and subtract unity from each element greater than a_i.

For example, suppose we have the permutation 4 6 1 3 5 2. If we circle the element which is to be deleted, we have

$$\textcircled{4}6\ 1\ 3\ 5\ 2 = 4\ 5\ 1\ 3\ 2 \qquad 4\ 6\ 1\ \textcircled{3}\ 5\ 2 = 3\ 5\ 1\ 4\ 2$$
$$4\ \textcircled{6}\ 1\ 3\ 5\ 2 = 4\ 1\ 3\ 5\ 2 \qquad 4\ 6\ 1\ 3\ \textcircled{5}\ 2 = 4\ 5\ 1\ 3\ 2$$
$$4\ 6\ \textcircled{1}\ 3\ 5\ 2 = 3\ 5\ 1\ 2\ 4 \qquad 4\ 6\ 1\ 3\ 5\ \textcircled{2} = 3\ 5\ 1\ 2\ 4$$

Since there are $(n + 1) \cdot (n + 1)!$ possible deletion operations, the theorem will be established if we can show that every permutation of $\{1, 2, \ldots, n\}$ is the result of exactly $(n + 1)^2$ deletions.

Let $b_1b_2 \ldots b_n$ be a permutation of $\{1, 2, \ldots, n\}$. We shall define $(n + 1)^2$ deletions, one for each pair i, j with $1 \leq i, j \leq n + 1$, as follows:

If $i < j$, the deletion is

$$b'_1 \ldots b'_{i-1}\,\textcircled{b_i}\,b'_{i+1} \ldots b'_{j-1}(b_i + 1)b'_j \ldots b'_n. \tag{7}$$

Here, as below, b'_k stands for either b_k or $b_k + 1$, depending on whether or not b_k is less than the circled element. This deletion corresponds to Case 2.

If $i > j$, the deletion is

$$b'_1 \ldots b'_{i-1}\,\textcircled{b_j}\,b'_i \ldots b'_n; \tag{8}$$

this deletion fits the definition of Case 1.

Finally, if $i = j$, we have another Case 1 deletion, namely

$$b'_1 \ldots b'_{i-1}\,\textcircled{$n+1$}\,b'_i \ldots b'_n. \tag{9}$$

As an example, let $n = 4$ and consider the 25 deletions which map into 3 1 4 2:

	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$
$j=1$	⑤3 1 4 2	4③1 5 2	4 1③5 2	4 1 5③2	4 1 5 2③
$j=2$	③4 1 5 2	3⑤1 4 2	4 2①5 3	4 2 5①3	4 2 5 3①
$j=3$	③1 4 5 2	4①2 5 3	3 1⑤4 2	3 1 5④2	3 1 5 2④
$j=4$	③1 5 4 2	4①5 2 3	3 1④5 2	3 1 4⑤2	4 1 5 3②
$j=5$	③1 5 2 4	4①5 3 2	3 1④2 5	4 1 5②3	3 1 4 2⑤

The circled element is always in position i, and for fixed i we have clearly constructed $n+1$ different deletions; hence $(n+1)^2$ different deletions have been constructed for each permutation $b_1 b_2 \ldots b_n$. Since only $(n+1)^2 n!$ deletions are possible, we must have found all of them. ∎

The proof of Theorem H not only tells us about the result of deletions, it also helps us analyze the running time in an average deletion. Exercise 12 shows that we can expect to execute step D2 slightly less than half the time, on the average, when deleting a random element from a random table.

Let us now consider how often the loop in step D3 needs to be performed: Suppose that we are deleting a node on level l, and that the *external* node immediately following in symmetric order is on level k. For example, if we are deleting CAPRICORN from Fig. 10, we have $l=0$ and $k=3$ since node $\boxed{4}$ is on level 3. If $k=l+1$, we have RLINK(T) $=\Lambda$ in step D1; and if $k>l+1$, we will set S $\leftarrow$ LLINK(R) exactly $k-l-2$ times in step D3. The average value of l is (internal path length)$/N$; the average value of k is (external path length $-$ distance to leftmost external node)$/N$. The distance to the leftmost external node is the number of left-to-right minima in the insertion sequence, so it has the average value H_N by the analysis of Section 1.2.10. Since external path length minus internal path length is $2N$, the average value of $k-l-2$ is $-H_N/N$. Adding to this the average number of times that $k-l-2$ is -1, we see that *the operation* S $\leftarrow$ LLINK(R) *in step D3 is performed only*

$$\frac{1}{2}+\frac{\frac{1}{2}-H_N}{N} \tag{10}$$

times, on the average, in a random deletion. This is reassuring, since the worst case can be pretty slow (see exercise 11).

As mentioned above, Algorithm D does not test for the case LLINK(T) $=\Lambda$, although this is one of the easy cases for deletion. We could add a new step between D1 and D2, namely,

D1½. [Is LLINK null?] If LLINK(T) $=\Lambda$, set Q $\leftarrow$ RLINK(T) and go to D4.

Exercise 14 shows that Algorithm D with this extra step always leaves a tree

that is at least as good as the original Algorithm D, in the path-length sense, and sometimes the result is even better. Thus, a sequence of insertions and deletions using this modification of Algorithm D will result in trees which are actually *better* than the theory of random trees would predict: the average computation time for search and insertion will tend to decrease as time goes on.

Frequency of access. So far we have assumed that each key was equally likely as a search argument. In a more general situation, let p_k be the probability that we will search for the kth element inserted, where $p_1 + \cdots + p_N = 1$. Then a straightforward modification of Eq. (2), if we retain the assumption of random order so that the shape of the tree stays random, shows that the average number of comparisons in a successful search will be

$$1 + \sum_{1\le k\le N} p_k(2H_k - 2) = 2 \sum_{1\le k\le N} p_k H_k - 1. \tag{11}$$

(Cf. Eq. (5).)

For example, if the probabilities obey Zipf's law, Eq. 6.1–8, the average number of comparisons reduces to

$$H_N - 1 + H_N^{(2)}/H_N \tag{12}$$

if we insert the keys in decreasing order of importance. (See exercise 18.) This is about half as many comparisons as predicted by the equal-frequency analysis, and it is less comparisons than we would make using binary search.

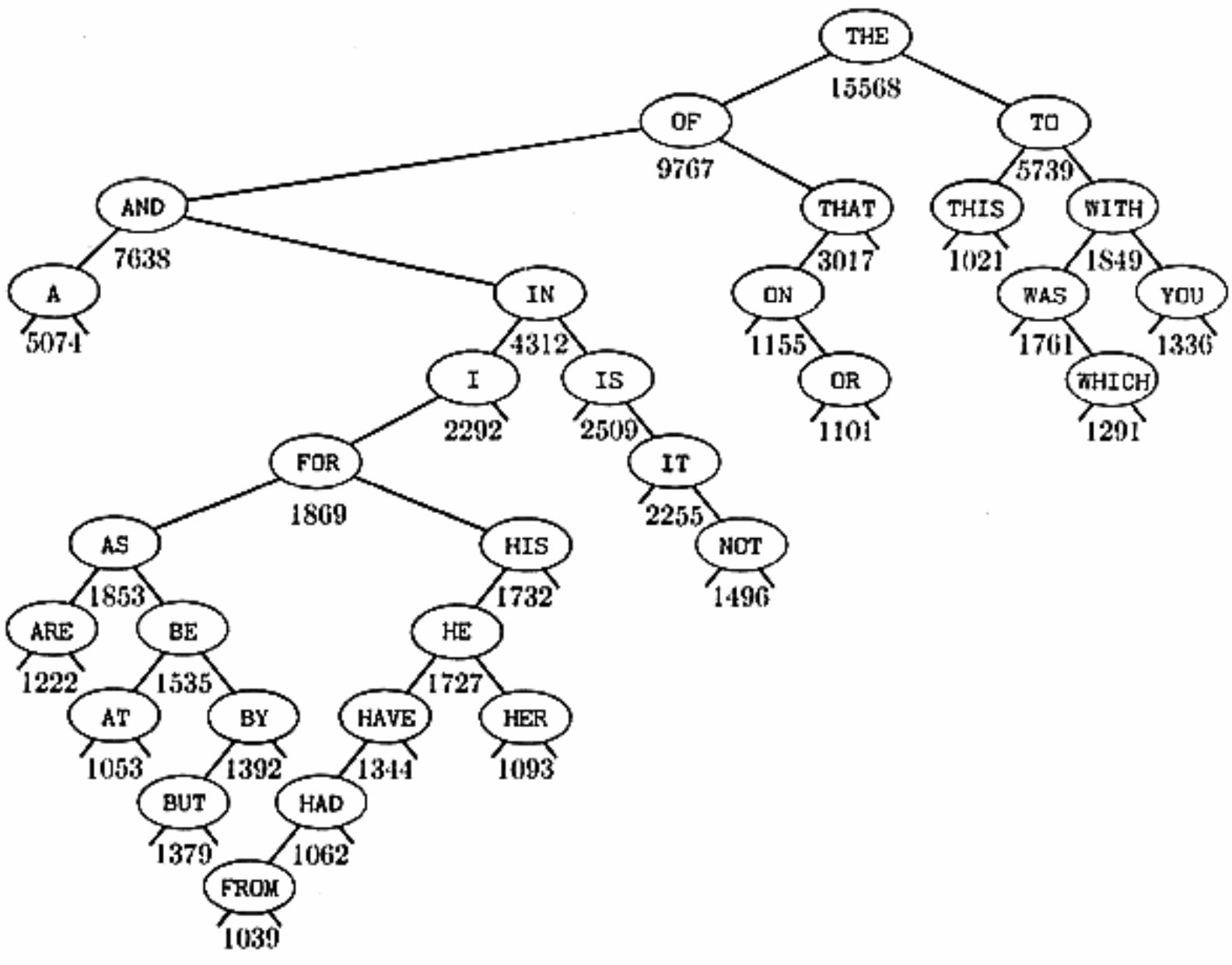


Fig. 12. The 31 most common English words, inserted in decreasing order of frequency.

For example, Fig. 12 shows the tree which results when the most common 31 words of English are entered in decreasing order of frequency. The relative frequency is shown with each word [cf. *Cryptanalysis* by H. F. Gaines (New York: Dover, 1956), p. 226]. The average number of comparisons for a successful search in this tree is 4.042; the corresponding binary search, using Algorithm 6.2.1B or C, would require 4.393 comparisons.

Optimum binary search trees. These considerations make it natural to ask about the best possible tree for searching a table of keys with given frequencies. For example, the optimum tree for the 31 most common English words is shown in Fig. 13; it requires only 3.437 comparisons for an average successful search.

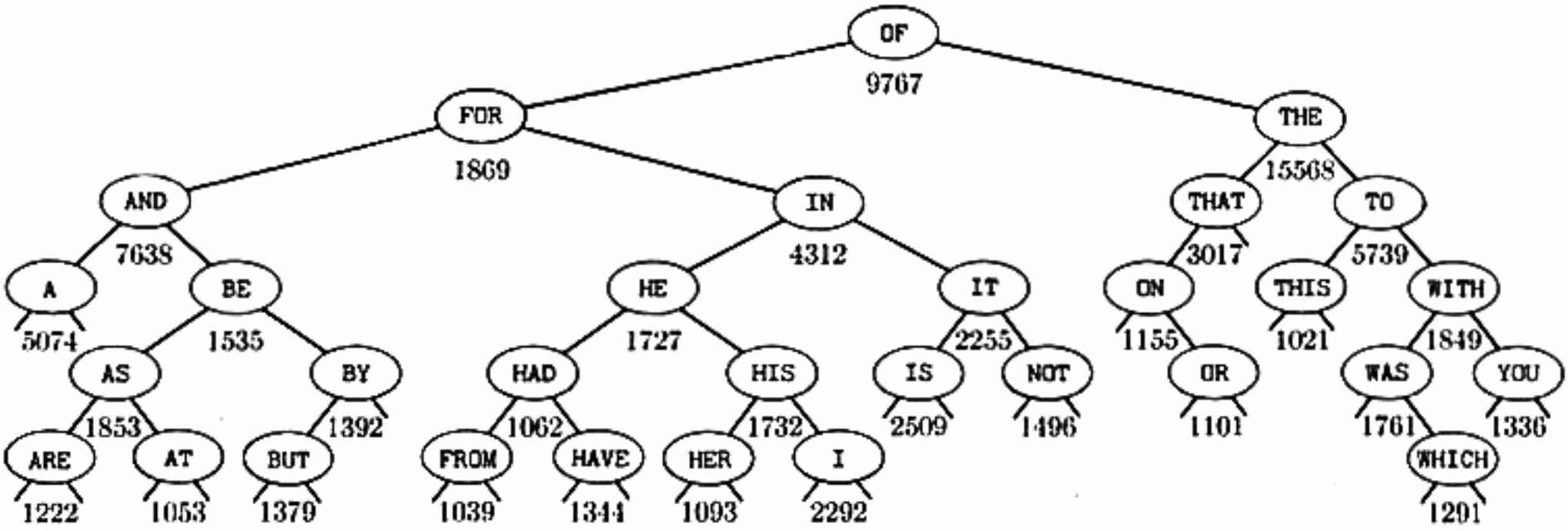
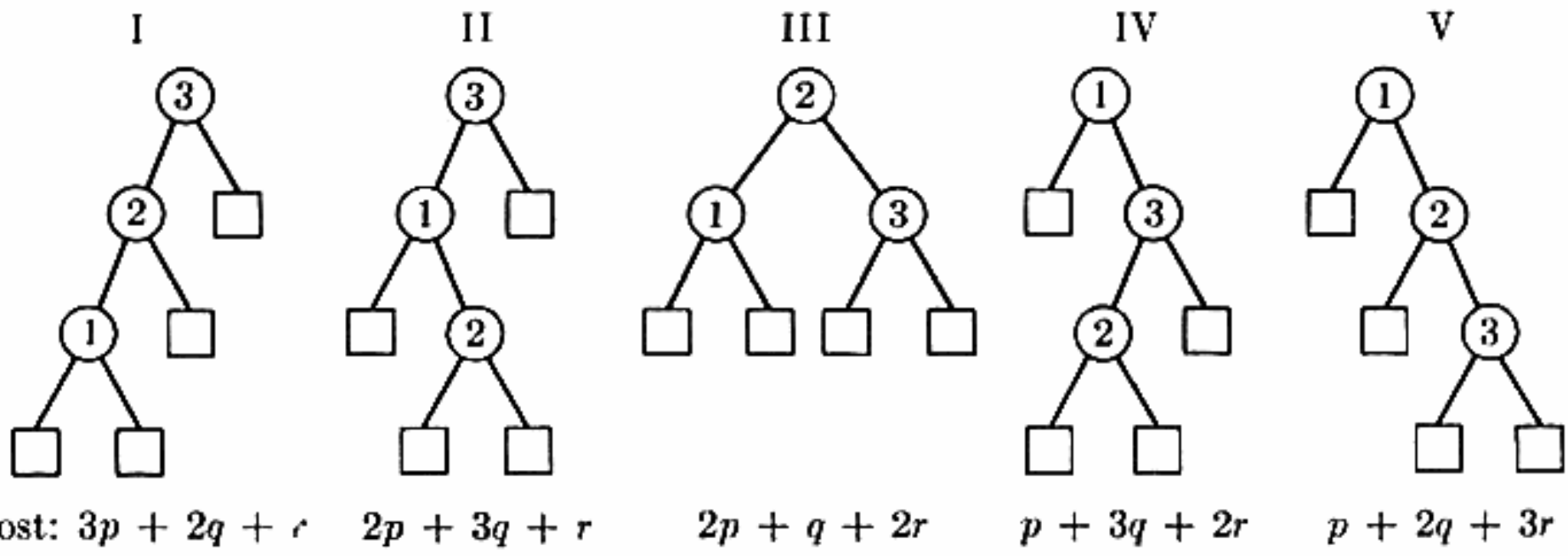


Fig. 13. Optimum search tree for the 31 most common English words.

Let us now explore the problem of finding the optimum tree. When $N = 3$, for example, let us assume that the keys $K_1 < K_2 < K_3$ have respective probabilities p, q, r. There are five possible trees:



(13)

Figure 14 shows the ranges of p, q, r for which each tree is optimum; the balanced tree is best about 45 percent of the time, if we choose p, q, r at random (see exercise 21).

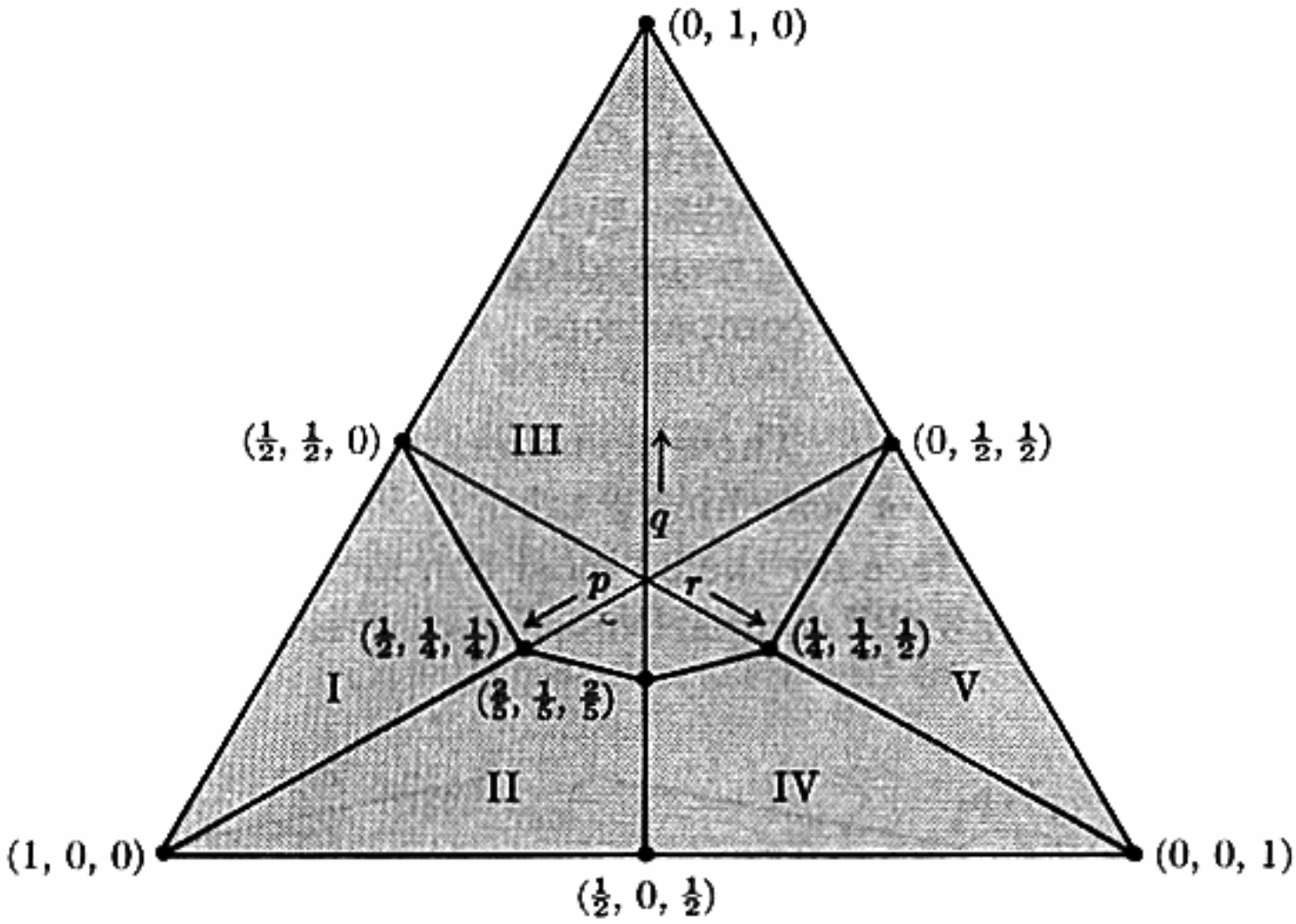


Fig. 14. If the relative frequencies of (K_1, K_2, K_3) are (p, q, r), this graph shows which of the five trees in (13) is best. The fact that $p + q + r = 1$ makes the graph two-dimensional although there are three coordinates.

Unfortunately, when N is large there are

$$\binom{2N}{N} \Big/ (N+1) \approx 4^N/(\sqrt{\pi}\, N^{3/2})$$

binary trees, so we can't just try them all and see which is best. Let us therefore study the properties of optimum binary search trees more closely, in order to discover a better way to find them.

So far we have considered only the probabilities for a successful search; in practice, the unsuccessful case must usually be considered as well. For example, the 31 words in Fig. 13 account for only about 36 percent of typical English text; the other 64 percent will certainly influence the structure of the optimum search tree.

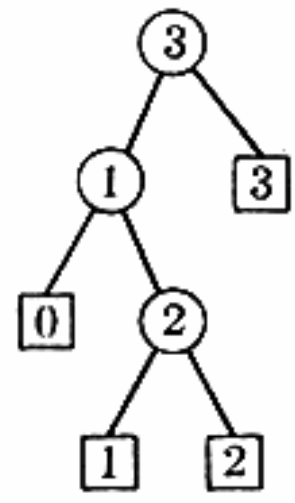
Therefore let us set the problem up in the following way: We are given $2n + 1$ probabilities $p_1, p_2, \ldots, p_n$ and $q_0, q_1, \ldots, q_n$, where

p_i = probability that K_i is the search argument;
q_i = probability that the search argument lies between K_i and K_{i+1}.

(By convention, q_0 is the probability that the search argument is less than K_1, and q_n is the probability that the search argument is greater than K_n.) Thus, $p_1 + p_2 + \cdots + p_n + q_0 + q_1 + \cdots + q_n = 1$, and we want to find a binary tree which minimizes the expected number of comparisons in the search, namely

$$\sum_{1 \le j \le n} p_j(\text{level}(\textcircled{j}) + 1) + \sum_{0 \le k \le n} q_k \,\text{level}(\boxed{k}), \tag{14}$$

where (j) is the jth internal node in symmetric order and [k] is the $(k+1)$st external node, and where the root has level zero. Thus the expected number of comparisons for the binary tree



(15)

is $2q_0 + 2p_1 + 3q_1 + 3p_2 + 3q_2 + p_3 + q_3$. Let us call this the *cost* of the tree; and let us say that a minimum-cost tree is *optimum*. In this definition there is no need to require that the p's and q's sum to unity, we can ask for a minimum-cost tree with any given sequence of "weights" $(p_1, \ldots, p_n; q_0, \ldots, q_n)$.

We have studied Huffman's procedure for constructing trees with minimum weighted path length, in Section 2.3.4.5; but that method requires all the p's to be zero, and the tree it produces will usually not have the external node weights $(q_0, \ldots, q_n)$ in the proper symmetric order from left to right. Therefore we need another approach.

The principle which saves us is that *all subtrees of an optimum tree are optimum.* For example if (15) is an optimum tree for the weights $(p_1, p_2, p_3; q_0, q_1, q_2, q_3)$, then the left subtree of the root must be optimum for $(p_1, p_2; q_0, q_1, q_2)$; any improvement to a subtree leads to an improvement in the whole tree.

This principle suggests a computation procedure which systematically finds larger and larger optimum subtrees. We have used much the same idea in Section 5.4.9 to construct optimum merge patterns; the general approach is known as "dynamic programming," and we shall consider it further in Chapter 7.

Let $c(i, j)$ be the cost of an optimum subtree with weights $(p_{i+1}, \ldots, p_j; q_i, \ldots, q_j)$; and let $w(i, j) = p_{i+1} + \cdots + p_j + q_i + \cdots + q_j$ be the sum of all those weights; $c(i, j)$ and $w(i, j)$ are defined for $0 \le i \le j \le n$. It follows that

$$\begin{aligned} c(i, i) &= 0, \\ c(i, j) &= w(i, j) + \min_{i \le k < j} \big(c(i, k-1) + c(k, j)\big), \qquad \text{for} \quad i < j, \end{aligned} \tag{16}$$

since the minimum possible cost of a tree with root (k) is $w(i, j) + c(i, k-1) + c(k, j)$. When $i < j$, let $R(i, j)$ be the set of all k for which the minimum is achieved in (16); this set specifies the possible roots of the optimum trees.

Equation (16) makes it possible to evaluate $c(i, j)$ for $j - i = 1, 2, 3, \ldots, n$; there are about $\frac{1}{2}n^2$ such values, and the minimization operation is carried out for about $\frac{1}{6}n^3$ values of k. This means we can determine an optimum tree in $O(n^3)$ units of time, using $O(n^2)$ cells of memory.

A factor of n can actually be removed from the running time if we make use of a "monotonicity" property. Let $r(i, j)$ denote an element of $R(i, j)$; we need not compute the entire set $R(i, j)$, a single representative is sufficient. Once we have found $r(i, j-1)$ and $r(i+1, j)$, the result of exercise 27 proves that we may always assume that

$$r(i, j-1) \le r(i, j) \le r(i+1, j) \tag{17}$$

when the weights are nonnegative. This limits the search for the minimum, since only $r(i+1, j) - r(i, j-1) + 1$ values of k need to be examined in (16) instead of $j - i$. The total amount of work when $j - i = d$ is now bounded by the telescoping series

$$\sum_{\substack{d \le j \le n \\ i = j-d}} (r(i+1, j) - r(i, j-1) + 1)$$
$$= r(n-d+1, n) - r(0, d-1) + n - d + 1 < 2n,$$

hence the total running time is reduced to $O(n^2)$.

The following algorithm describes this procedure in detail.

Algorithm K (*Find optimum binary search trees*). Given $2n + 1$ nonnegative weights $(p_1, \ldots, p_n; q_0, \ldots, q_n)$, this algorithm constructs binary trees $t(i, j)$ which have minimum cost for the weights $(p_{i+1}, \ldots, p_j; q_i, \ldots, q_j)$ in the sense defined above. Three arrays are computed, namely

$c[i, j]$,	for	$0 \le i \le j \le n$,	the cost of $t(i, j)$;
$r[i, j]$,	for	$0 \le i \le j \le n$,	the root of $t(i, j)$;
$w[i, j]$,	for	$0 \le i \le j \le n$,	the total weight of $t(i, j)$.

The results of the algorithm are specified by the r array: If $i = j$, $t(i, j)$ is null; else its left subtree is $t(i, r[i, j] - 1)$ and its right subtree is $t(r[i, j], j)$.

K1. [Initialize.] For $0 \le i \le n$, set $c[i, i] \leftarrow 0$ and $w[i, i] \leftarrow q_i$ and $w[i, j] \leftarrow w[i, j-1] + p_j + q_j$ for $j = i+1, \ldots, n$. Then for $1 \le j \le n$ set $c[j-1, j] \leftarrow w[j-1, j]$ and $r[j-1, j] \leftarrow j$. (This determines all the 1-node optimum trees.)

K2. [Loop on d.] Do step K3 for $d = 2, 3, \ldots, n$, then terminate the algorithm.

K3. [Loop on j.] (We have already determined the optimum trees of less than d nodes. This step determines all the d-node optimum trees.) Do step K4 for $j = d, d+1, \ldots, n$.

K4. [Find $c[i, j]$, $r[i, j]$.] Set $i \leftarrow j - d$. Then set

$$c[i, j] \leftarrow w[i, j] + \min_{r[i, j-1] \le k \le r[i+1, j]} (c[i, k-1] + c[k, j]),$$

and set $r[i, j]$ to a value of k for which the minimum occurs. (Exercise 22 proves that $r[i, j-1] \le r[i+1, j]$.) ∎

As an example of Algorithm K, consider Fig. 15, which is based on a "keyword-in-context" (KWIC) indexing application. The titles of all articles in the first ten volumes of the ACM *Journal* were sorted to prepare a concordance in which there is one line for every word of every title. However, certain words like "THE" and "EQUATION" were felt to be sufficiently uninformative that they were left out of the index. These special words and their frequency of occurrence are shown in the internal nodes of Fig. 15. Note that a title such as "On the solution of an equation for a certain new problem" would be so uninformative, it wouldn't appear in the index at all! The idea of KWIC indexing is due to H. P. Luhn, *Amer. Documentation* **11** (1960), 288–295. (See W. W. Youden, *JACM* **10** (1963), 583–646, where the full KWIC index appears.)

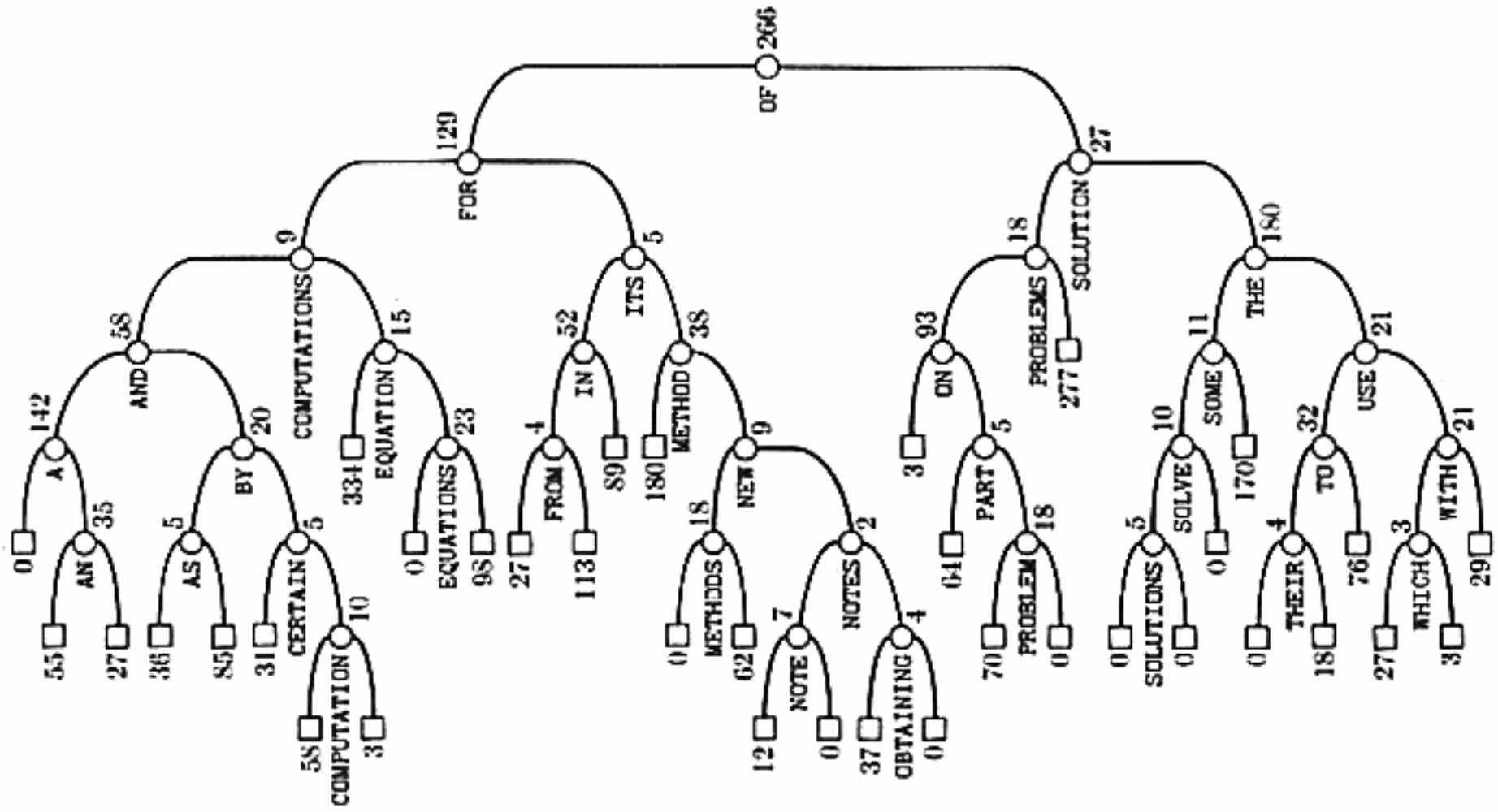


Fig. 15. An optimum binary search tree for a KWIC indexing application.

When preparing a KWIC index file for sorting, we might want to use a binary search tree in order to test whether or not each particular word is to be indexed. The other words fall between two of the unindexed words, with the frequencies shown in the external nodes of Fig. 15; thus, exactly 277 words which are alphabetically between "PROBLEMS" and "SOLUTION" appeared in the *JACM* titles during 1954–1963.

Figure 15 shows the optimum tree obtained by Algorithm K, with $n = 35$. The computed values of $r[0, j]$ for $j = 1, 2, \ldots, 35$ are (1, 1, 2, 3, 3, 3, 3, 8, 8, 8, 8, 8, 8, 11, 11, . . . , 11, 21, 21, 21, 21, 21, 21); the values of $r[i, 35]$ for $i = 0, 1, \ldots, 34$ are (21, 21, . . . , 21, 25, 25, 25, 25, 25, 25, 26, 26, 26, 30, 30, 30, 30, 30, 30, 30, 33, 33, 33, 35, 35).

The "betweenness frequencies" q_j have a noticeable effect on the optimum tree structure; Fig. 16(a) shows the optimum tree that would have been obtained with the q_j set to zero. Similarly, the internal frequencies p_i are important: Fig. 16(b) shows the optimum tree when the p_i are set to zero. Considering

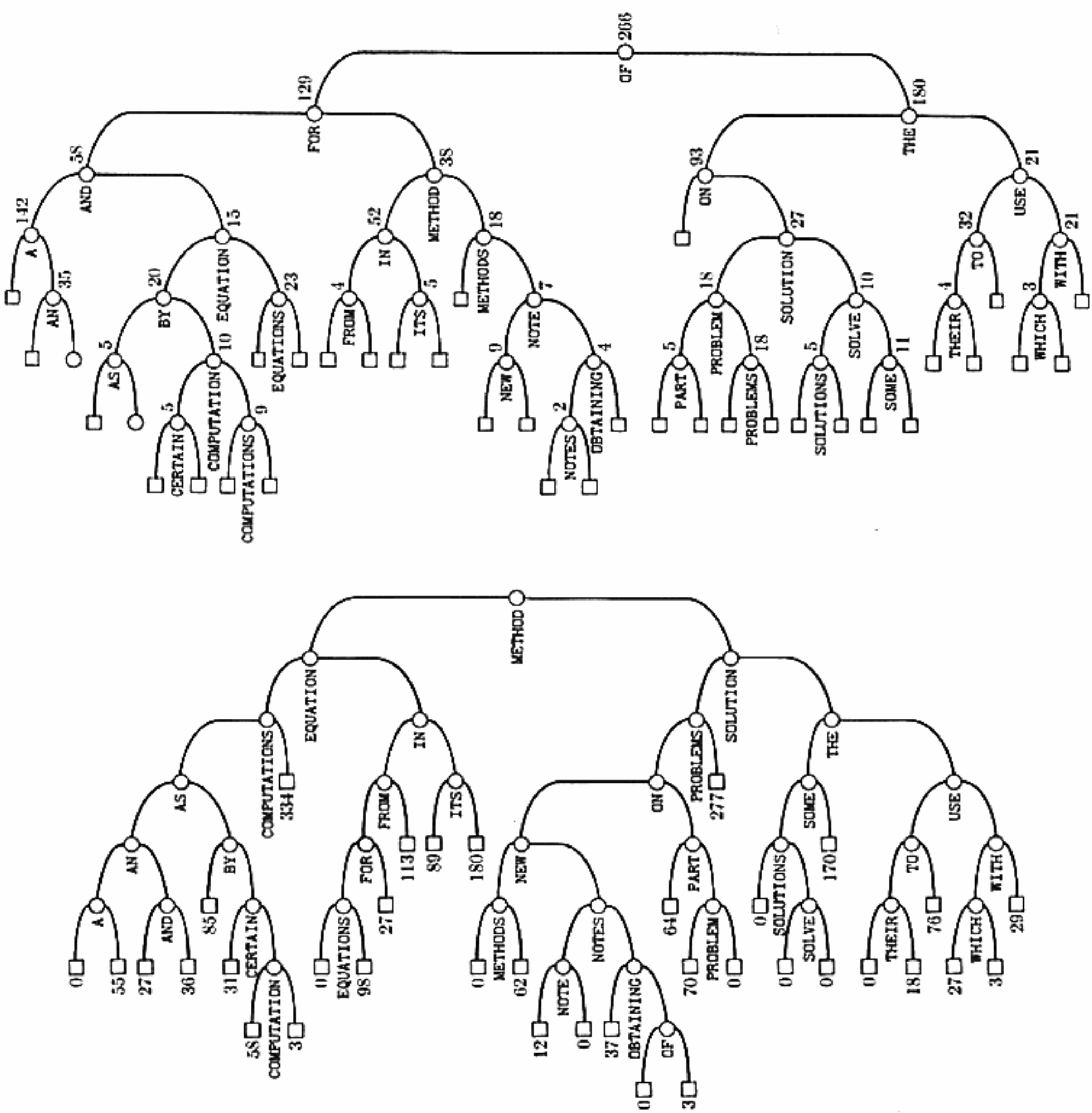


Fig. 16. Optimum binary search trees based on half of the data of Fig. 15; (a) external frequencies suppressed, (b) internal frequencies suppressed.

the full set of frequencies, the tree of Fig. 15 requires only 4.75 comparisons, on the average, while the trees of Fig. 16 require, respectively, 5.29 and 5.32. (A straight binary search would have been better than the trees of Fig. 16, in this example.)

Since Algorithm K requires time and space proportional to n^2, it becomes impractical to use it when n is very large. Of course we may not really want to use binary search trees for large n, in view of the other search techniques to be discussed later in this chapter; but let's assume anyway that we want to find an optimum or nearly optimum tree when n is large.

We have seen that the idea of inserting the keys in order of decreasing frequency can tend to make a fairly good tree, on the average; but it can also

be very bad (see exercise 20), and it is not usually very near the optimum, since it makes no use of the q_j weights. Another approach is to choose the root k so that the weights $w(0, k-1)$ and $w(k, n)$ of the resulting subtrees are as near to being equal as possible. This approach also fails, because it may choose a node with very small p_k to be the root.

A fairly satisfactory procedure can be obtained by combining these two methods, as suggested by W. A. Walker and C. C. Gotlieb [*Graph Theory and Computing* (Academic Press, 1972)]: Try to equalize the left-hand and right-hand weights, but be prepared to move the root a few steps to the left or right to find a node with relatively large p_k. Figure 17 shows why this method is reasonable: If we plot $c(0, k-1) + c(k, n)$ as a function of k, for the KWIC data of Fig. 15, we see that the result is quite sensitive to the magnitude of p_k.

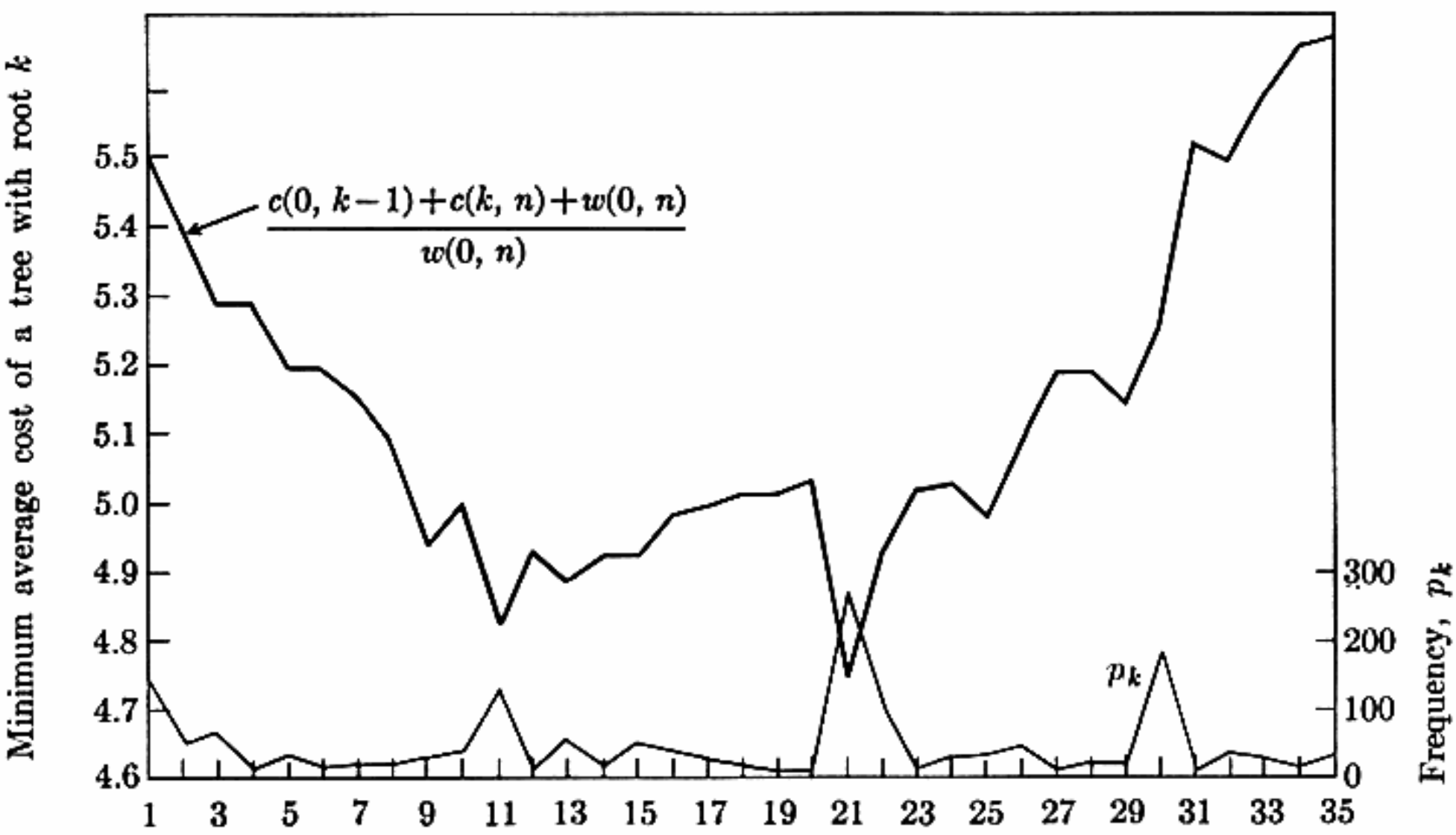


Fig. 17. Behavior of the cost as a function of the root, k.

A "top-down" method such as this can be used for large n to choose the root and then to work on the left and the right subtrees. When we get down to a sufficiently small subtree we can apply Algorithm K. The resulting method yields fairly good trees (reportedly within 2 or 3 percent of the optimum), and it requires only $O(n)$ units of space, $O(n \log n)$ units of time.

***The Hu-Tucker algorithm.** In the special case that all the p's are zero, T. C. Hu and A. C. Tucker have discovered a remarkable "bottom-up" way to construct optimum trees; if appropriate data structures are used, their method requires $O(n)$ units of space and $O(n \log n)$ units of time, and it constructs a tree which is *really* optimum (not just approximately so).

The Hu-Tucker algorithm can be described as follows.

• PHASE 1, Combination. Start with the "working sequence" of weights written inside of *external* nodes,

$$\boxed{q_0} \quad \boxed{q_1} \quad \boxed{q_2} \quad \cdots \quad \boxed{q_n} . \tag{18}$$

Then repeatedly combine two weights q_i and q_j for $i < j$ into a single weight $q_i + q_j$, deleting the node containing q_j from the working sequence and replacing the node containing q_i by the *internal* node

$$\textcircled{q_i + q_j} . \tag{19}$$

This combination is to be done on the unique pair of weights (q_i, q_j) satisfying the following rules:

i) No external nodes occur between q_i and q_j. (This is the most important rule which distinguishes the algorithm from Huffman's method.)

ii) The sum $q_i + q_j$ is minimum over all (q_i, q_j) satisfying rule (i).

iii) The index i is minimum over all (q_i, q_j) satisfying rules (i), (ii).

iv) The index j is minimum over all (q_i, q_j) satisfying rules (i), (ii), (iii).

• PHASE 2, Level assignment. When Phase 1 ends, there is a single node left in the working sequence. Mark it with level number 0. Then undo the steps of Phase 1 in reverse order, marking level numbers of the corresponding tree; if (19) has level l, the nodes containing q_i and q_j which formed it are marked with level $l + 1$.

• PHASE 3, Recombination. Now we have the working sequence of external nodes and levels

$$\underset{l_1}{\boxed{q_0}} \quad \underset{l_2}{\boxed{q_1}} \quad \underset{l_3}{\boxed{q_2}} \quad \cdots \quad \underset{l_n}{\boxed{q_n}} .$$

The internal nodes used in Phases 1 and 2 are now discarded, we shall create new ones by combining weights (q_i, q_j) according to the following new rules:

i′) The nodes containing q_i and q_j must be adjacent in the working sequence.

ii′) The levels l_i and l_j must both be the maximum among all remaining levels.

iii′) The index i must be minimum over all (q_i, q_j) satisfying (i′), (ii′).

The new node (19) is assigned level $l_i - 1$. The binary tree formed during this phase has minimum weighted path length over all binary trees whose external nodes are weighted $q_0, q_1, \ldots, q_n$ from left to right.

Figure 18 shows an example of this algorithm; the weights q_i are the relative frequencies of the letters ␣, A, B, . . . , Z in English text. During Phase 1, the

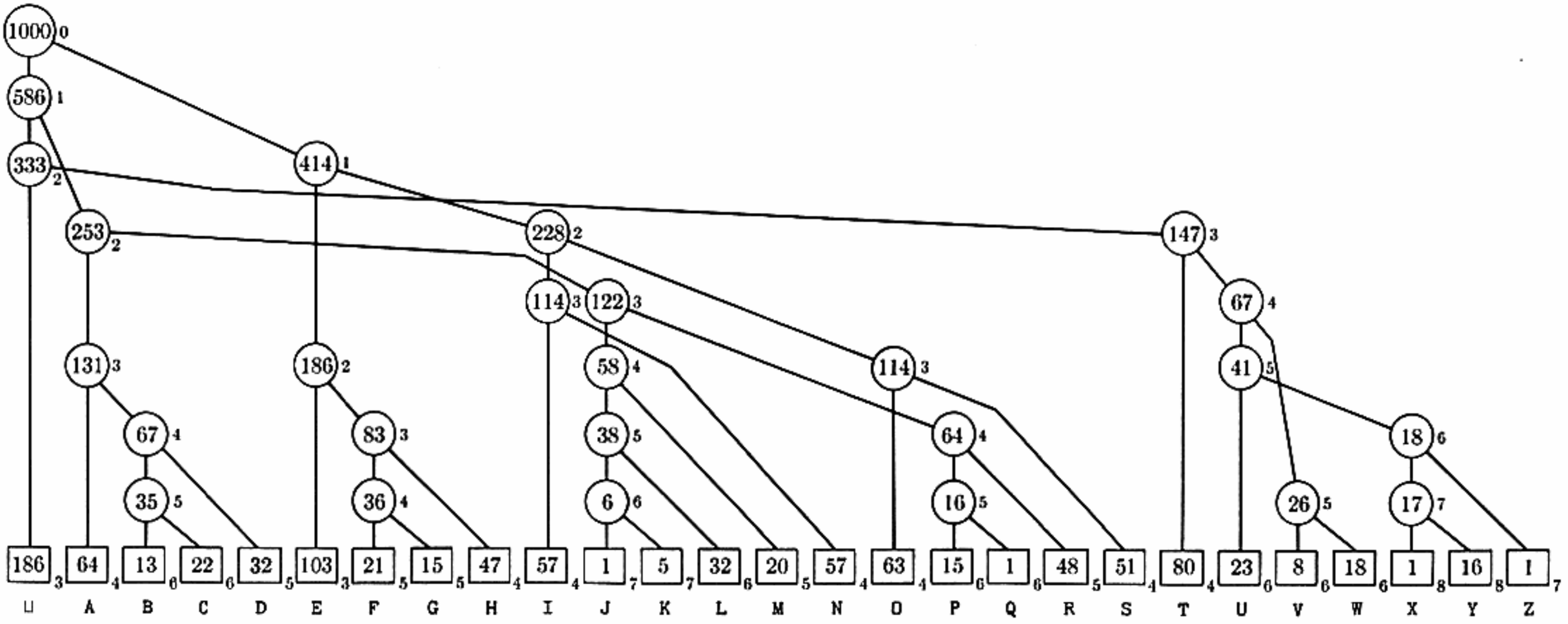


Fig. 18. The Hu-Tucker algorithm applied to alphabetic frequency data: Phases 1 and 2.

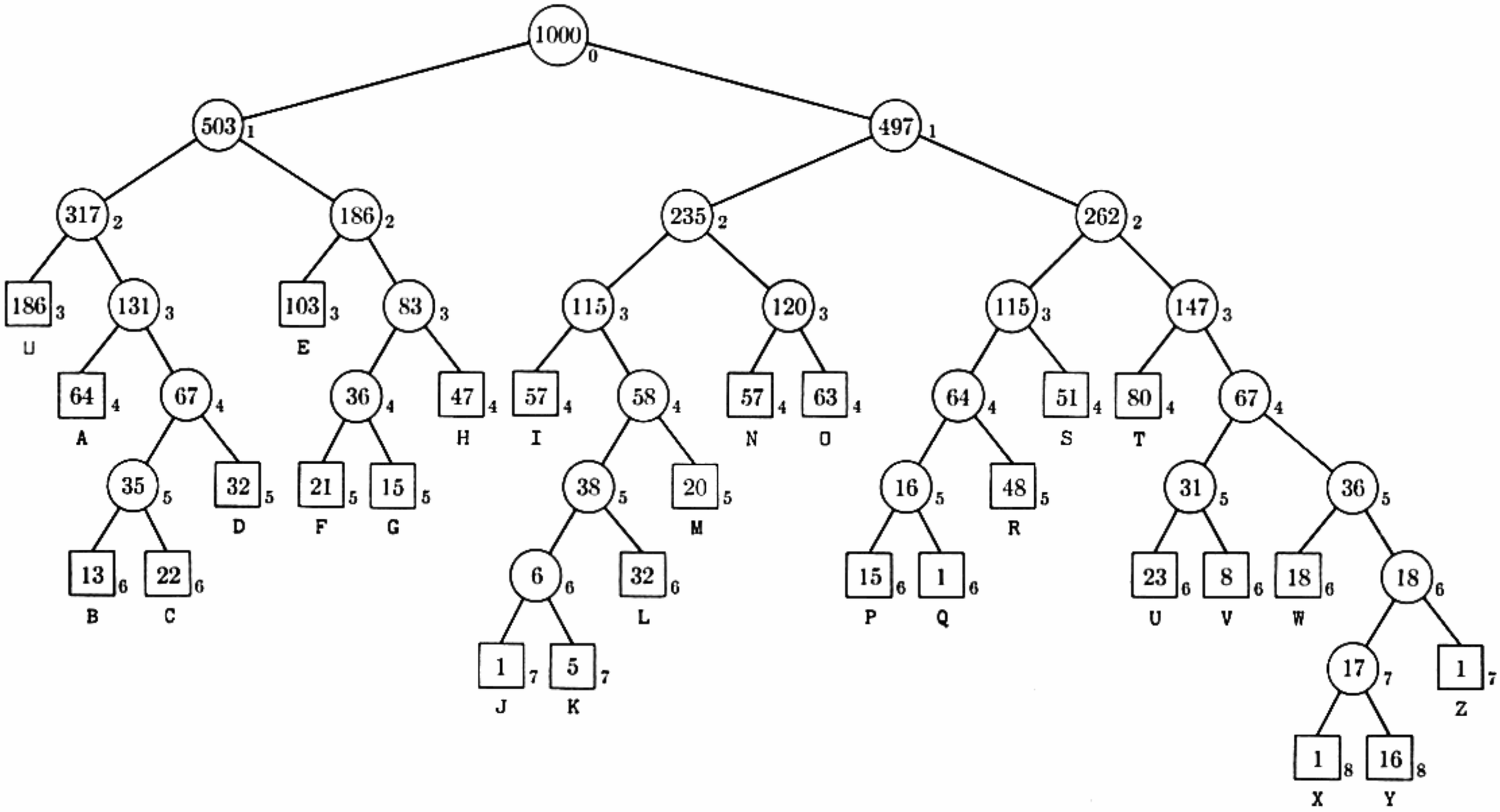


Fig. 19. The Hu-Tucker algorithm applied to alphabetic frequency data: Phase 3.

first node formed is (6), combining the J and K frequencies; then the node (16) is formed (combining P and Q), then

$$(17),\ (18),\ (26),\ (35),\ (36),\ (38),\ (41),\ (58),\ (64),\ (67),\ (67),\ (83);$$

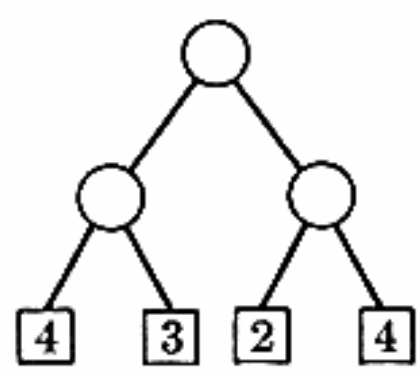
at this point we have the working sequence

$$\boxed{186}\ \boxed{64}\ (67)\ \boxed{103}\ (83)\ \boxed{57}\ (58)\ \boxed{57}\ \boxed{63}\ (64)\ \boxed{51}\ \boxed{80}\ (67). \tag{20}$$

Rule (i) allows us to combine nonadjacent weights only if they are separated by internal nodes; so we can combine $57 + 57$, then $63 + 51$, then $58 + 64$, etc.

The level numbers assigned during Phase 2 appear at the right of each node in Fig. 18. The recombination during Phase 3 now yields the tree shown in Fig. 19; note that things must be associated differently in this tree than in Fig. 18, because Fig. 18 does not preserve the left-to-right ordering. But Fig. 19 has the same cost as Fig. 18, since the external nodes appear at the same levels in both trees.

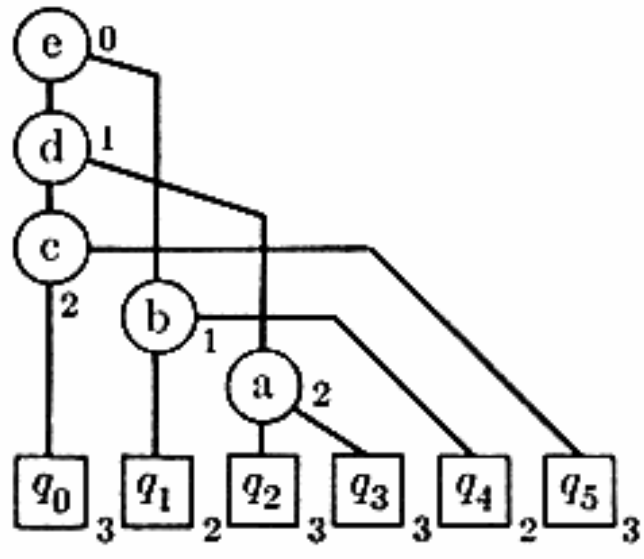
Consider a simple example where the weights are 4, 3, 2, 4; the unique optimum tree is easily shown to be



(21)

This example shows that the two smallest weights, 2 and 3, should *not* always be combined in an optimum tree, even when they are adjacent; some recombination phase is needed.

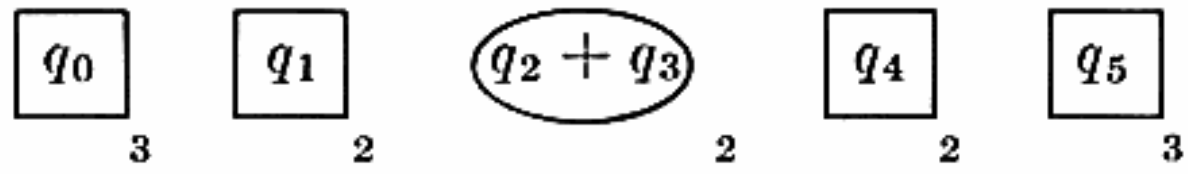
It is beyond the scope of this book to give a proof that the Hu-Tucker algorithm is valid; no simple proof is known, and it is quite possible that no simple proof will ever be found! In order to illustrate the inherent complexities of the situation, note that Phase 3 must combine all nodes into a single tree, and this is not obviously possible. For example, suppose that Phases 1 and 2 were to construct the tree



(22)

by combining nodes (a), (b), (c), (d), (e) in this order; this accords with rule

(i). Then Phase 3 will get stuck after forming



because the two level-3 nodes are not adjacent! Rule (i) does not by itself guarantee that Phase 3 will be able to proceed, and it is necessary to prove that configurations like (22) will *never* be constructed during Phase 1.

When implementing the Hu-Tucker algorithm, we can maintain priority queues for the sets of node weights which are not separated by external nodes. For example, (20) could be represented by priority queues containing, respectively,

$$\begin{array}{cccccccc} 64 & 64 & 57 & 57 & 57 & 51 & 51 & 67 \\ 186 & 67 & 83 & 57 & 63 & 63 & 80 & 80 \\ & 103 & 103 & 58 & & 64 & & \end{array} \qquad (23)$$

plus information about which of these is external, and an indication of left-to-right order for breaking ties by rules (iii) and (iv). Another "master" priority queue can keep track of the sums of the two least elements in the other queues. The creation of the new node $57 + 57$ causes three of the above priority queues to be merged. When priority queues are represented as leftist trees (cf. Section 5.2.3), each combination step of Phase 1 requires at most $O(\log n)$ operations; thus $O(n \log n)$ operations suffice as $n \to \infty$. Of course for small n it will be more efficient to use a comparatively straightforward $O(n^2)$ method of implementation.

The optimum binary tree in Fig. 19 has an interesting application to coding theory as well as to searching: Using 0 to stand for a left branch in the tree and 1 to stand for a right branch, we obtain the following variable-length codewords:

␣	000	I	1000	R	11001
A	0010	J	1001000	S	1101
B	001100	K	1001001	T	1110
C	001101	L	100101	U	111100
D	00111	M	10011	V	111101
E	010	N	1010	W	111110
F	01100	O	1011	X	11111100
G	01101	P	110000	Y	11111101
H	0111	Q	110001	Z	1111111

(24)

Thus a message like "RIGHT ON" would be encoded by the string

110011000011010111111000010111010.

Note that decoding from left to right is easy, in spite of the variable length of the codewords, because the tree structure tells us when one codeword ends and another begins. This method of coding preserves the alphabetical order of messages, and it uses an average of about 4.2 bits per letter. Thus the code could be used to compress data files, without destroying lexicographic order of alphabetic information. (The figure of 4.2 bits per letter is minimum over all binary tree codes, although it could be reduced to 4.1 bits per letter if we disregarded the alphabetic ordering constraint. A further reduction, preserving alphabetic order, could be achieved if pairs of letters instead of single letters were encoded.)

An interesting asymptotic bound on the minimum weighted path length of search trees has been derived by E. N. Gilbert and E. F. Moore:

Theorem G. *If* $p_1 = p_2 = \cdots = p_n = 0$, *the weighted path length of an optimum binary search tree lies between*

$$\sum_{0\le i\le n} q_i \log_2 (Q/q_i) \qquad \textit{and} \qquad 2Q + \sum_{0\le i\le n} q_i \log_2 (Q/q_i),$$

where $Q = \sum_{0\le i\le n} q_i$.

Proof. To get the lower bound, we use induction on n. If $n > 0$ the weighted external path length is at least

$$Q + \sum_{0\le i<k} q_i \log_2 (Q_1/q_i) + \sum_{k\le i\le n} q_i \log_2 ((Q - Q_1)/q_i) \ge \sum_{0\le i\le n} q_i \log_2 (Q/q_i) + f(Q_1),$$

for some k, where

$$Q_1 = \sum_{0\le i<k} q_i,$$

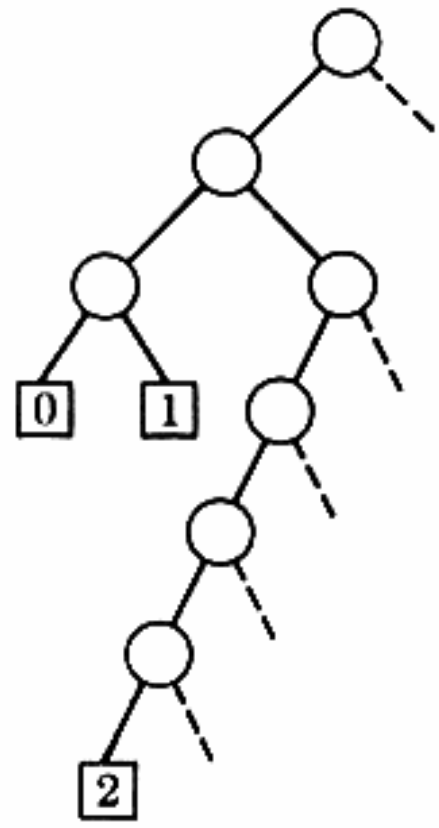
and

$$f(Q_1) = Q + Q_1 \log_2 Q_1 + (Q - Q_1) \log_2 (Q - Q_1) - Q \log_2 Q.$$

The function $f(Q_1)$ is nonnegative, and it takes its minimum value 0 when $Q_1 = \frac{1}{2}Q$.

To get the upper bound, we may assume that $Q = 1$. Let $e_0, \ldots, e_n$ be integers such that $2^{-e_i} \le q_i < 2^{1-e_i}$, for $0 \le i \le n$. Construct codewords C_i of 0's and 1's, by using the most significant $e_i + 1$ binary digits of the fraction $\sum_{0\le k<i} q_k + \frac{1}{2}q_i$, expressed in binary notation. Exercise 35 proves that C_i is never an initial substring of C_j when $i \ne j$; it follows that we can construct a binary search tree corresponding to these codewords. For example when the q's are the letter frequencies of Fig. 19, this construction gives $C_0 = 0001$,

$C_1 = 00110$, $C_2 = 01000001$, $C_3 = 0100011$, etc.; the tree begins



(25)

(Redundant bits at the right end of the codes can often be removed.) The weighted path length of the binary tree constructed by this general procedure is

$$\leq \sum_{0\leq i\leq n} (e_i + 1)q_i < \sum_{0\leq i\leq n} q_i(2 + \log(1/q_i)). \quad \blacksquare$$

The first part of the above proof is readily extended to show that the weighted path length of *every* binary tree must be at least $\sum_{0\leq i\leq n} q_i \log_2 (Q/q_i)$, whether or not the weights are required to be in order from left to right. (This fundamental result is due to Claude Shannon.) Therefore the left-to-right constraint does not raise the cost of the minimum tree by more than the cost of two extra levels, i.e., twice the total weight.

History and bibliography. The tree search methods of this section were discovered independently by several people during the 1950's. In an unpublished memorandum dated August, 1952, A. I. Dumey described a primitive form of tree insertion:

"Consider a drum with 2^n item storages in it, each having a binary address. Follow this program:

"1. Read in the first item and store it in address 2^{n-1}, i.e., at the halfway storage place.

"2. Read in the next item. Compare it with the first.

"3. If it is larger, put it in address $2^{n-1} + 2^{n-2}$. If it is smaller, put it at 2^{n-2}. . . ."

Another early form of tree insertion was introduced by D. J. Wheeler, who actually allowed multiway branching similar to what we shall discuss in Section 6.2.4; and a binary tree insertion technique was also independently devised by C. M. Berners-Lee [see *Comp. J.* **2** (1959), 5].

The first published descriptions of tree insertion were by P. F. Windley [*Comp. J.* **3** (1960), 84–88], A. D. Booth and A. J. T. Colin [*Information and Control* **3** (1960), 327–334], and Thomas N. Hibbard [*JACM* **9** (1962), 13–28]. All three of these authors seem to have developed the method independently of one another, and all three authors gave somewhat different proofs of the average number of comparisons (6). The three authors also went on to treat different aspects of the algorithm: Windley gave a detailed discussion of tree insertion sorting; Booth and Colin discussed the effect of preconditioning by making the first $2^n - 1$ elements form a perfectly balanced tree (see exercise 4); Hibbard introduced the idea of deletion and showed the connection between the analysis of tree insertion and the analysis of quicksort.

The idea of *optimum* binary search trees was first developed for the special case $p_1 = \cdots = p_n = 0$, in the context of alphabetic binary encodings like (24). A very interesting paper by E. N. Gilbert and E. F. Moore [*Bell System Tech. J.* **38** (1959), 933–968] discussed this problem and its relation to other coding problems. Gilbert and Moore observed, among other things, that an optimum tree could be constructed in $O(n^3)$ steps, using a method like Algorithm K but without the monotonicity relation (17). K. E. Iverson [*A Programming Language* (Wiley, 1962), 142–144] independently considered the *other* case, when all the q's are zero. He suggested that an optimum tree would be obtained if the root is chosen so as to equalize the left and right subtree probabilities as much as possible; unfortunately we have seen that this idea doesn't work. D. E. Knuth [*Acta Informatica* **1** (1971), 14–25, 270] subsequently considered the case of general p and q weights and proved that the algorithm could be reduced to $O(n^2)$ steps; he also presented an example from a compiler application, where the keys in the tree are "reserved words" in an ALGOL-like language. T. C. Hu had been studying his own algorithm for the $p = 0$ case for several years; a rigorous proof of the validity of that algorithm was difficult to find because of the complexity of the problem, but he eventually obtained a proof jointly with A. C. Tucker in 1969 [*SIAM J. Applied Math.* **21** (1971), 514–532].

EXERCISES

1. [*15*] Algorithm T has been stated only for nonempty trees. What changes should be made so that it works properly for the empty tree too?

2. [*2*] Modify Algorithm T so that it works with *right-threaded* trees. (Cf. Section 2.3.1; symmetric traversal is easier in such trees.)

▸ **3.** [*20*] In Section 6.1 we saw that a slight change to the sequential search Algorithm 6.1S made it faster (Algorithm 6.1Q). Can a similar trick be used to speed up Algorithm T?

4. [*M24*] (A. D. Booth and A. J. T. Colin.) Given N keys in random order, suppose that we use the first $2^n - 1$ to construct a perfectly balanced tree, placing 2^k keys on level k for $0 \leq k < n$; then we use Algorithm T to insert the remaining keys. What is the average number of comparisons in a successful search? [*Hint:* Modify Eq. (2).]

▸ **5.** [*M25*] There are 11! = 39,916,800 different orders in which the names CAPRICORN, AQUARIUS, etc. could have been inserted into a binary search tree. (a) How many of these arrangements will produce Fig. 10? (b) How many of these arrangements will produce a degenerate tree, in which LLINK or RLINK is Λ in each node?

6. [*M26*] Let P_{nk} be the number of permutations $a_1a_2 \ldots a_n$ of $\{1, 2, \ldots, n\}$ such that, if Algorithm T is used to insert $a_1, a_2, \ldots, a_n$ successively into an initially empty tree, exactly k comparisons are made when a_n is inserted. (In this problem, we will ignore the comparisons made when $a_1, \ldots, a_{n-1}$ were inserted. In the notation of the text, we have $C'_{n-1} = (\sum kP_{nk})/n!$, since this is the average number of comparisons made in an unsuccessful search of a tree containing $n - 1$ elements.)

a) Prove that $P_{n+1,k} = 2P_{n,k-1} + (n - 1)P_{n,k}$. [*Hint:* Consider whether or not a_{n+1} falls below a_n in the tree.]

b) Find a simple formula for the generating function $G_n(z) = \sum_k P_{nk}z^k$, and use your formula to express P_{nk} in terms of Stirling numbers.

c) What is the *variance* of C'_{n-1}?

7. [*M30*] (S. R. Arora and W. T. Dent.) After n elements have been inserted into an initially empty tree, in random order, what is the average number of comparisons needed to find the mth largest element?

8. [*M38*] Let $p(n, k)$ be the probability that k is the total internal path length of a tree built by Algorithm T from n randomly ordered keys. (The internal path length is the number of comparisons made by tree insertion sorting as the tree is being built.) (a) Find a recurrence relation which defines the corresponding generating function. (b) Compute the variance of this distribution. [Several of the exercises in Section 1.2.7 may be helpful here!]

9. [*41*] We have proved that tree search and insertion requires only about $2 \ln N$ comparisons when the keys are inserted in random order; but in practice, the order may not be random. Make empirical studies to see how suitable tree insertion really is for symbol tables within a compiler and/or assembler. Do the identifiers used in typical large programs lead to fairly well-balanced binary search trees?

▸ **10.** [*22*] Suppose that a programmer is not interested in the sorting property of this algorithm, but he expects the input will come in with nonrandom order. Discuss methods by which he can still use the tree search by making the input "appear to be" in random order.

11. [*20*] What is the maximum number of times S ← LLINK(R) can be performed in step D3 when deleting a node from a tree of size N?

12. [*M22*] When making a random deletion from a random tree of N items, how often does step D1 go to D4, on the average? (See the proof of Theorem H.)

▸ **13.** [*M23*] If the root of a random tree is deleted by Algorithm D, is the resulting tree still random?

▸ **14.** [*22*] Prove that the path length of the tree produced by Algorithm D with step D1½ added is never more than the path length of the tree produced without that step. Find a case where step D1½ actually decreases the path length.

15. [*M47*] When the new step D1½ added to Algorithm D, exactly how much is the average running time of Algorithm T affected after a long sequence of insertions and deletions?

▶**16.** [*25*] Is the deletion operation *commutative*? That is, if Algorithm D is used to delete X and then Y, is the resulting tree the same as if Algorithm D is used to delete Y and then X?

17. [*25*] Show that if the roles of left and right are completely reversed in Algorithm D, it is easy to extend the algorithm so that it deletes a given node from a *right-threaded* tree, preserving the necessary threads. (Cf. exercise 2.)

18. [*M21*] Show that Zipf's law yields (12).

19. [*M23*] What is the approximate average number of comparisons, (11), when the input probabilities satisfy the "80-20" law defined in Eqs. 6.1–11, 12?

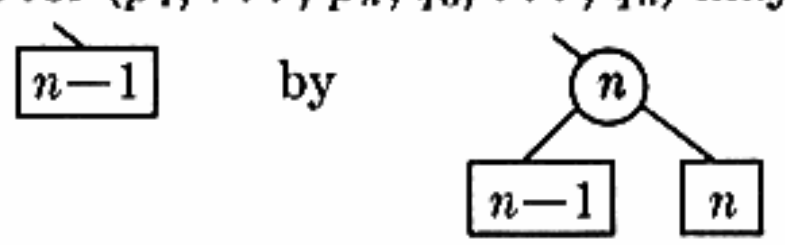
20. [*M20*] Suppose we have inserted keys into a tree in order of decreasing frequency $p_1 \geq p_2 \geq \cdots \geq p_N$. Can this tree be substantially worse than the optimum search tree?

21. [*M20*] If p, q, r are probabilities chosen at random, subject to the condition that $p + q + r = 1$, what are the probabilities that trees I, II, III, IV, V of (13) are optimal, respectively? (Consider the relative areas of the regions in Fig. 14.)

22. [*M20*] Prove that $r[i, j-1]$ is never greater than $r[i+1, j]$ when step K4 of Algorithm K is performed.

▶**23.** [*M23*] Find an optimum binary search tree for the case $n = 40$, with weights $p_1 = 5$, $p_2 = p_3 = \cdots = p_{40} = 1$, $q_0 = q_1 = \cdots = q_{40} = 0$. (Don't use a computer.)

24. [*M25*] Given that $p_n = q_n = 0$ and that the other weights are nonnegative, prove that an optimum tree for $(p_1, \ldots, p_n; q_0, \ldots, q_n)$ may be obtained by replacing



in any optimum tree for $(p_1, \ldots, p_{n-1}; q_0, \ldots, q_{n-1})$.

25. [*M20*] Let A and B be nonempty sets of real numbers, and define $A \leq B$ if the following property holds:

$$(a \in A, \quad b \in B, \quad \text{and} \quad b < a) \qquad \text{implies} \qquad (a \in B \quad \text{and} \quad b \in A).$$

(a) Prove that this relation is transitive on nonempty sets. (b) Prove or disprove: $A \leq B$ if and only if $A \leq A \cup B \leq B$.

26. [*M22*] Let $(p_1, \ldots, p_n; q_0, \ldots, q_n)$ be nonnegative weights, where $p_n + q_n = x$. Prove that as x varies from 0 to ∞, while $(p_1, \ldots, p_{n-1}; q_0, \ldots, q_{n-1})$ are held constant, the cost $c(0, n)$ of an optimum binary search tree is a "convex, continuous, piece-wise linear" function of x with integer slopes. In other words, prove that there exist positive integers $l_0 > l_1 > \cdots > l_m$ and real constants $0 = x_0 < x_1 < \cdots < x_m < x_{m+1} = \infty$ and $y_0 < y_1 < \cdots < y_m$ such that $c(0, n) = y_h + l_h x$ when $x_h \leq x \leq x_{h+1}$, for $0 \leq h \leq m$.

27. [*M33*] The object of this exercise is to prove that the sets of roots $R(i, j)$ of optimum binary search trees satisfy

$$R(i, j-1) \leq R(i, j) \leq R(i+1, j), \qquad \text{for} \qquad j - i \geq 2,$$

in terms of the relation defined in exercise 25, whenever the weights $(p_1, \ldots, p_n; q_0, \ldots, q_n)$ are nonnegative. The proof is by induction on $j - i$; our task is to prove

that $R(0, n-1) \le R(0, n)$, assuming that $n \ge 2$ and that the above relation holds for $j - i < n$. [By left-right symmetry it follows that $R(0, n) \le R(1, n)$.]

a) Prove that $R(0, n-1) \le R(0, n)$ if $p_n = q_n = 0$. (See exercise 24.)

b) Let $p_n + q_n = x$. In the notation of exercise 26, let R_h be the set $R(0, n)$ of optimum roots when $x_h < x < x_{h+1}$, and let R'_h be the set of optimum roots when $x = x_h$. Prove that

$$R'_0 \le R_0 \le R'_1 \le R_1 \le \cdots \le R'_m \le R_m.$$

Hence by part (a) and exercise 25 we have $R(0, n-1) \le R(0, n)$ for all x.
[*Hint:* Consider the case $x = x_h$, and assume that both the trees

are optimum, with $s < r$ and $l \ge l'$. Use the induction hypothesis to prove that there is an optimum tree with root (r) such that $\boxed{n}$ is at level l', and an optimum tree with root (s) such that $\boxed{n}$ is at level l.]

28. [*24*] Use some macro-assembly language to define a "optimum binary search" macro, whose parameter is a nested specification of an optimum binary tree.

29. [*40*] What is the *worst* possible binary search tree for the 31 most common English words, using the frequency data of Fig. 12?

30. [*M46*] Prove or disprove that the costs of optimum binary search trees satisfy $c(i, j) + c(i+1, j-1) \ge c(i, j-1) + c(i+1, j)$.

31. [*M20*] (a) If the weights $(q_0, \ldots, q_5)$ in (22) are (2, 3, 1, 1, 3, 2), respectively, what is the weighted path length of the tree? (b) What is the weighted path length of the *optimum* binary search tree having this sequence of weights?

▸ **32.** [*M22*] (T. C. Hu and A. C. Tucker.) Prove that the new node weights $q_i + q_j$ formed during Phase 1 of the Hu-Tucker algorithm are created in nondecreasing order.

33. [*M41*] In order to find the binary search tree which minimizes the running time of Program T, we should minimize the quantity $7C + C1$ instead of simply minimizing the number of comparisons C. Develop an algorithm which finds optimum binary search trees when different costs are associated with left and right branches in the tree. (Incidentally when the right cost is twice the left cost, and the node frequencies are all equal, the Fibonacci trees turn out to be optimum. Cf. L. E. Stanfel, *JACM* **17** (1970), 508–517.)

34. [*41*] Write a computer program for the Hu-Tucker algorithm, using $O(n)$ units of storage and $O(n \log n)$ units of time.

35. [*M23*] Show that the codewords constructed in the proof of Theorem G have the property that C_i never begins with C_j when $i \ne j$.

36. [*M26*] Generalizing the upper bound of Theorem G, prove that the cost of any optimum binary search tree with nonnegative weights must be less than

$$2S + q_0 \log_2 (S/q_0) + \sum_{1 \le i \le n} (p_i + q_i) \log_2 (S/(p_i + q_i)),$$

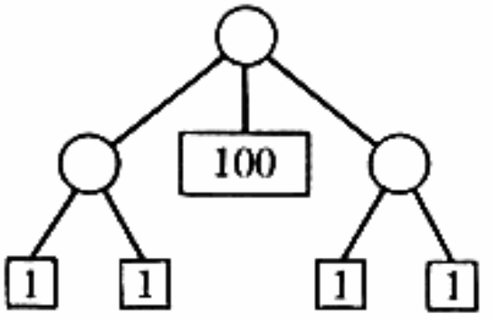
where $S = q_0 + \sum_{1 \le i \le n} (p_i + q_i)$ is the total weight.

▸ **37.** [*M25*] (T. C. Hu and K. C. Tan.) Let $n+1 = 2^m + k$, where $0 \le k \le 2^m$. There are $\binom{2^m}{k}$ binary trees in which all external nodes appear on levels m and $m+1$. Show that, among all these trees, we obtain one with the minimum weighted path length for the weight sequence $(q_0, \ldots, q_n)$ if we apply the Hu-Tucker algorithm to the weights $(M + q_0, \ldots, M + q_n)$ for sufficiently large M.

38. [*M35*] (K. C. Tan.) Prove that, among all sets of probabilities $(p_1, \ldots, p_n; q_0, \ldots, q_n)$ with $p_1 + \cdots + p_n + q_0 + \cdots + q_n = 1$, the most expensive minimum-cost tree occurs when $p_i = 0$ for all i, $q_j = 0$ for all even j, and $q_j = 1/\lceil n/2 \rceil$ for all odd j. [*Hint:* Given arbitrary probabilities $(p_1, \ldots, p_n; q_0, \ldots, q_n)$, let $c_0 = q_0$, $c_i = p_i + q_i$ for $1 \le i \le n$, and $S(0) = \emptyset$; and for $1 \le r \le \lceil n/2 \rceil$ let $S(r) = S(r-1) \cup \{i, j\}$, where $c_i + c_j$ is minimum over all $i < j$ such that $i, j \notin S(r-1)$ and $k \in S(r-1)$ for all $i < k < j$. Construct the binary tree T with the external nodes of $S(n + 1 - 2^q)$ on level $q+1$ and the other external nodes on level q, where $q = \lfloor \log_2 n \rfloor$. Prove that the cost of this tree is $\le f(n)$, where $f(n)$ is the cost of the optimum search tree for the stated "worst" probabilities.]

39. [*M30*] (C. K. Wong and Shi-Kuo Chang.) Consider a scheme whereby a binary search tree is constructed by Algorithm T, except that whenever the number of nodes reaches a number of the form $2^n - 1$ the tree is reorganized into a perfectly-balanced uniform tree, with 2^k nodes on level k for $0 \le k < n$. Prove that the number of comparisons made while constructing such a tree is $N \log_2 N + O(N)$ on the average. (It is not difficult to show that the amount of time needed for the reorganizations is $O(N)$.)

40. [*M50*] Can the Hu-Tucker algorithm be generalized to find optimum trees where each node has degree at most t? For example, when $t = 3$ and the sequence of weights is (1, 1, 100, 1, 1) the optimum tree is



6.2.3. Balanced Trees

The tree insertion algorithm we have just learned will produce good search trees, when the input data is random, but there is still the annoying possibility that a degenerate tree will occur. Perhaps we could devise an algorithm which keeps the tree optimum at all times; but unfortunately that seems to be very difficult. Another idea is to keep track of the total path length, and to completely reorganize the tree whenever its path length exceeds $5N \log_2 N$, say. But this approach might require about $\sqrt{N/2}$ reorganizations as the tree is being built.

A very pretty solution to the problem of maintaining a good search tree was discovered in 1962 by two Russian mathematicians, G. M. Adel'son-Vel'skiĭ and E. M. Landis [*Doklady Akademiia Nauk SSSR* **146** (1962), 263–266; English translation in *Soviet Math.* **3**, 1259–1263]. Their method requires only two extra bits per node, and it never uses more than $O(\log N)$ operations to search

the tree or to insert an item. In fact, we shall see that their approach also leads to a general technique that is good for representing arbitrary *linear lists* of length N, so that each of the following operations can be done in only $O(\log N)$ units of time:

i) Find an item having a given key.
ii) Find the kth item, given k.
iii) Insert an item at a specified place.
iv) Delete a specified item.

If we use sequential allocation for linear lists, operations (i) and (ii) are efficient but operations (iii) and (iv) take order N steps; on the other hand, if we use linked allocation, operations (iii) and (iv) are efficient but operations (i) and (ii) take order N steps. A tree representation of linear lists can do *all four* operations in $O(\log N)$ steps. And it is also possible to do other standard operations with comparable efficiency, so that, for example, we can concatenate a list of M elements with a list of N elements in $O(\log(M + N))$ steps.

The method for achieving all this involves what we shall call "balanced trees." The preceding paragraph is an advertisement for balanced trees, which makes them sound like a universal panacea that makes all other forms of data representation obsolete; but of course we ought to have a balanced attitude about balanced trees! In applications which do not involve all four of the above operations, we may be able to get by with substantially less overhead and simpler programming. Furthermore, there is no advantage to balanced trees unless N is reasonably large; thus if we have an efficient method that takes $20 \log_2 N$ units of time and an inefficient method that takes $2N$ units of time, we should use the inefficient method unless N is greater than 1024. On the other hand, N shouldn't be too large, either; balanced trees are appropriate chiefly for *internal* storage of data, and we shall study better methods for external direct-access files in Section 6.2.4. Since internal memories seem to be getting larger and larger as time goes by, balanced trees are becoming more and more important.

The *height* of a tree is defined to be its maximum level, the length of the longest path from the root to an external node. A binary tree is called *balanced* if the height of the left subtree of every node never differs by more than ± 1 from the height of its right subtree. Figure 20 shows a balanced tree with 17 internal nodes and height 5; the *balance factor* within each node is shown as $+$, $\cdot$, or $-$ according as the right subtree height minus the left subtree height is $+1$, 0, or -1. The Fibonacci tree in Fig. 8 (Section 6.2.1) is another balanced binary tree of height 5, having only 12 internal nodes; most of the balance factors in that tree are -1. The zodiac tree in Fig. 10 (Section 6.2.2) is *not* balanced, because the height restriction on subtrees fails at both the `AQUARIUS` and `GEMINI` nodes.

This definition of balance represents a compromise between *optimum* binary trees (with all external nodes required to be on two adjacent levels) and *arbitrary*

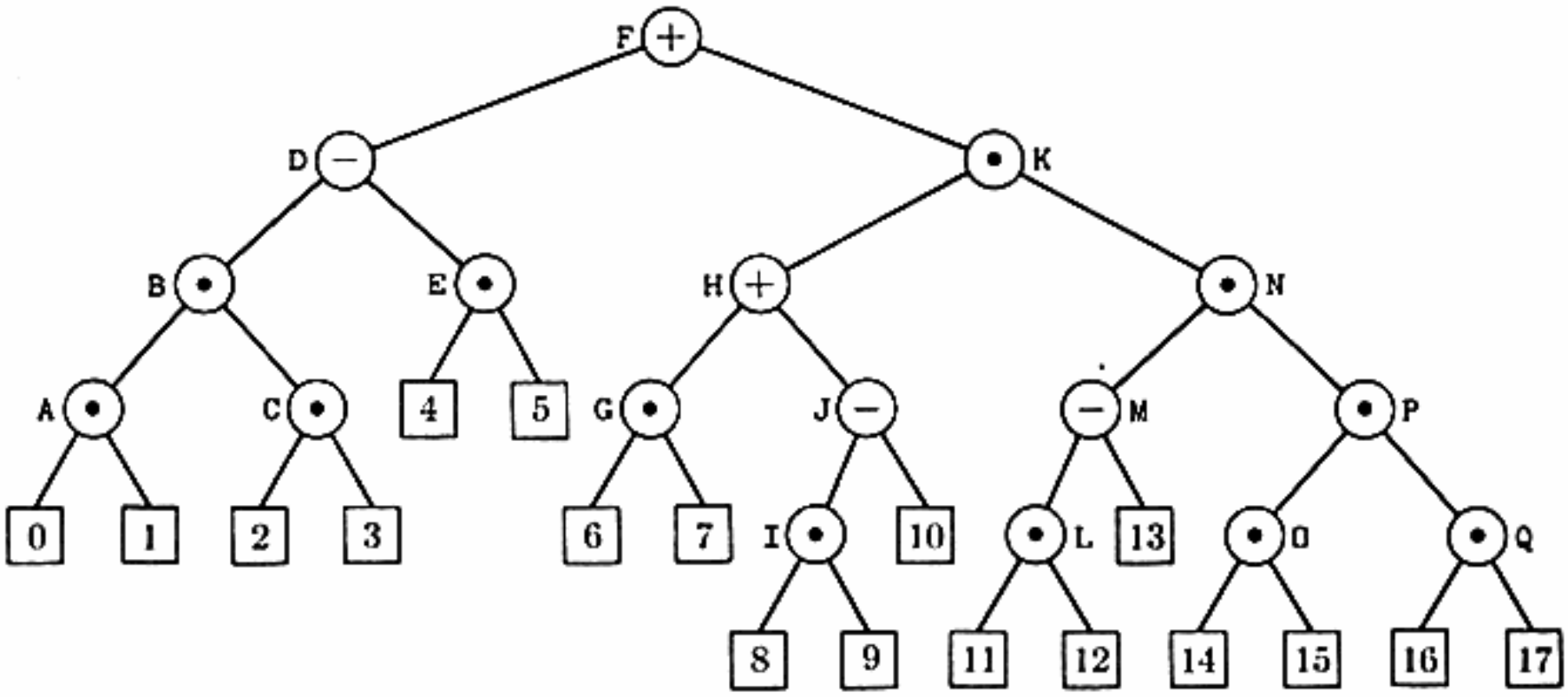


Fig. 20. A balanced binary tree.

binary trees (unrestricted). It is therefore natural to ask how far from optimum a balanced tree can be. The answer is that its search paths will never be more than 45 percent longer than the optimum:

Theorem A (Adel'son-Vel'skiĭ and Landis). *The height of a balanced tree with* N *internal nodes always lies between* $\log_2 (N + 1)$ *and* $1.4404 \log_2 (N + 2) - 0.328$.

Proof. A binary tree of height h obviously cannot have more than 2^h external nodes; so $N + 1 \leq 2^h$, that is, $h \geq \lceil \log_2 (N + 1) \rceil$ in any binary tree.

In order to find the maximum value of h, let us turn the problem around and ask for the minimum number of nodes possible in a balanced tree of height h. Let T_h be such a tree with fewest possible nodes; then one of the subtrees of the root, say the left subtree, has height $h - 1$, and the other subtree has height $h - 1$ or $h - 2$. Since we want T_h to have the minimum number of nodes, we may assume that the left subtree of the root is T_{h-1}, and that the right subtree is T_{h-2}. This argument shows that the *Fibonacci tree* of order $h + 1$ has the fewest possible nodes among all possible balanced trees of height h. (See the definition of Fibonacci trees in Section 6.2.1.) Thus

$$N \geq F_{h+2} - 1 > \phi^{h+2}/\sqrt{5} - 2,$$

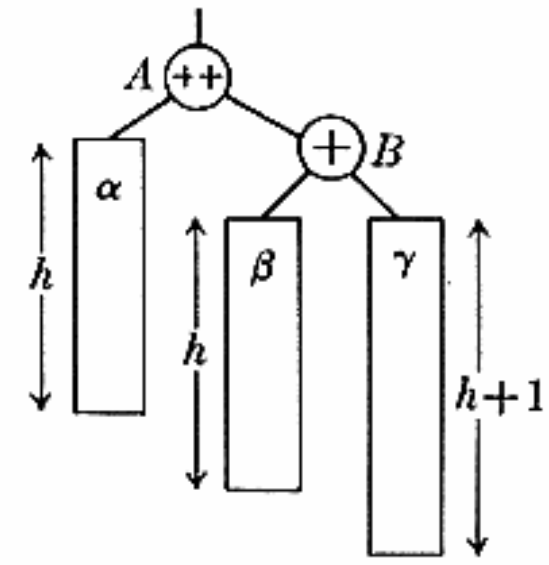
and the stated result follows as in the corollary to Theorem 4.5.3F. ∎

The proof of this theorem shows that a search in a balanced tree will require more than 25 comparisons only if the tree contains at least $F_{27} - 1 = 196{,}417$ nodes.

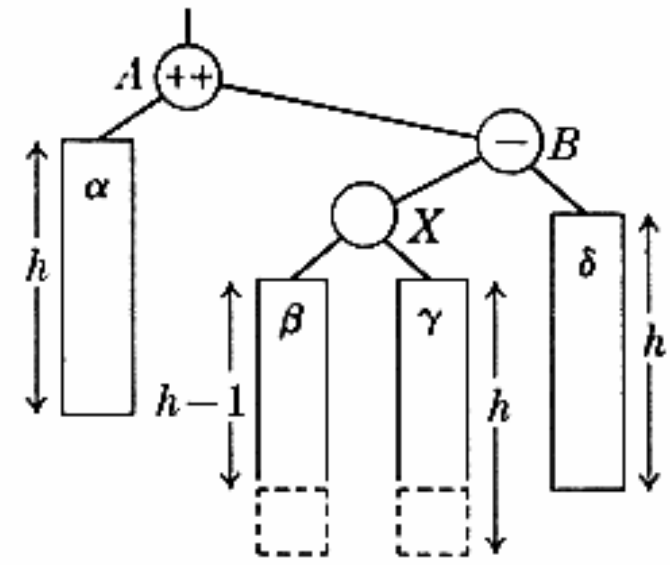
Consider now what happens when a new node is inserted into a balanced tree using tree insertion (Algorithm 6.2.2T). In Fig. 20, the tree will still be balanced if the new node takes the place of [4], [5], [6], [7], [10], or [13], but some adjustment will be needed if the new node falls elsewhere. The problem arises when we have a node with a balance factor of +1 whose right subtree got higher after the insertion; or, dually, if the balance factor is −1 and the left

subtree got higher. It is not difficult to see that there are essentially only two cases which cause trouble:

Case 1



Case 2

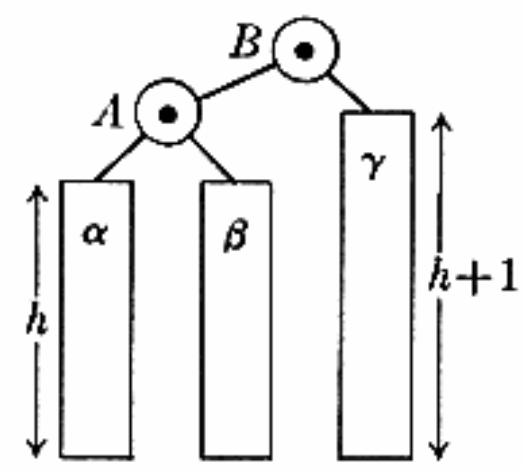


(1)

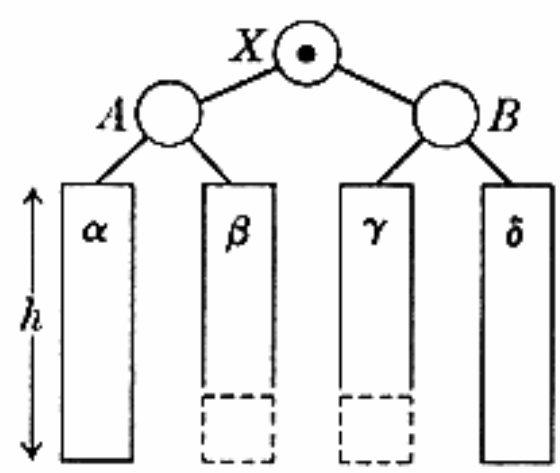
(Two other essentially identical cases occur if we reflect these diagrams, interchanging left and right.) In these diagrams the large rectangles α, β, γ, δ represent subtrees having the respective heights shown. Case 1 occurs when a new element has just increased the height of node B's right subtree from h to $h + 1$, and Case 2 occurs when the new element has increased the height of B's left subtree. In the second case, we have either $h = 0$ (so that X itself was the new node), or else node X has two subtrees of respective heights $(h - 1, h)$ or $(h, h - 1)$.

Simple transformations will restore balance in both of the above cases, while preserving the symmetric order of the tree nodes:

Case 1



Case 2



(2)

In Case 1 we simply "rotate" the tree to the left, attaching β to A instead of B. This transformation is like applying the associative law to an algebraic formula, replacing $\alpha(\beta\gamma)$ by $(\alpha\beta)\gamma$. In Case 2 we use a double rotation, first rotating (X, B) right, then (A, X) left. In both cases only a few links of the tree need to be changed. Furthermore, the new trees have height $h + 2$, which is exactly the height that was present before the insertion; hence the rest of the tree (if any) that was originally above node A always remains balanced.

For example, if we insert a new node into position 17 of Fig. 20 we obtain the balanced tree shown in Fig. 21, after a single rotation (Case 1). Note that several of the balance factors have changed.

The details of this insertion procedure can be worked out in several ways. At first glance an auxiliary stack seems to be necessary, in order to keep track of which nodes will be affected, but the following algorithm gains some speed by avoiding the need for a stack in a slightly tricky way.

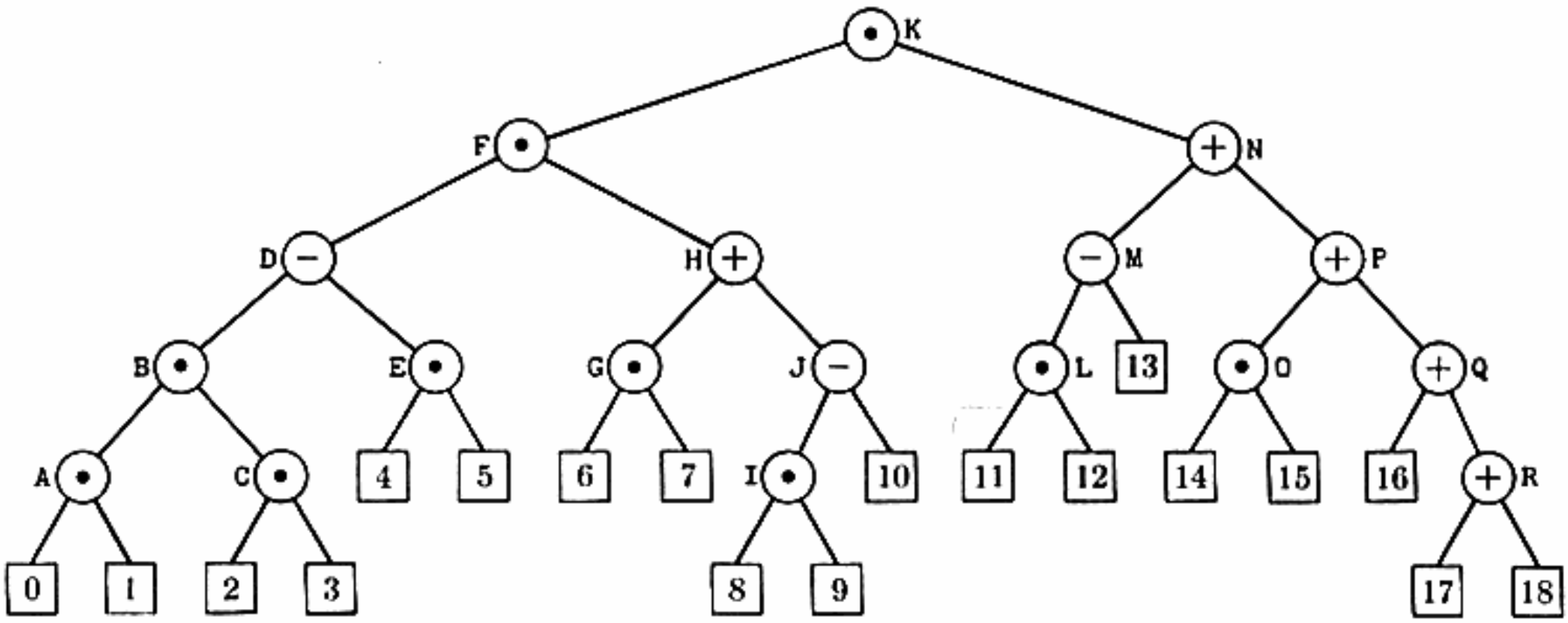


Fig. 21. The tree of Fig. 20, rebalanced after a new key R has been inserted.

Algorithm A (*Balanced tree search and insertion*). Given a table of records which form a balanced binary tree as described above, this algorithm searches for a given argument K. If K is not in the table, a new node containing K is inserted into the tree in the appropriate place and the tree is rebalanced if necessary.

The nodes of the tree are assumed to contain KEY, LLINK, and RLINK fields as in Algorithm 6.2.2T. We also have a new field

$$\texttt{B(P)} = \text{balance factor of } \texttt{NODE(P)},$$

the height of the right subtree minus the height of the left subtree; this field always contains either $+1$, 0 or -1. A special header node also appears at the top of the tree, in location HEAD; the value of RLINK(HEAD) is a pointer to the root of the tree, and LLINK(HEAD) is used to keep track of the overall height of the tree. (Knowledge of the height is not really necessary for this algorithm, but it is useful in the concatenation procedure discussed below.) We assume that the tree is *nonempty*, i.e., that $\texttt{RLINK(HEAD)} \neq \Lambda$.

For convenience in description, the algorithm uses the notation LINK(a, P) as a synonym for LLINK(P) if $a = -1$, and for RLINK(P) if $a = +1$.

A1. [Initialize.] Set T $\leftarrow$ HEAD, S $\leftarrow$ P $\leftarrow$ RLINK(HEAD). (The pointer variable P will move down the tree; S will point to the place where rebalancing may be necessary, and T always points to the father of S.)

A2. [Compare.] If $K <$ KEY(P), go to A3; if $K >$ KEY(P), go to A4; and if $K =$ KEY(P), the search terminates successfully.

A3. [Move left.] Set Q $\leftarrow$ LLINK(P). If Q $= \Lambda$, set Q $\Leftarrow$ AVAIL and LLINK(P) $\leftarrow$ Q and go to step A5. Otherwise if B(Q) $\neq 0$, set T $\leftarrow$ P and S $\leftarrow$ Q. Finally set P $\leftarrow$ Q and return to step A2.

A4. [Move right.] Set Q $\leftarrow$ RLINK(P). If Q $= \Lambda$, set Q $\Leftarrow$ AVAIL and RLINK(P) $\leftarrow$ Q and go to step A5. Otherwise if B(Q) $\neq 0$, set T $\leftarrow$ P and S $\leftarrow$ Q. Finally set P $\leftarrow$ Q and return to step A2. (The last part of this step may be combined with the last part of step A3.)

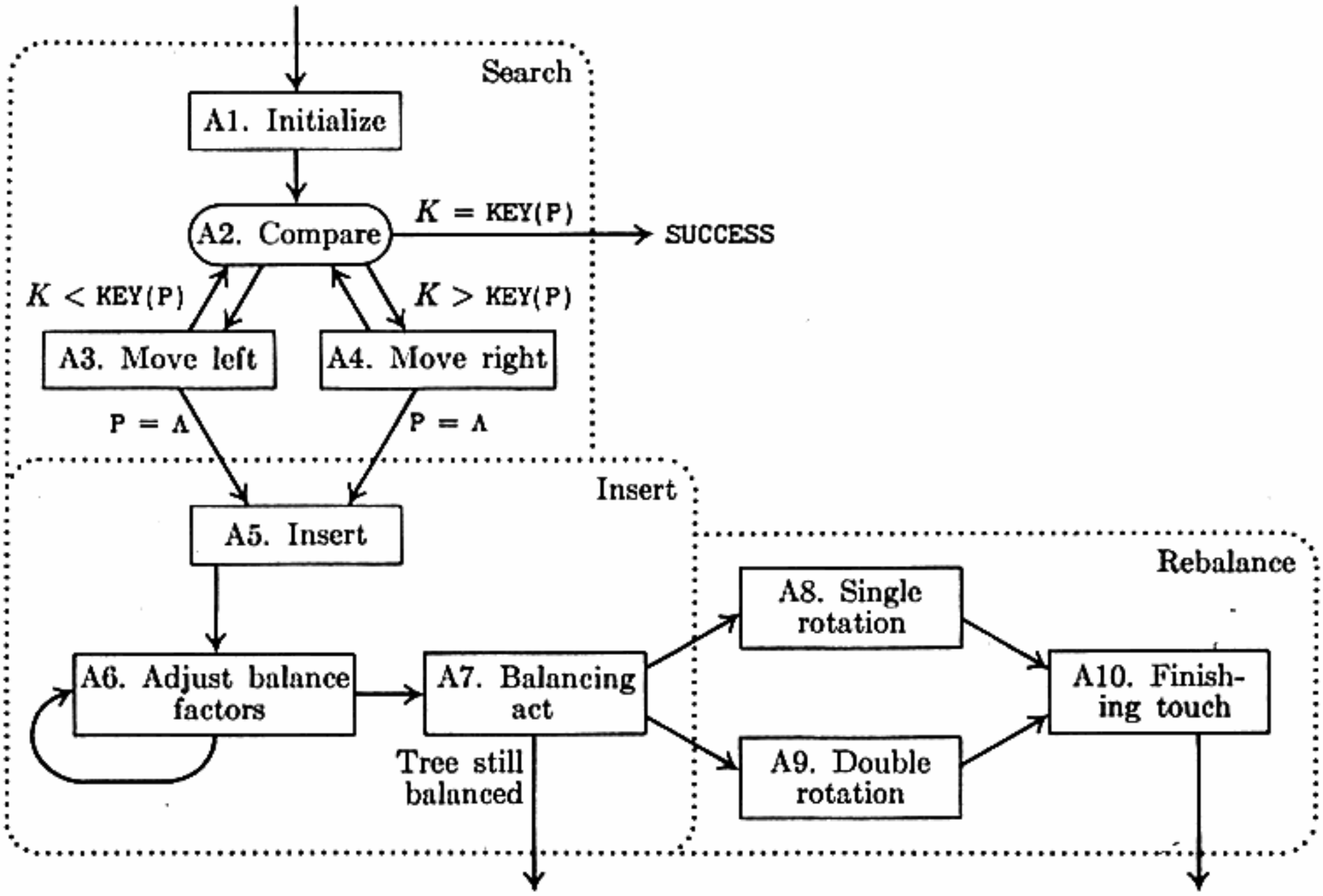


Fig. 22. Balanced tree search and insertion.

A5. [Insert.] (We have just linked a new node, NODE(Q), into the tree, and its fields need to be initialized.) Set KEY(Q) ← K, LLINK(Q) ← RLINK(Q) ← Λ, B(Q) ← 0.

A6. [Adjust balance factors.] (Now the balance factors on nodes between S and Q need to be changed from zero to ±1.) If $K <$ KEY(S), set R ← P ← LLINK(S), otherwise set R ← P ← RLINK(S). Then repeatedly do the following operation zero or more times until P = Q: If $K <$ KEY(P) set B(P) ← −1 and P ← LLINK(P); if $K >$ KEY(P), set B(P) ← +1 and P ← RLINK(P). (If $K =$ KEY(P), then P = Q and we may go on to the next step.)

A7. [Balancing act.] If $K <$ KEY(S) set $a \leftarrow -1$, otherwise set $a \leftarrow +1$. Several cases now arise:

i) If B(S) = 0 (the tree has grown higher), set B(S) ← a, LLINK(HEAD) ← LLINK(HEAD) + 1, and terminate the algorithm.

ii) If B(S) = $-a$ (the tree has gotten more balanced), set B(S) ← 0 and terminate the algorithm.

iii) If B(S) = a (the tree has gotten out of balance), go to step A8 if B(R) = a, to A9 if B(R) = $-a$.

(Case (iii) corresponds to the situations depicted in (1) when $a = +1$; S and R point, respectively, to nodes A and B, and LINK($-a$, S) points to α, etc.)

A8. [Single rotation.] Set P ← R, LINK(a, S) ← LINK($-a$, R), LINK($-a$, R) ← S, B(S) ← B(R) ← 0. Go to A10.

A9. [Double rotation.] Set P ← LINK($-a$, R), LINK($-a$ R) ← LINK(a, P), LINK(a, P) ← R, LINK(a, S) ← LINK($-a$, P), LINK($-a$, P) ← S. Now set

$$(\mathtt{B(S)}, \mathtt{B(R)}) \leftarrow \begin{cases} (-a, 0), & \text{if } \mathtt{B(P)} = a; \\ (0, 0), & \text{if } \mathtt{B(P)} = 0; \\ (0, a), & \text{if } \mathtt{B(P)} = -a; \end{cases} \tag{3}$$

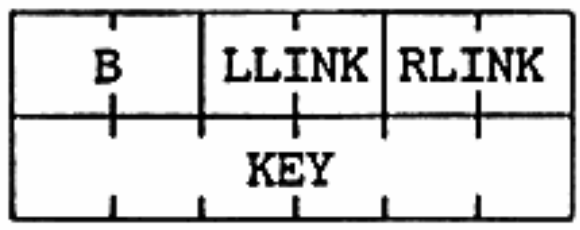
and then set B(P) ← 0.

A10. [Finishing touch.] (We have completed the rebalancing transformation, taking (1) to (2), with P pointing to the new root and T pointing to the father of the old root.) If S = RLINK(T) then set RLINK(T) ← P, otherwise set LLINK(T) ← P. ∎

This algorithm is rather long, but it divides into three simple parts: Steps A1–A4 do the search, steps A5–A7 insert a new node, and steps A8–A10 rebalance the tree if necessary.

We know that the algorithm takes about $C \log N$ units of time, for some C, but it is important to know the approximate value of C so that we can tell how large N should be in order to make balanced trees worth all the trouble. The following MIX implementation gives some insight into this question.

Program A (*Balanced tree search and insertion*). This program for Algorithm A uses tree nodes having the form



; (4)

rA ≡ K, rI1 ≡ P, rI2 ≡ Q, rI3 ≡ R, rI4 ≡ S, rI5 ≡ T. The code for steps A7–A9 is duplicated so that the value of a appears implicitly (not explicitly) in the program.

```
01    B       EQU   0:1
02    LLINK   EQU   2:3
03    RLINK   EQU   4:5
04    START   LDA   K                  1        A1. Initialize.
05            ENT5  HEAD                 1      T ← HEAD.
06            LD2   0,5(RLINK)           1      Q ← RLINK(HEAD).
07            JMP   2F                   1      To A2 with S ← P ← Q.
08    4H      LD2   0,1(RLINK)          C2      A4. Move right. Q ← RLINK(P).
09            J2Z   5F                  C2      To A5 if Q = Λ.
10    1H      LDX   0,2(B)           C − 1      rX ← B(Q).
11            JXZ   *+3              C − 1      Jump if B(Q) = 0.
12            ENT5  0,1              D − 1      T ← P.
13    2H      ENT4  0,2                 D       S ← Q.
14            ENT1  0,2                 C       P ← Q.
15            CMPA  1,1                 C       A2. Compare.
16            JG    4B                  C       To A4 if K > KEY(P).
17            JE    SUCCESS             C1      Exit if K = KEY(P).
18            LD2   0,1(LLINK)       C1 − S      A3. Move left. Q ← LLINK(P).
19            J2NZ  1B               C1 − S      Jump if Q ≠ Λ.
20–29   5H (copy here lines 14–23 of Program 6.2.2T)   A5. Insert.
```

```
30          6H      CMPA    1,4             1 − S       A6. Adjust balance factors.
31                  JL      *+3             1 − S       Jump if K < KEY(S).
32                  LD3     0,4(RLINK)        E         R ← RLINK(S).
33                  JMP     *+2               E
34                  LD3     0,4(LLINK)    1 − S − E     R ← LLINK(S).
35                  ENT1    0,3             1 − S       P ← R.
36                  ENTX    -1              1 − S       rX ← −1.
37                  JMP     1F              1 − S       To comparison loop.
38          4H      JE      7F            F2 + 1 − S    To A7 if K = KEY(P).
39                  STX     0,1(1:1)          F2        B(P) ← +1 (it was +0).
40                  LD1     0,1(RLINK)        F2        P ← RLINK(P).
41          1H      CMPA    1,1           F + 1 − S
42                  JGE     4B            F + 1 − S     Jump if K ≥ KEY(P).
43                  STX     0,1(B)            F1        B(P) ← −1.
44                  LD1     0,1(LLINK)        F1        P ← LLINK(P).
45                  JMP     1B                F1        To comparison loop.
46          7H      LD2     0,4(B)          1 − S       A7. Balancing act. rI2 ← B(S).
47                  STZ     0,4(B)          1 − S       B(S) ← 0.
48                  CMPA    1,4             1 − S
49                  JG      A7R             1 − S       To a = +1 routine if K > KEY(S).

50      A7L     J2P     DONE            68      A7R     J2N     DONE            1 − S       Exit if rI2 = −a.
51              J2Z     7F              69              J2Z     6F              G + J       Jump if B(S) was zero.
52              ENT1    0,3             70              ENT1    0,3               G         P ← R.
53              LD2     0,3(B)          71              LD2     0,3(B)            G         rI2 ← B(R).
54              J2N     A8L             72              J2P     A8R               G         To A8 if rI2 = a.
55      A9L     LD1     0,3(RLINK)      73      A9R     LD1     0,3(LLINK)        H         A9. Double rotation.
56              LDX     0,1(LLINK)      74              LDX     0,1(RLINK)        H         LINK(a, P ← LINK(−a, R))
57              STX     0,3(RLINK)      75              STX     0,3(LLINK)        H         → LINK(−a, R).
58              ST3     0,1(LLINK)      76              ST3     0,1(RLINK)        H         LINK(a, P)← R.
59              LD2     0,1(B)          77              LD2     0,1(B)            H         rI2 ← B(P).
60              LDX     T1,2            78              LDX     T2,2              H         −a, 0, or 0
61              STX     0,4(B)          79              STX     0,4(B)            H         → B(S).
62              LDX     T2,2            80              LDX     T1,2              H         0, 0, or a
63              STX     0,3(B)          81              STX     0,3(B)            H         → B(R).
64      A8L     LDX     0,1(RLINK)      82      A8R     LDX     0,1(LLINK)        G         A8. Single rotation.
65              STX     0,4(LLINK)      83              STX     0,4(RLINK)        G         LINK(a, S) ← LINK(−a, P).
66              ST4     0,1(RLINK)      84              ST4     0,1(LLINK)        G         LINK(−a, P) ← S.
67              JMP     A8R1            85      A8R1    STZ     0,1(B)            G         B(P) ← 0.

86          A10     CMP4    0,5(RLINK)        G         A10. Finishing touch.
87                  JNE     *+3               G         Jump if RLINK(T) ≠ S.
88                  ST1     0,5(RLINK)        G2        RLINK(T) ← P.
89                  JMP     DONE              G2        Exit.
90                  ST1     0,5(LLINK)        G1        LLINK(T) ← P.
91                  JMP     DONE              G1        Exit.
92                  CON     +1
93          T1      CON     0                           Table for (3)
94          T2      CON     0
95                  CON     -1
96          6H      ENTX    +1                J2        rX ← +1.
97          7H      STX     0,4(B)            J         B(S) ← a.
98                  LDX     HEAD(LLINK)       J         LLINK(HEAD)
99                  INCX    1                 J         + 1
100                 STX     HEAD(LLINK)       J         → LLINK(HEAD).
101         DONE    EQU     *               1 − S       Insertion is complete. ∎
```

Analysis of balanced tree insertion. [Nonmathematical readers, please skip to (10).] In order to figure out the running time of Algorithm A, we would like to know the answers to the following questions:

- How many comparisons are made during the search?
- How far apart will nodes `S` and `Q` be? (In other words, how much adjustment is needed in step A6?)
- How often do we need to do a single or double rotation?

It is not difficult to derive upper bounds on the worst case running time, using Theorem A, but of course in practice we want to know the average behavior. No theoretical determination of the average behavior has been successfully completed as yet, since the algorithm appears to be rather complicated, but some interesting empirical results have been obtained.

In the first place we can ask about the number B_{nh} of balanced binary trees with n internal nodes and height h. It is not difficult to compute the generating function $B_h(z) = \sum_{n\geq 0} B_{nh} z^n$ for small h, from the relations

$$B_0(z) = 1, \qquad B_1(z) = z, \qquad B_{h+1}(z) = zB_h(z)\big(B_h(z) + 2B_{h-1}(z)\big). \tag{5}$$

(See exercise 6.) Thus

$$\begin{aligned} B_2(z) &= 2z^2 + z^3, \\ B_3(z) &= 4z^4 + 6z^5 + 4z^6 + z^7, \\ B_4(z) &= 16z^7 + 32z^8 + 44z^9 + \cdots + 8z^{14} + z^{15}, \end{aligned}$$

and in general $B_h(z)$ has the form

$$2^{F_{h+1}-1} z^{F_{h+2}-1} + 2^{F_{h+1}-2} L_{h-1} z^{F_{h+2}} + \text{complicated terms} + 2^{h-1} z^{2^h-2} + z^{2^h-1} \tag{6}$$

for $h \geq 3$, where $L_k = F_{k+1} + F_{k-1}$. (This formula generalizes Theorem A.) The total number of balanced trees with height h is $B_h = B_h(1)$, which satisfies the recurrence

$$B_0 = B_1 = 1, \qquad B_{h+1} = B_h^2 + 2B_hB_{h-1}, \tag{7}$$

so that $B_2 = 3$, $B_3 = 3 \cdot 5$, $B_4 = 3^2 \cdot 5 \cdot 7$, $B_5 = 3^3 \cdot 5^2 \cdot 7 \cdot 23$; and, in general,

$$B_h = A_0^{F_h} \cdot A_1^{F_{h-1}} \cdot \ldots \cdot A_{h-1}^{F_1} \cdot A_h^{F_0}, \tag{8}$$

where $A_0 = 1$, $A_1 = 3$, $A_2 = 5$, $A_3 = 7$, $A_4 = 23$, $A_5 = 347$, $\ldots$, $A_h = A_{h-1}B_{h-2} + 2$. The sequences B_h and A_h grow very rapidly, in fact, they are "doubly exponential": Exercise 7 shows that there is a real number $\theta \approx 1.43684$ such that

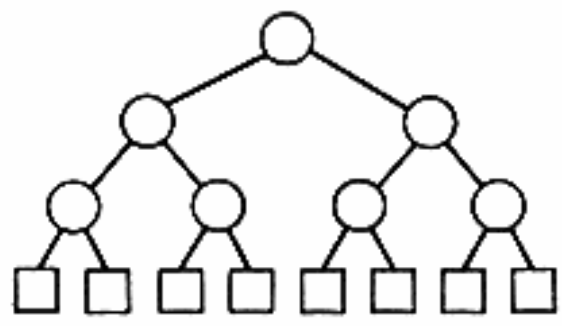
$$B_h = \lfloor \theta^{2^h} \rfloor - \lfloor \theta^{2^{h-1}} \rfloor + \lfloor \theta^{2^{h-2}} \rfloor - \cdots + (-1)^h \lfloor \theta^{2^0} \rfloor. \tag{9}$$

If we consider each of the B_h trees to be equally likely, exercise 8 shows that the average number of nodes in a tree of height h is

$$B_h'(1)/B_h(1) \approx (0.70118)2^h. \tag{10}$$

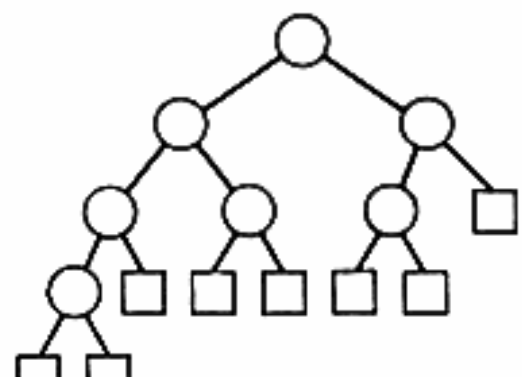
This indicates that the height of a balanced tree with n nodes usually is much closer to $\log_2 n$ than to $\log_\phi n$.

Unfortunately, none of these results have much to do with Algorithm A, since the mechanism of that algorithm makes some trees much more probable than others. For example, consider the case $N = 7$, where 17 balanced trees are possible. There are $7! = 5040$ possible orderings in which seven keys can be inserted, and the perfectly balanced "complete" tree,



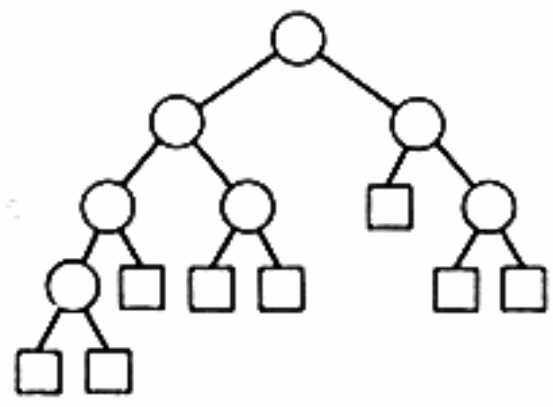
(11)

is obtained 2160 times. By contrast, the Fibonacci tree,



(12)

occurs only 144 times, and the similar tree,



(13)

occurs 216 times. [Replacing the left subtrees of (12) and (13) by arbitrary four-node balanced trees, and then reflecting left and right, yields 16 different trees; the eight generated from (12) each occur 144 times, and those generated from (13) each occur 216 times. It is somewhat surprising that (13) is more common than (12).]

The fact that the perfectly balanced tree is obtained with such high probability—together with (10), which corresponds to the case of equal probabilities—makes it extremely plausible that the average search time for a balanced tree is about $\log_2 N + c$ comparisons for some small constant c. Empirical tests support this conjecture: The average number of comparisons needed to insert the Nth item seems to be approximately $\log_2 N + 0.25$ for large N.

In order to study the behavior of the insertion and rebalancing phases of Algorithm A, we can classify the external nodes of balanced trees as shown in

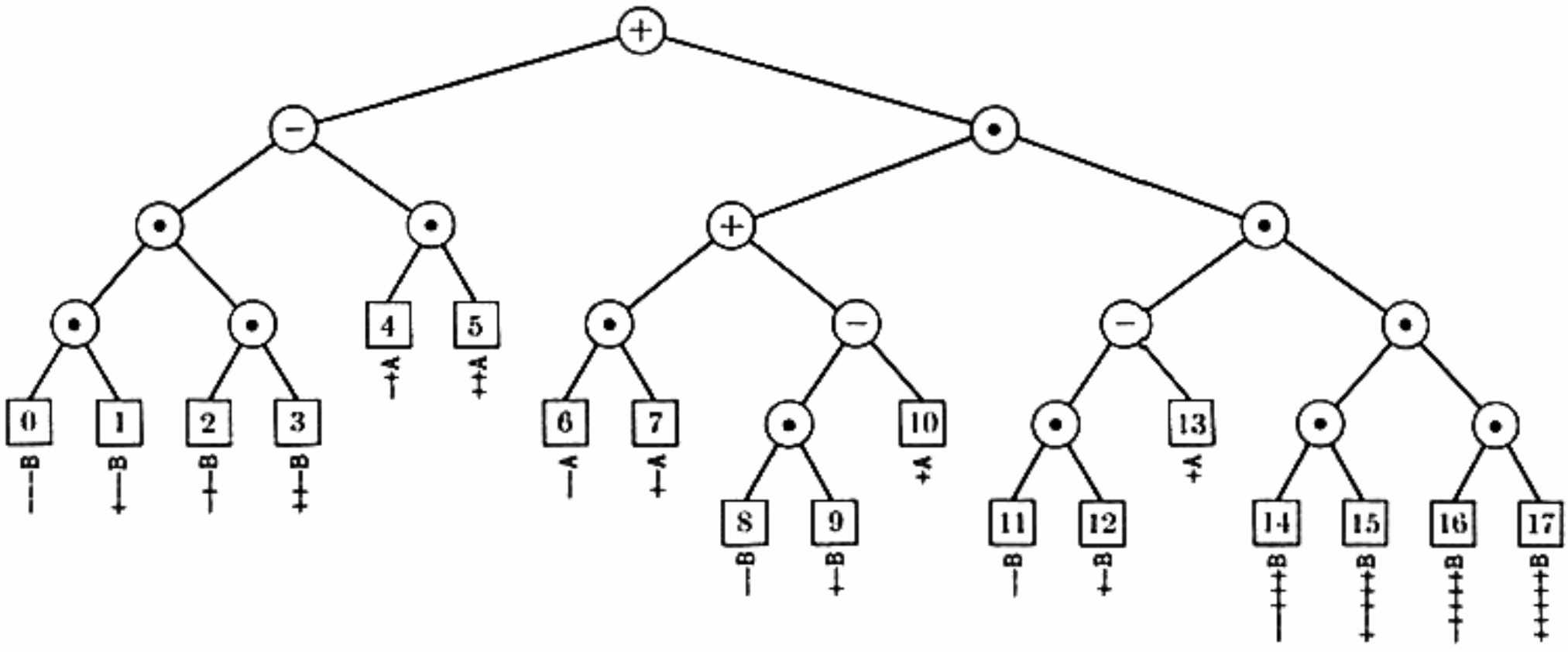


Fig. 23. Classification codes which specify the behavior of Algorithm A after insertion.

Fig. 23. The path leading up from an external node can be specified by a sequence of +'s and −'s (+ for a right link, − for a left link); we write down the link specifications until reaching the first node with a nonzero balance factor, or until reaching the root, if there is no such node. Then we write A or B according as the new tree will be balanced or unbalanced when an internal node is inserted in the given place. Thus the path up from [3] is ++−B, meaning "right link, right link, left link, unbalance." A specification ending in A requires no rebalancing after insertion of a new node; a specification ending in ++B or −−B requires a single rotation; and a specification ending in +−B or −+B requires a double rotation. When k links appear in the specification, step A6 has to adjust exactly $k - 1$ balance factors. Thus the specifications give the essential facts governing the running time of steps A6 to A10.

Empirical tests on random numbers for $100 \leq N \leq 2000$ gave the approximate probabilities shown in Table 1 for paths of various types; apparently these probabilities rapidly approach limiting values as $N \to \infty$. Table 2 gives the exact probabilities corresponding to Table 1 when $N = 10$, considering the 10! permutations of the input as equally probable.

Table 1

APPROXIMATE PROBABILITIES FOR INSERTING THE NTH ITEM

Path length k	No rebalancing	Single rotation	Double rotation
1	.144	.000	.000
2	.153	.144	.144
3	.093	.048	.048
4	.058	.023	.023
5	.036	.010	.010
> 5	.051	.008	.007
ave 2.8	.535	.233	.232

Table 2

EXACT PROBABILITIES FOR INSERTING THE 10TH ITEM

Path length k	No rebalancing	Single rotation	Double rotation
1	1/7	0	0
2	6/35	1/7	1/7
3	4/21	2/35	2/35
4	0	1/21	1/21
ave	247/105	53/105	26/105	26/105

From Table 1 we can see that k is ≤ 2 with probability $.144 + .153 + .144 + .144 = .585$; thus, step A6 is quite simple almost 60 percent of the time. The average number of balance factors changed from 0 to ± 1 in that step is about 1.8. The average number of balance factors changed from ± 1 to 0 in steps A7 through A10 is $.535 + 2(.233 + .232) = 1.5$; thus, inserting one new node adds about .3 unbalanced nodes, on the average. This agrees with the fact that about 68 percent of all nodes were found to be balanced in random trees built by Algorithm A.

An approximate model of the behavior of Algorithm A has been proposed by C. C. Foster [*Proc. ACM Nat. Conf.* **20** (1965), 192–205]. This model is not rigorously accurate, but it is close enough to the truth to give some insight. Let us assume that p is the probability that the balance factor of a given node in a large tree built by Algorithm A is 0; then the balance factor is $+1$ with probability $\frac{1}{2}(1 - p)$, and it is -1 with the same probability $\frac{1}{2}(1 - p)$. Let us assume further (without justification) that the balance factors of all nodes are independent. Then the probability that step A6 sets exactly $k - 1$ balance factors nonzero is $p^{k-1}(1 - p)$, so the average value of $k - 1$ is $p/(1 - p)$. The probability that we need to rotate part of the tree is $\frac{1}{2}$. Inserting a new node should increase the number of balanced nodes by p, on the average; this number is actually increased by 1 in step A5, by $-p/(1 - p)$ in step A6, by $\frac{1}{2}$ in step A7, and by $\frac{1}{2} \cdot 2$ in step A8 or A9, so we should have

$$p = 1 - p/(1 - p) + \tfrac{1}{2} + 1.$$

Solving for p yields fair agreement with Table 1:

$$p = \frac{9 - \sqrt{41}}{4} \approx 0.649; \qquad p/(1 - p) \approx 1.851. \tag{14}$$

The running time of the search phase of Program A (lines 01–19) is

$$10C + C1 + 2D + 2 - 3S, \tag{15}$$

where C, $C1$, S are the same as in previous algorithms of this chapter and D is the number of unbalanced nodes encountered on the search path. Empirical

tests show that we may take $D \approx \frac{1}{3}C, C1 \approx \frac{1}{2}(C+S), C+S \approx \log_2 N + 0.25$, so the average search time is approximately $11.17 \log_2 N + 4.8 - 13.7S$ units. (If searching is done much more often than insertion, we could of course use a separate, faster program for searching, since it would be unnecessary to look at the balance factors; the average running time for a successful search would then be only about $(6.5 \log_2 N + 4.1)u$, and the worst case running time would in fact be better than the average running time obtained with Algorithm 6.2.2T.)

The running time of the insertion phase of Program A (lines 20–45) is $8F + 26 +$ (0, 1, or 2) units, when the search is unsuccessful. The data of Table 1 indicate that $F \approx 1.8$ on the average. The rebalancing phase (lines 46–101) takes either 16.5, 8, 27.5, or 45.5 (± 0.5) units, depending on whether we increase the total height, or simply exit without rebalancing, or do a single or double rotation. The first case almost never occurs, and the others occur with the approximate probabilities .535, .233, .232, so the average running time of the combined insertion-rebalancing portion of Program A is about $63u$.

These figures indicate that maintenance of a balanced tree in memory is reasonably fast, even though the program is rather lengthy. If the input data are random, the simple tree insertion algorithm of Section 6.2.2 is roughly $50u$ faster per insertion; but the balanced tree algorithm is guaranteed to be reliable even with nonrandom input data.

One way to compare Program A with Program 6.2.2T is to consider the worst case of the latter. If we study the amount of time necessary to insert N keys in increasing order into an initially empty tree, it turns out that Program A is slower for $N \leq 26$ and faster for $N \geq 27$.

Linear list representation. Now let us return to the claim made at the beginning of this section, that balanced trees can be used to represent linear lists in such a way that we can insert items rapidly (overcoming the difficulty of sequential allocation), yet we can also perform random accesses to list items (overcoming the difficulty of linked allocation).

The idea is to introduce a new field in each node, called the RANK field. This field indicates the relative position of that node in its subtree, i.e., one plus the

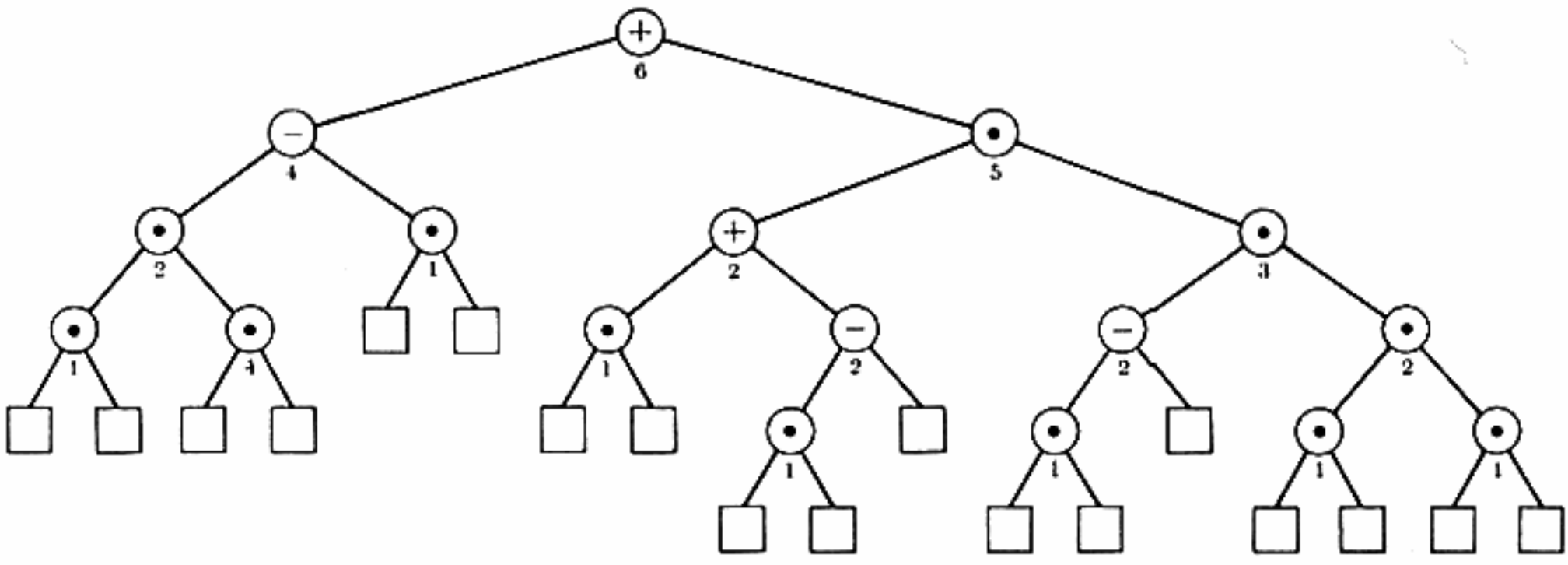


Fig. 24. RANK fields, used for searching by position.

number of nodes in its left subtree. Figure 24 shows the RANK values for the binary tree of Fig. 23. We can eliminate the KEY field entirely; or, if desired, we can have both KEY and RANK fields, so that it is possible to retrieve items either by their key value or by their relative position in the list.

Using such a RANK field, retrieval by position is a straightforward modification of the search algorithms we have been studying.

Algorithm B (*Tree search by position*). Given a linear list represented as a binary tree, this algorithm finds the kth element of the list (the kth node of the tree in symmetric order), given k. The binary tree is assumed to have LLINK and RLINK fields and a header as in Algorithm A, plus a RANK field as described above.

B1. [Initialize.] Set $\texttt{M} \leftarrow k$, $\texttt{P} \leftarrow \texttt{RLINK(HEAD)}$.

B2. [Compare.] If $\texttt{P} = \Lambda$, the algorithm terminates unsuccessfully. (This can happen only if k was greater than the number of nodes in the tree, or $k \leq 0$.) Otherwise if $\texttt{M} < \texttt{RANK(P)}$, go to B3; if $\texttt{M} > \texttt{RANK(P)}$, go to B4; and if $\texttt{M} = \texttt{RANK(P)}$, the algorithm terminates successfully (P points to the kth node).

B3. [Move left.] Set $\texttt{P} \leftarrow \texttt{LLINK(P)}$ and return to B2.

B4. [Move right.] Set $\texttt{M} \leftarrow \texttt{M} - \texttt{RANK(P)}$ and $\texttt{P} \leftarrow \texttt{RLINK(P)}$ and return to B2. ∎

The only new point of interest in this algorithm is the manipulation of M in step B4. We can modify the insertion procedure in a similar way, although the details are somewhat trickier:

Algorithm C (*Balanced tree insertion by position*). Given a linear list represented as a balanced binary tree, this algorithm inserts a new node just before the kth element of the list, given k and a pointer Q to the new node. If $k = N + 1$, the new node is inserted just after the last element of the list.

The binary tree is assumed to be nonempty and to have LLINK, RLINK, and B fields and a header, as in Algorithm A, plus a RANK field as described above. This algorithm is merely a transcription of Algorithm A; the difference is that it uses and updates the RANK fields instead of the KEY fields.

C1. [Initialize.] Set $\texttt{T} \leftarrow \texttt{HEAD}$, $\texttt{S} \leftarrow \texttt{P} \leftarrow \texttt{RLINK(HEAD)}$, $\texttt{U} \leftarrow \texttt{M} \leftarrow k$.

C2. [Compare.] If $\texttt{M} \leq \texttt{R(P)}$, go to C3, otherwise go to C4.

C3. [Move left.] Set $\texttt{RANK(P)} \leftarrow \texttt{RANK(P)} + 1$ (we will be inserting a new node to the left of P). Set $\texttt{R} \leftarrow \texttt{LLINK(P)}$. If $\texttt{R} = \Lambda$, set $\texttt{LLINK(P)} \leftarrow \texttt{Q}$ and go to C5. Otherwise if $\texttt{B(R)} \neq 0$ set $\texttt{T} \leftarrow \texttt{P}$, $\texttt{S} \leftarrow \texttt{R}$, and $\texttt{U} \leftarrow \texttt{M}$. Finally set $\texttt{P} \leftarrow \texttt{R}$ and return to C2.

C4. [Move right.] Set $\texttt{M} \leftarrow \texttt{M} - \texttt{RANK(P)}$, and $\texttt{R} \leftarrow \texttt{RLINK(P)}$. If $\texttt{R} = \Lambda$, set $\texttt{RLINK(P)} \leftarrow \texttt{Q}$ and go to C5. Otherwise if $\texttt{B(R)} \neq 0$ set $\texttt{T} \leftarrow \texttt{P}$, $\texttt{S} \leftarrow \texttt{R}$, and $\texttt{U} \leftarrow \texttt{M}$. Finally set $\texttt{P} \leftarrow \texttt{R}$ and return to C2.

C5. [Insert.] Set $\texttt{RANK(Q)} \leftarrow 1$, $\texttt{LLINK(Q)} \leftarrow \texttt{RLINK(Q)} \leftarrow \Lambda$, $\texttt{B(Q)} \leftarrow 0$.

C6. [Adjust balance factors.] Set M ← U. (This restores the former value of M when P was S; all RANK fields are now properly set.) If M < RANK(S), set R ← P ← LLINK(S), otherwise set R ← P ← RLINK(S) and M ← M − RANK(S). Then repeatedly do the following operation until P = Q: If M < RANK(P), set B(P) ← −1 and P ← LLINK(P); if M > RANK(P), set B(P) ← +1 and M ← M − RANK(P) and P ← RLINK(P). (If M = RANK(P), then P = Q and we may go on to the next step.)

C7. [Balancing act.] If U < RANK(S), set $a \leftarrow -1$, otherwise set $a \leftarrow +1$. Several cases now arise:

i) If B(S) = 0, set B(S) ← a, LLINK(HEAD) ← LLINK(HEAD) + 1, and terminate the algorithm.

ii) If B(S) = $-a$, set B(S) ← 0 and terminate the algorithm.

iii) If B(S) = a, go to step C8 if B(R) = a, to C9 if B(R) = $-a$.

C8. [Single rotation.] Set P ← R, LINK(a, S) ← LINK($-a$, R), LINK($-a$, R) ← S, B(S) ← B(R) ← 0. If $a = +1$, set RANK(R) ← RANK(R) + RANK(S); if $a = -1$, set RANK(S) ← RANK(S) − RANK(R).

C9. [Double rotation.] Do all the operations of step A9 (Algorithm A). Then if $a = +1$, set RANK(R) ← RANK(R) − RANK(P), RANK(P) ← RANK(P) + RANK(S); if $a = -1$, set RANK(P) ← RANK(P) + RANK(R), then RANK(S) ← RANK(S) − RANK(P).

C10. [Finishing touch.] If S = RLINK(T) then set RLINK(T) ← P, otherwise set LLINK(T) ← P. ∎

***Deletion, concatenation, etc.** It is possible to do many other things to balanced trees and maintain the balance, but the algorithms are sufficiently lengthy that the details are beyond the scope of this book. We shall discuss the general ideas here, and an interested reader will be able to fill in the details without much difficulty.

The problem of deletion can be solved in $O(\log N)$ steps if we approach it correctly [C. C. Foster, "A Study of AVL Trees," Goodyear Aerospace Corp. report *GER*-12158 (April, 1965)]. In the first place we can reduce deletion of an arbitrary node to the simple deletion of a node P for which LLINK(P) or RLINK(P) is Λ, as in Algorithm 6.2.2D. The algorithm should also be modified so that it constructs a list of pointers which specifies the path to node P, namely

$$(P_0, a_0), \quad (P_1, a_1), \quad \ldots, \quad (P_l, a_l), \tag{16}$$

where P_0 = HEAD, $a_0 = +1$; LINK(a_i, P_i) = P_{i+1}, for $0 \le i < l$; P_l = P; and LINK(a_l, P_l) = Λ. This list can be placed on an auxiliary stack as we search down the tree. The process of deleting node P sets LINK(a_{l-1}, P_{l-1}) ← LINK($-a_l$, P_l), and we must adjust the balance factor at node P_{l-1}. Suppose that we need to adjust the balance factor at node P_k, because the a_k subtree of this node has just decreased in height; the following adjustment procedure should be used: If $k = 0$, set LLINK(HEAD) ← LLINK(HEAD) − 1 and terminate the algorithm, since the whole tree has decreased in height. Otherwise look at

the balance factor $B(P_k)$; there are three cases:

i) $\texttt{B}(P_k) = a_k$. Set $\texttt{B}(P_k) \leftarrow 0$, decrease k by 1, and repeat the adjustment procedure for this new value of k.

ii) $\texttt{B}(P_k) = 0$. Set $\texttt{B}(P_k)$ to $-a_k$ and terminate the deletion algorithm.

iii) $\texttt{B}(P_k) = -a_k$. Rebalancing is required!

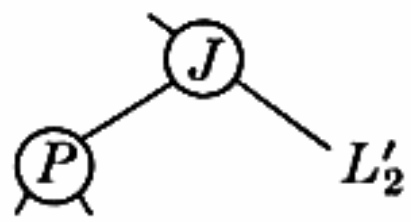
The situations requiring rebalancing are almost the same as we met in the insertion algorithm; referring again to (1), A is node P_k, and B is node $\texttt{LINK}(-a_k, P_k)$, the *opposite* branch from where the deletion has occurred. The only new feature is that node B might be balanced; this leads to a new Case 3 which is like Case 1 except that β has height $h + 1$. In Cases 1 and 2, rebalancing as in (2) means that we decrease the height, so we set $\texttt{LINK}(a_{k-1}, P_{k-1})$ to the root of (2), decrease k by 1, and restart the adjustment procedure for this new value of k. In Case 3 we do a single rotation, and this leaves the balance factors of both A and B nonzero without changing the overall height; after making $\texttt{LINK}(a_{k-1}, P_{k-1})$ point to node B, we therefore terminate the algorithm.

The important difference between deletion and insertion is that deletion might require up to $\log N$ rotations, while insertion never needs more than one. The reason for this becomes clear if we try to delete the rightmost node of a Fibonacci tree (see Fig. 8 in Section 6.2.1).

The use of balanced trees for linear list representation suggests also the need for a *concatenation* algorithm, where we want to insert an entire tree L_2 to the right of tree L_1, without destroying the balance. An elegant algorithm for concatenation has been devised by Clark A. Crane: Assume that $\text{height}(L_1) \geq \text{height}(L_2)$; the other case is similar. Delete the first node of L_2, calling it the "juncture node" J, and let L_2' be the new tree for $L_2 \backslash \{J\}$. Now go down the right links of L_1 until reaching a node P such that

$$\text{height}(P) - \text{height}(L_2') = 0 \text{ or } 1;$$

this is always possible, since the height changes by 1 or 2 each time we go down one level. Then replace (P) by



and proceed to adjust L_1 as if the new node J had just been inserted by Algorithm A.

Crane has also solved the more difficult inverse problem, to *split* a list into two parts whose concatenation would be the original list. Consider, for example, the problem of splitting the list in Fig. 20 to obtain two lists, one containing $\{\texttt{A}, \ldots, \texttt{I}\}$ and the other containing $\{\texttt{J}, \ldots, \texttt{Q}\}$; a major reassembly of the subtrees is required. In general, when we want to split a tree at some given node P, the path to P will be something like that in Fig. 25. We wish to con-

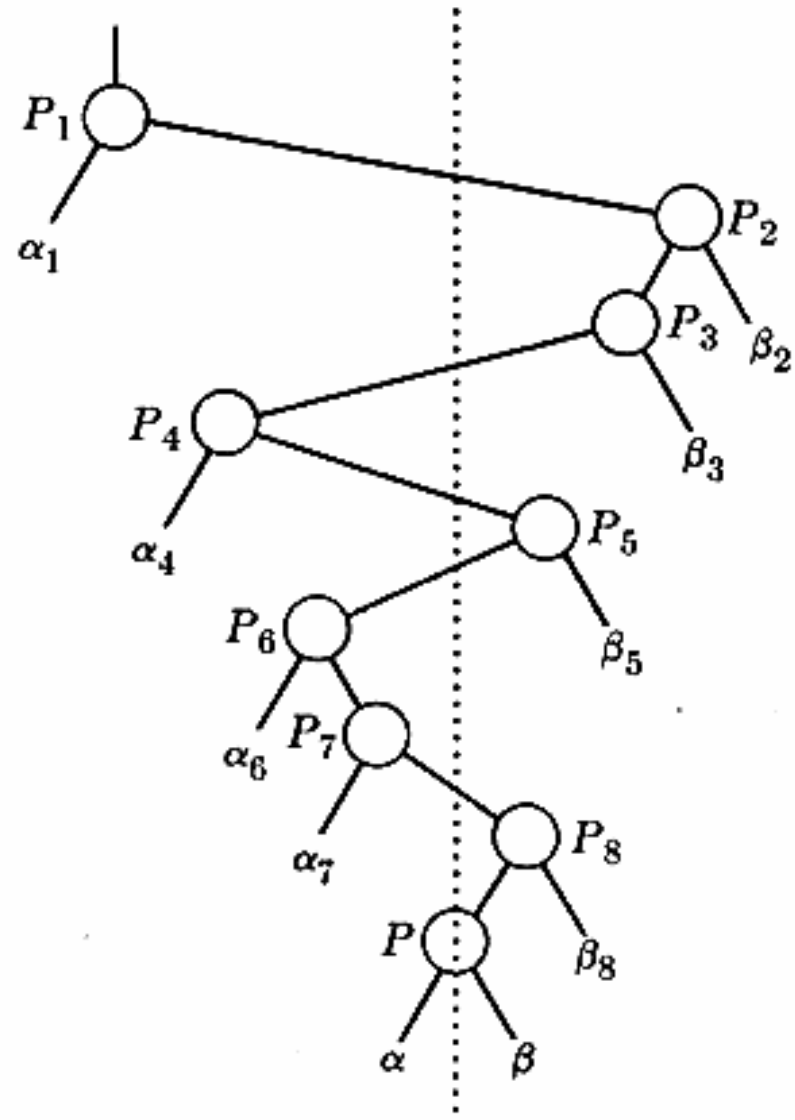


Fig. 25. The problem of splitting a list.

struct a left tree which contains the nodes of $\alpha_1, P_1, \alpha_4, P_4, \alpha_6, P_6, \alpha_7, P_7, \alpha, P$ in symmetric order, and a right tree containing $\beta, P_8, \beta_8, P_5, \beta_5, P_3, \beta_3, P_2, \beta_2$. This can be done by a sequence of concatenations: First insert P at the right of α, then concatenate β with β_8 using P_8 as juncture node, concatenate α_7 with αP using P_7 as juncture node, α_6 with $\alpha_7 P_7 \alpha P$ using P_6, $\beta P_8 \beta_8$ with β_5 using P_5, etc.; the nodes $P_8, P_7, \ldots, P_1$ on the path to P are used as juncture nodes. Crane has proved that this splitting algorithm takes only $O(\log N)$ units of time, when the original tree contains N nodes; the essential reason is that concatenation using a given juncture node takes $O(k)$ steps, where k is the difference in heights between the trees being concatenated, and the values of k that must be summed essentially form a telescoping series for both the left and right trees being constructed.

All of these algorithms can be used with either KEY or RANK fields or both (although in the case of concatenation the keys of L_2 must all be greater than the keys of L_1). For general purposes it is often preferable to use a *triply-linked tree*, with UP links as well as LLINKs and RLINKs, together with a new one-bit field which specifies whether a node is the left or right son of its father. The triply-linked tree representation simplifies the algorithms slightly, and makes it possible to specify nodes in the tree without explicitly tracing the path to that node; we can write a subroutine to delete NODE(P), given P, or to delete the NODE(P$) which follows P in symmetric order, or to find the list containing NODE(P), etc. In the deletion algorithm for triply-linked trees it is unnecessary to construct the list (16), since the UP links provide the information we need. Of course a triply-linked tree requires that a few more links be changed when insertions, deletions, and rotations are being performed. The use of a triply-

linked tree instead of a doubly-linked tree is analogous to the use of two-way linking instead of one-way: we can start at any point and go either forward or backward. A complete description of list algorithms based on triply-linked balanced trees appears in Clark A. Crane's Ph.D. thesis (Stanford University, 1972).

Alternatives to balanced trees. Some other ways of organizing trees, so as to guarantee logarithmic accessing time, have recently been proposed. At the present time they have not yet been studied very thoroughly; it is possible that they may prove to be superior to balanced trees on some computers.

The interesting concept of *weight-balanced tree* has been studied by J. Nievergelt, E. Reingold, and C. K. Wong. Instead of considering the height of trees, we stipulate that the subtrees of all nodes must satisfy

$$\sqrt{2} - 1 < \frac{\text{left weight}}{\text{right weight}} < \sqrt{2} + 1, \tag{17}$$

where the left and right weights count the number of *external* nodes in the left and right subtrees, respectively. It is possible to show that weight-balance can be maintained under insertion, using only single and double rotations for rebalancing as in Algorithm A (see exercise 25). However, it may be necessary to do many rebalancings during a single insertion. It is possible to relax the conditions of (17), decreasing the amount of rebalancing at the expense of increased search time.

Weight-balanced trees may seem at first glance to require more memory than plain balanced trees, but in fact they sometimes require slightly less! If we already have a RANK field in each node, for the linear list representation, this is precisely the left weight, and it is possible to keep track of the corresponding right weights as we move down the tree. However, it appears that the bookkeeping required for maintaining weight balance takes more time than Algorithm A, and this small savings of two bits per node is probably not worth the trouble.

Another interesting alternative to balanced trees, called "3-2 trees," was introduced by John Hopcroft in 1970 (unpublished). The idea is to have either 2-way or 3-way branching at each node, and to stipulate that all external nodes appear on the same level. Every internal node contains either one or two keys, as shown in Fig. 26.

Insertion into a 3-2 tree is somewhat easier to explain than insertion into a balanced tree: If we want to put a new key into a node that contains just one

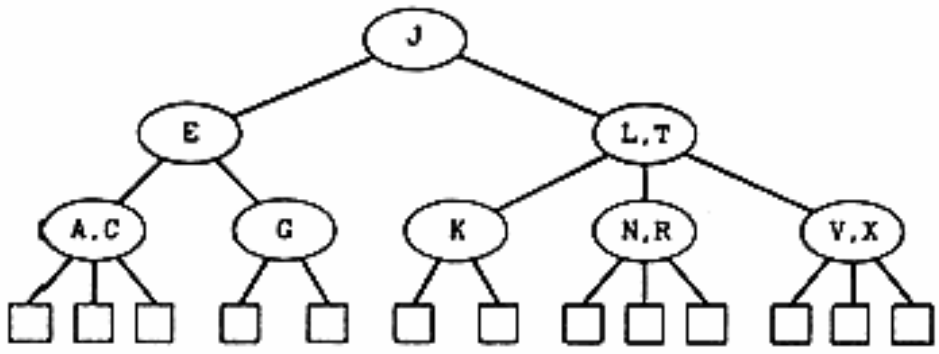


Fig. 26. A 3-2 tree.

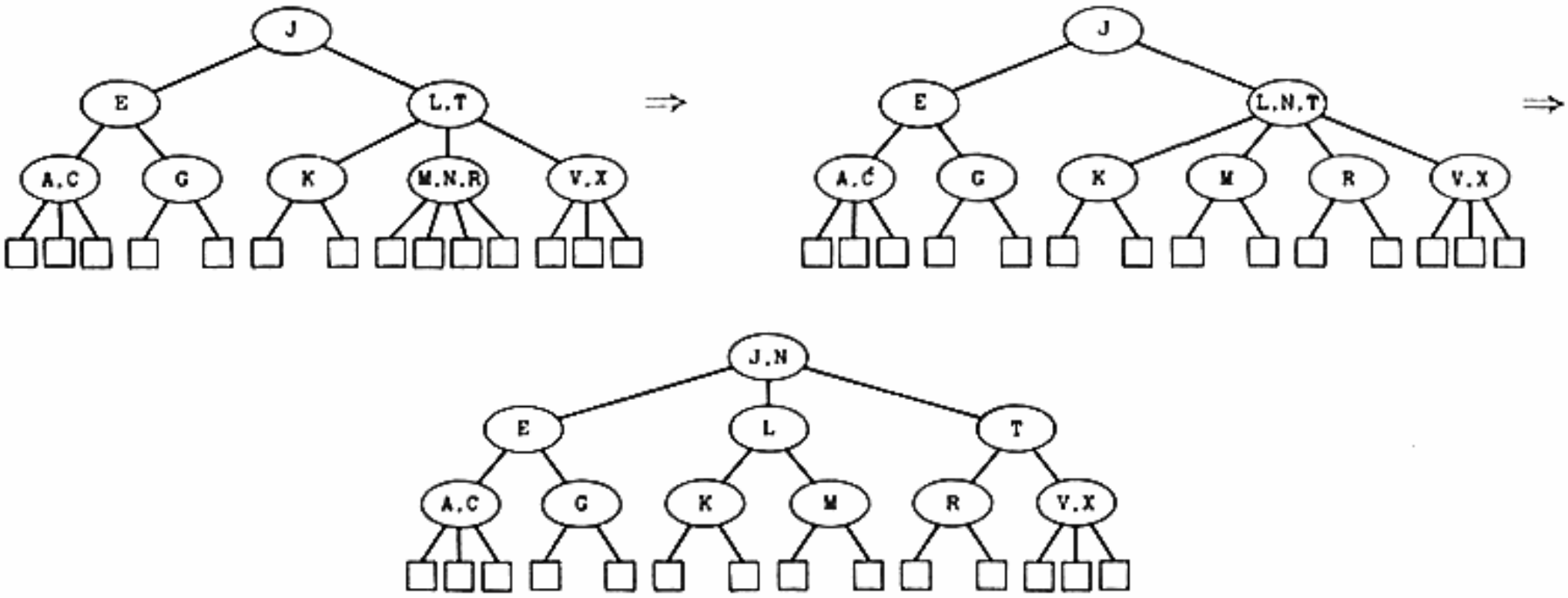


Fig. 27. Inserting the new key "M" into the 3-2 tree of Fig. 26.

key, we simply insert it as the second key. On the other hand, if the node already contains two keys, we divide it into two one-key nodes, and insert the middle key into the parent node. This may cause the parent node to be divided in a similar way, if it already contains two keys. Figure 27 shows the process of inserting a new key into the 3-2 tree of Fig. 26.

Hopcroft has observed that deletion, concatenation, and splitting can all be done with 3-2 trees, in a reasonably straightforward manner analogous to the corresponding operations with balanced trees.

R. Bayer [*Proc. ACM-SIGFIDET Workshop* (1971), 219–235] has suggested an interesting binary tree representation for 3-2 trees. See Fig. 28, which shows the binary tree representation of Fig. 26; one bit in each node is used to distinguish "horizontal" RLINKs from "vertical" ones. Note that the keys of the tree appear from left to right in symmetric order, just as in any binary search tree. It turns out that the transformations we need to perform on such a binary tree, while inserting a new key as in Fig. 27, are precisely the single and double rotations used while inserting a new key into a balanced tree, although we need just one version of each rotation (not the left-right reflections as in Algorithms A and C).

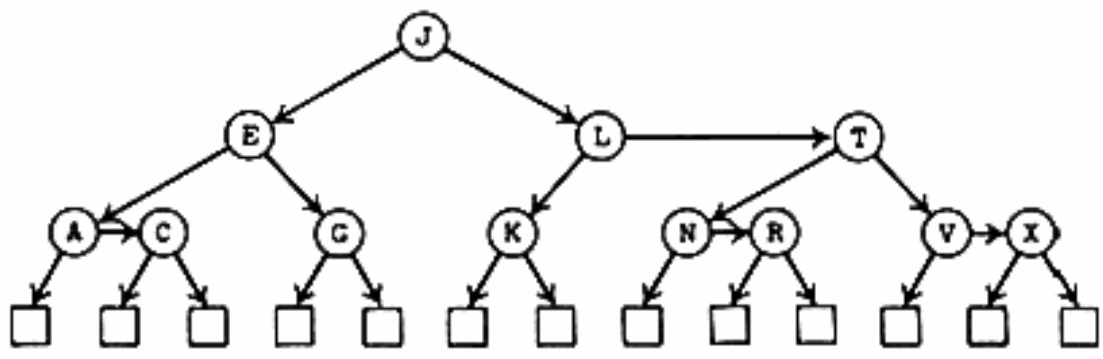


Fig. 28. The 3-2 tree of Fig. 26 represented as a binary search tree.

EXERCISES

1. [*01*] In Case 2 of (1), why isn't it a good idea to restore the balance by simply interchanging the left subtrees of A and B?

2. [*16*] Explain why the tree has gotten one level higher if we reach step A7 with B(S) = 0.

▸ **3.** [*M25*] Prove that a balanced tree with N internal nodes never contains more than $(\phi - 1)N = 0.61803\,N$ nodes whose balance factor is nonzero.

4. [*M22*] Prove or disprove: Among all balanced trees with $F_{n+1} - 1$ internal nodes, the Fibonacci tree of order n has the greatest internal path length.

▸ **5.** [*M25*] Prove or disprove: If Algorithm A is used to insert N keys into an initially empty tree, and if these keys arrive in increasing order, the tree produced is always *optimum* (i.e., it has minimum internal path length over all N-node binary trees).

6. [*M21*] Prove that Eq. (5) defines the generating function for balanced trees of height h.

7. [*M27*] (N. J. A. Sloane and A. V. Aho.) Prove the remarkable formula (9) for the number of balanced trees of height h. [*Hint:* Let $C_n = B_n + B_{n-1}$, and use the fact that $\log (C_{n+1}/C_n^2)$ is exceedingly small for large n.]

8. [*M24*] (L. A. Khizder.) Show that there is a constant β such that $B_h'(1)/B_h(1) - 2^h\beta \to 1$ as $h \to \infty$.

9. [*M47*] What is the asymptotic number of balanced binary trees with n internal nodes, $\sum_{h\geq 0} B_{nh}$? What is the asymptotic average height, $\sum_{h\geq 0} hB_{nh}/\sum_{h\geq 0} B_{nh}$?

10. [*M48*] Does Algorithm A make an average of $\sim\log_2 N + c$ comparisons to insert the Nth item, for some constant c?

▸ **11.** [*22*] The value .144 appears three times in Table 1, once for $k = 1$ and twice for $k = 2$. The value $\frac{1}{7}$ appears in the same three places in Table 2. Is it a coincidence that the same value should appear in all three places, or is there some good reason for this?

▸ **12.** [*24*] What is the maximum possible running time of Program A when the eighth node is inserted into a balanced tree? What is the minimum possible running time for this insertion?

13. [*10*] Why is it better to use RANK fields as defined in the text, instead of simply to store the index of each node as its key (calling the first node "1", the second node "2", and so on)?

14. [*11*] Could Algorithms 6.2.2T and 6.2.2D be adapted to work with linear lists, using a RANK field, just as the balanced tree algorithms of this section have been so adapted?

15. [*18*] (C. A. Crane.) Suppose that an ordered linear list is being represented as a binary tree, with both KEY and RANK fields in each node. Design an algorithm which searches the tree for a given key, K, and determines the position of K in the list; i.e., it finds the number M such that K is the Mth smallest key.

▸ **16.** [*20*] Draw the balanced tree that would be obtained if the root node F were deleted from Fig. 20, using the deletion algorithm suggested in the text.

▸ **17.** [*21*] Draw the balanced trees that would be obtained if the Fibonacci tree (12) were concatenated (a) to the right, (b) to the left, of the tree in Fig. 20, using the concatenation algorithm suggested in the text.

18. [*21*] Draw the balanced trees that would be obtained if Fig. 20 were split into two parts {A, . . . , I} and {J, . . . , Q}, using the splitting algorithm suggested in the text.

▸**19.** [*26*] Find a way to transform a given balanced tree so that the balance factor at the root is not -1. Your transformation should preserve the symmetric order of the nodes; and it should produce another balanced tree in $O(1)$ units of time, regardless of the size of the original tree.

20. [*40*] Explore the idea of using the restricted class of balanced trees whose nodes all have balance factors of 0 or $+1$. (Then the length of the B field can be reduced to one bit.) Is there a reasonably efficient insertion procedure for such trees?

▸**21.** [*30*] Design an algorithm which constructs optimum N-node binary trees (in the sense of exercise 5), in $O(N)$ steps. Your algorithm should be "on line," in the sense that it inputs the nodes one by one in increasing order and builds partial trees as it goes, without knowing the final value of N in advance. (It would be appropriate to use such an algorithm when restructuring a badly balanced tree, or when merging the keys of two trees into a single tree.)

22. [*M20*] What is the analog of Theorem A, for weight-balanced trees?

23. [*M20*] (E. Reingold.) (a) Prove that there exist balanced trees whose weight balance (left weight)/(right weight) is arbitrarily small. (b) Prove that there exist weight-balanced trees having arbitrarily large differences between left and right subtree heights.

24. [*M22*] (E. Reingold.) Prove that if we strengthen condition (17) to

$$\frac{1}{2} < \frac{\text{left weight}}{\text{right weight}} < 2,$$

the only binary trees which satisfy this condition are perfectly balanced trees with $2^n - 1$ internal nodes. (In such trees, the left and right weights are exactly equal at all nodes.)

25. [*27*] (J. Nievergelt, E. Reingold, C. Wong.) Show that it is possible to design an insertion algorithm for weight-balanced trees so that condition (17) is preserved, making at most $O(\log N)$ rotations per insertion.

26. [*40*] Explore the properties of balanced t-ary trees, for $t > 2$.

▸**27.** [*M23*] Estimate the maximum number of comparisons needed to search in a 3-2 tree with N internal nodes.

28. [*41*] Prepare efficient implementations of 3-2 tree algorithms.

29. [*M47*] Analyze the average behavior of 3-2 trees under random insertions.

30. [*26*] (E. McCreight.) Section 2.5 discusses several strategies for dynamic storage allocation, including "best-fit" (choosing an available area as small as possible from among all those which fulfill the request) and "first-fit" (choosing the available area with lowest address among all those that fulfill the request). Show that if the available space is linked together as a balanced tree in an appropriate way, it is possible to do (a) best-fit (b) first-fit allocation in only $O(\log n)$ units of time, where n is the number of available areas. (The algorithms given in Section 2.5 take order n steps.)

6.2.4. Multiway Trees

The tree search methods we have been discussing were developed primarily for internal searching, when we want to look at a table that is contained entirely within a computer's high-speed internal memory. Let's now consider the prob-

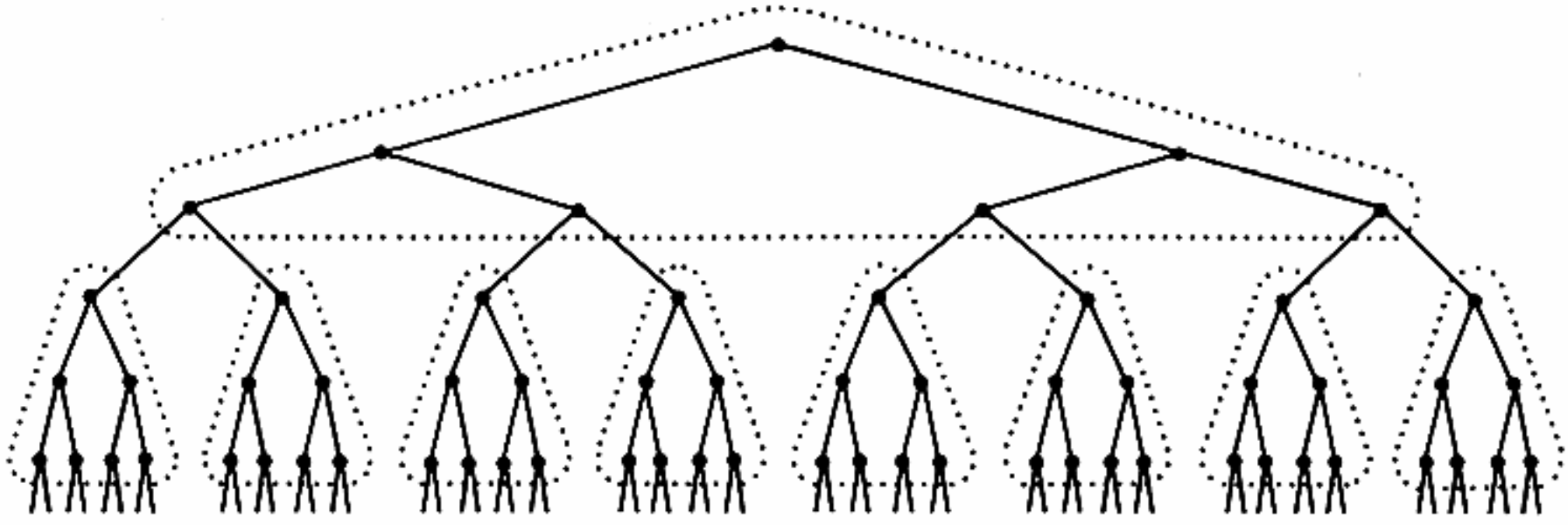


Fig. 29. A large binary search tree can be divided into "pages."

lem of *external* searching, when we want to retrieve information from a very large file that appears on direct access storage units such as disks or drums. (An introduction to disks and drums appears in Section 5.4.9.)

Tree structures lend themselves nicely to external searching, if we choose an appropriate way to represent the tree. Consider the large binary search tree shown in Fig. 29, and imagine that it has been stored in a disk file. (The LLINKs and RLINKs of the tree are now disk addresses instead of internal memory addresses.) If we search this tree in a naïve manner, simply applying the algorithms we have learned for internal tree searching, we will have to make about $\log_2 N$ disk accesses before our search is complete. When N is a million, this means we will need 20 or so seeks. But suppose we divide the table into 7-node "pages," as shown by the dotted lines in Fig. 29; if we access one page at a time, we need only about one third as many seeks, so the search goes about three times as fast!

Grouping the nodes into pages in this way essentially changes the tree from a binary tree to an octonary tree, with 8-way branching at each page-node. If we let the pages be still larger, with 128-way branching after each disk access, we can find any desired key in a million-entry table after looking at only three pages. We can keep the root page in the internal memory at all times, so that only two references to the disk are required even though we never have more than 254 keys in the internal memory at any time.

Of course we don't want to make the pages arbitrarily large, since the internal memory size is limited and also since it takes a longer time to read in a larger page. For example, suppose that it takes $72.5 + 0.05m$ milliseconds to read a page that allows m-way branching. The internal processing time per page will be about $a + b \log m$, where a is small compared to 72.5 ms, so the total amount of time needed for searching a large table is approximately proportional to $\log N$ times

$$(72.5 + 0.05m)/\log m + b.$$

This quantity achieves a minimum when $m \approx 350$; actually the minimum is very "broad," a nearly optimum value is achieved for all m between 200 and 500. In practice there will be a similar range of good values for m, based on

the characteristics of particular external memory devices and on the length of the records in the table.

W. I. Landauer [*IEEE Trans.* **EC-12** (1963), 863–871] suggested building an m-ary tree by requiring level l to become nearly full before anything is allowed to appear on level $l + 1$. This scheme requires a rather complicated rotation method, since we may have to make major changes throughout the tree just to insert a single new item; Landauer was assuming that we need to search for items in the tree much more often than we need to insert or delete them.

When a file is stored on disk, and is subject to comparatively few insertions and deletions, a three-level tree is appropriate, where the first level of branching determines what cylinder is to be used, the second level of branching determines the appropriate track on that cylinder, and the third level contains the records themselves. This method is called *indexed-sequential* file organization [cf. *JACM* **16** (1969), 569–571].

R. Muntz and R. Uzgalis [*Proc. Princeton Conf. on Inf. Sciences and Systems* **4** (1970), 345–349] have suggested modifying the tree search and insertion method, Algorithm 6.2.2T, so that all insertions go onto nodes belonging to the same page as their father node, whenever possible; if that page is full, a new page is started, whenever possible. If the number of pages is unlimited, and if the data arrives in random order, it can be shown that the average number of page accesses is approximately $H_N/(H_m - 1)$, only slightly more than we would obtain in the best possible m-ary tree. (See exercise 10.)

B-trees. A new approach to external searching by means of multiway tree branching was discovered in 1970 by R. Bayer and E. McCreight [*Acta Informatica* (1972), 173–189], and independently at about the same time by M. Kaufman [unpublished]. Their idea, based on a versatile new kind of data structure called a *B-tree*, makes it possible both to search and to update a large file with "guaranteed" efficiency, in the worst case, using comparatively simple algorithms.

A *B-tree of order m* is a tree which satisfies the following properties:

i) Every node has $\leq m$ sons.

ii) Every node, except for the root and the leaves, has $\geq m/2$ sons.

iii) The root has at least 2 sons (unless it is a leaf).

iv) All leaves appear on the same level, and carry no information.

v) A nonleaf node with k sons contains $k - 1$ keys.

(As usual, a "leaf" is a terminal node, one with no sons. Since the leaves carry no information, we may regard them as external nodes which aren't really in the tree, so that Λ is a pointer to a leaf.)

Figure 30 shows a B-tree of order 7. Each node (except for the root and the leaves) has between $\lceil 7/2 \rceil$ and 7 sons, so it contains 3, 4, 5, or 6 keys. The root node is allowed to contain from 1 to 6 keys; in this case it has 2. All of the leaves are at level 3. Note that (a) the keys appear in increasing order from

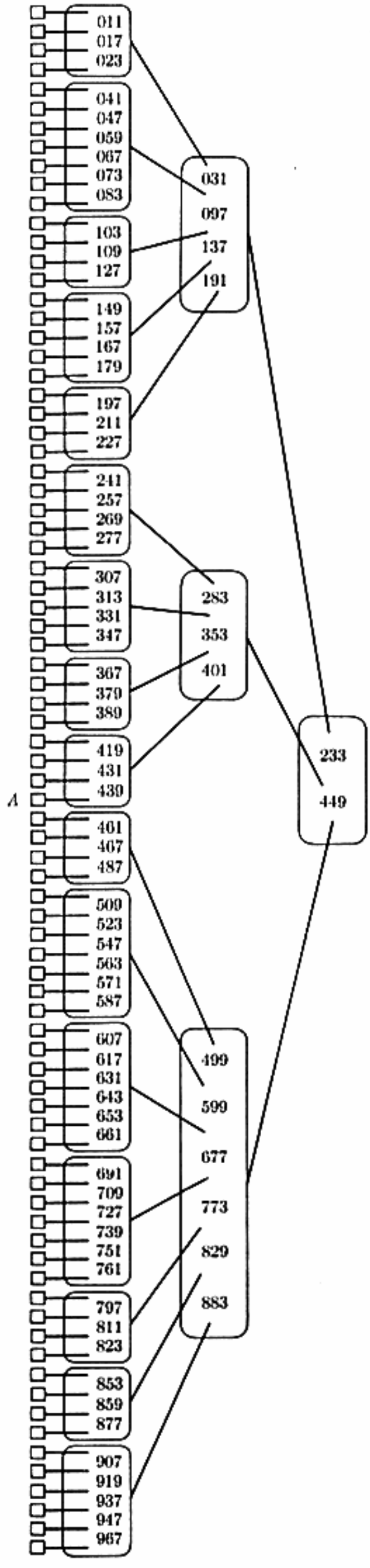
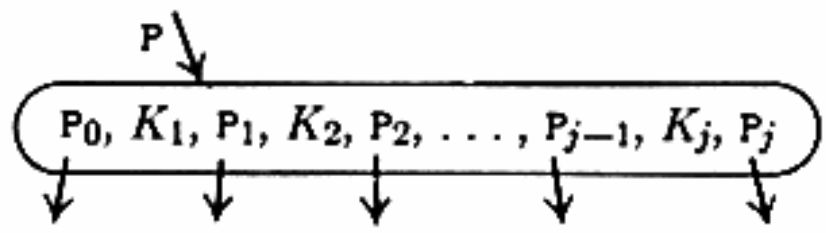


Fig. 30. A B-tree of order 7, with all leaves on level 3.

left to right, using a natural extension of the concept of symmetric order; and (b) the number of leaves is exactly one greater than the number of keys.

B-trees of order 1 or 2 are obviously uninteresting, so we will consider only the case $m \geq 3$. Note that "3-2 trees," defined at the close of Section 6.2.3, are *B*-trees of order 3; and conversely, a *B*-tree of order 3 is a 3-2 tree.

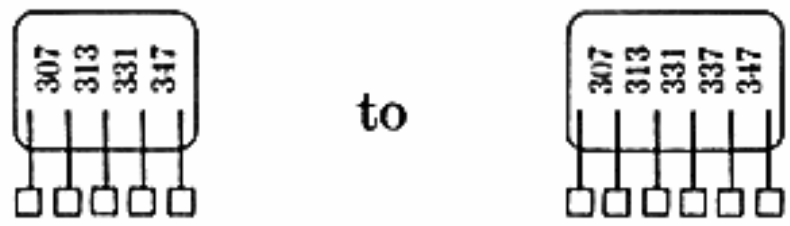
A node which contains j keys and $j + 1$ pointers can be represented as



(1)

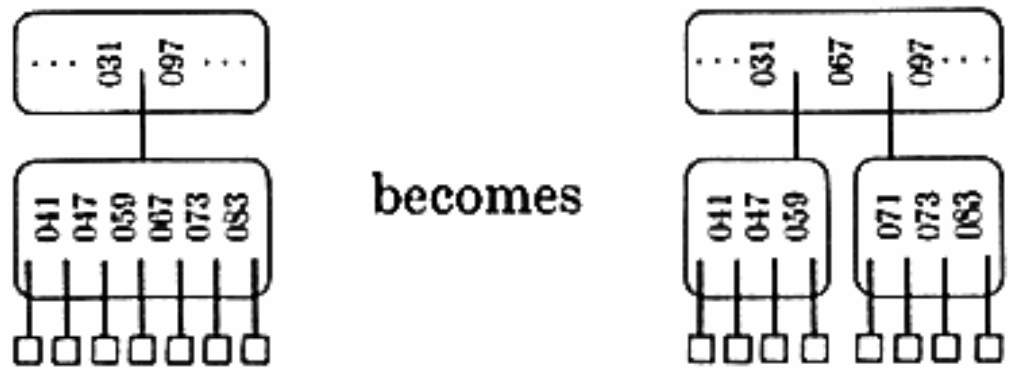
where $K_1 < K_2 < \cdots < K_j$ and $\mathtt{P}_i$ points to the subtree for keys between K_i and K_{i+1}. Therefore searching in a *B*-tree is quite straightforward: After node (1) has been fetched into the internal memory, we search for the given argument among the keys $K_1, K_2, \ldots, K_j$. (When j is large, we probably do a binary search; but when j is smallish, a sequential search is best.) If the search is successful, we have found the desired key; but if the search is unsuccessful because the argument lies between K_i and K_{i+1}, we fetch the node indicated by $\mathtt{P}_i$ and continue the process. The pointer $\mathtt{P}_0$ is used if the argument is less than K_1, and $\mathtt{P}_j$ is used if the argument is greater than K_j. If $\mathtt{P}_i = \Lambda$, the search is unsuccessful.

The nice thing about *B*-trees is that insertion is also quite simple. Consider Fig. 30, for example; every leaf corresponds to a place where a new insertion might happen. If we want to insert the new key 337, we simply change the appropriate node from



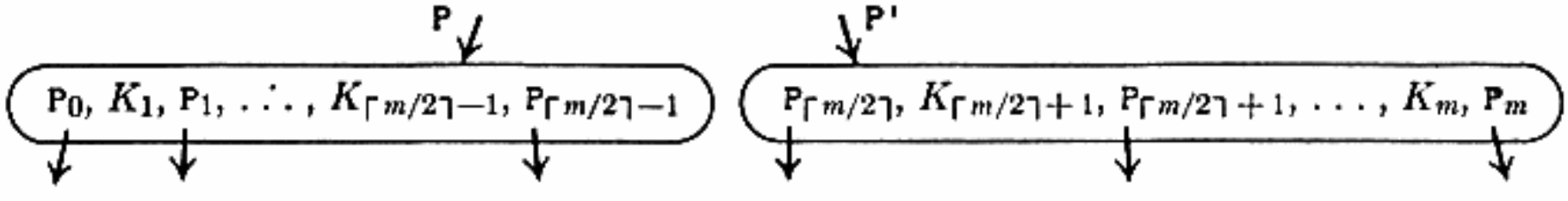
(2)

On the other hand, if we want to insert the new key 071, there is no room since the corresponding node on level 2 is already "full." This case can be handled by splitting the node into two parts, with three keys in each part, and passing the middle key up to level 1:



(3)

In general, if we want to insert a new item into a *B*-tree of order m, when all the leaves are at level l, we insert the new key into the appropriate node on level $l - 1$. If that node now contains m keys, so that it has the form (1) with $j = m$, we split it into two nodes



(4)

and insert the key $K_{\lceil m/2 \rceil}$ into the father of the original node. (Thus the pointer P in the father node is replaced by the sequence P, $K_{\lceil m/2 \rceil}$, P′.) This insertion may cause the father node to contain m keys, and if so, it should be split in the same way. (Cf. Fig. 27, which shows the case $m = 3$.) If we need to split the root node, which has no father, we simply create a new root node containing the single key $K_{\lceil m/2 \rceil}$; the tree gets one level taller in this case.

This insertion procedure neatly preserves all of the B-tree properties; in order to appreciate the full beauty of the idea, the reader should work exercise 1. Note that the tree more or less grows up from the top, instead of down from the bottom, since it gains in height only when the root splits.

Deletion from B-trees is only slightly more complicated than insertion (see exercise 7).

Upper bounds on the performance. Let us now see how many nodes have to be accessed in the worst case, while searching in a B-tree of order m. Suppose that there are N keys, and that the $N + 1$ leaves appear on level l. Then the number of nodes on levels 1, 2, 3, . . . is at least 2, $2\lceil m/2 \rceil$, $2\lceil m/2 \rceil^2$, . . .; hence

$$N + 1 \geq 2 \lceil m/2 \rceil^{l-1}. \tag{5}$$

In other words,

$$l \leq 1 + \log_{\lceil m/2 \rceil} \left(\frac{N+1}{2} \right); \tag{6}$$

this means, for example, that if $N = 1{,}999{,}998$ and $m = 199$, then l is at most 3. Since we need to access at most l nodes during a search, this formula guarantees that the running time is quite small.

When a new node is being inserted, we may have to split as many as l nodes. However, the average number of nodes that need to be split is much less, since the total number of splittings that occur while the entire tree is being constructed is just one less than the total number of nodes in the tree. If there are p nodes, there are at least $1 + (\lceil m/2 \rceil - 1)(p - 1)$ keys; hence

$$p \leq 1 + \frac{N-1}{\lceil m/2 \rceil - 1}. \tag{7}$$

It follows that the average number of times we need to split a node is less than $1/(\lceil m/2 \rceil - 1)$ split per insertion!

Refinements and variations. There are several ways to improve upon the basic B-tree structure defined above, by breaking the rules a little.

In the first place, we note that all of the pointers in the level $l - 1$ nodes are Λ, and none of the pointers in the other levels are Λ. This often represents a significant amount of wasted space, so we can save both time and space by eliminating all the Λ's and using a different value of m for all of the "bottom"

nodes. This use of two different m's does not foul up the insertion algorithm, since both halves of a node that is being split remain on the same level as the original node. We could in fact define a generalized B-tree of orders m_1, m_2, m_3, . . . by requiring all nonroot nodes on level $l - i$ to have between $m_i/2$ and m_i sons; such a B-tree has different m's on each level, yet the insertion algorithm still works essentially as before.

To carry the idea in the preceding paragraph even further, we might use a completely different node format in each level of the tree, and we might also store information in the leaves. Sometimes the keys form only a small part of the records in a file, and in such cases it is a mistake to store the entire records in the branch nodes near the root of the tree; this would make m too small for efficient multiway branching.

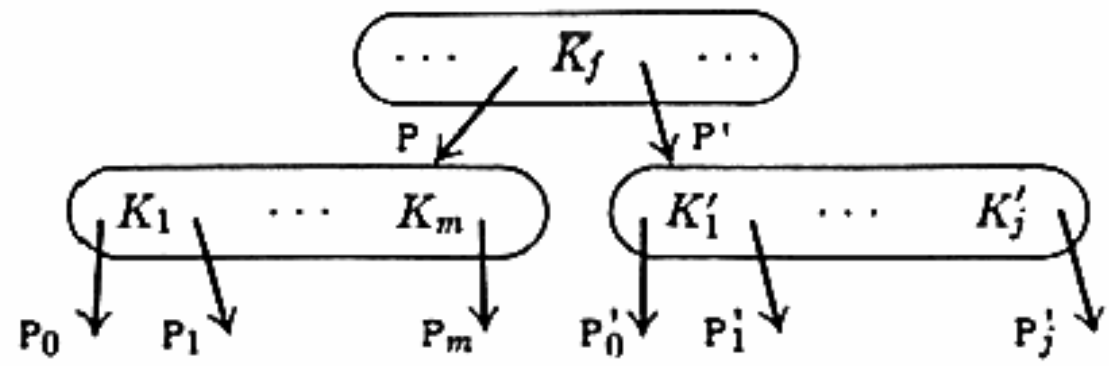
We can therefore reconsider Fig. 30, imagining that all the records of the file are now stored in the leaves, and that only a few of the keys have been duplicated in the branch nodes. Under this interpretation, the leftmost leaf contains all records whose key is ≤ 011; the leaf marked A contains all records whose key satisfies $439 < K \leq 449$; and so on. Under this interpretation the leaf nodes grow and split just as the branch nodes do, except that a record is never passed up from a leaf to the next level. Thus the leaves are always at least half filled to capacity. A new key enters the nonleaf part of the tree whenever a leaf splits. If each leaf is linked to its successor in symmetric order, we gain the ability to traverse the file both sequentially and randomly in an efficient and convenient manner.

Some calculations by S. P. Ghosh and M. E. Senko [*JACM* **16** (1969), 569–579] suggest that it might be a good idea to make the leaves fairly large, say up to about 10 consecutive pages long. By linear interpolation in the known range of keys for each leaf, we can guess which of the 10 pages probably contains a given search argument. If our guess is wrong, we lose time, but experiments indicate that this loss might be less than the time we save by decreasing the size of the tree.

T. H. Martin [unpublished] has pointed out that the idea underlying B-trees can be used also for *variable-length* keys. We need not put bounds $[m/2, m]$ on the number of sons of each node, instead we can say merely that each node should be at least about half full of data. The insertion and splitting mechanism still works fine, even though the exact number of keys per node depends on whether the keys are long or short. However, the keys shouldn't be allowed to get extremely long, or they can mess things up. (See exercise 5.)

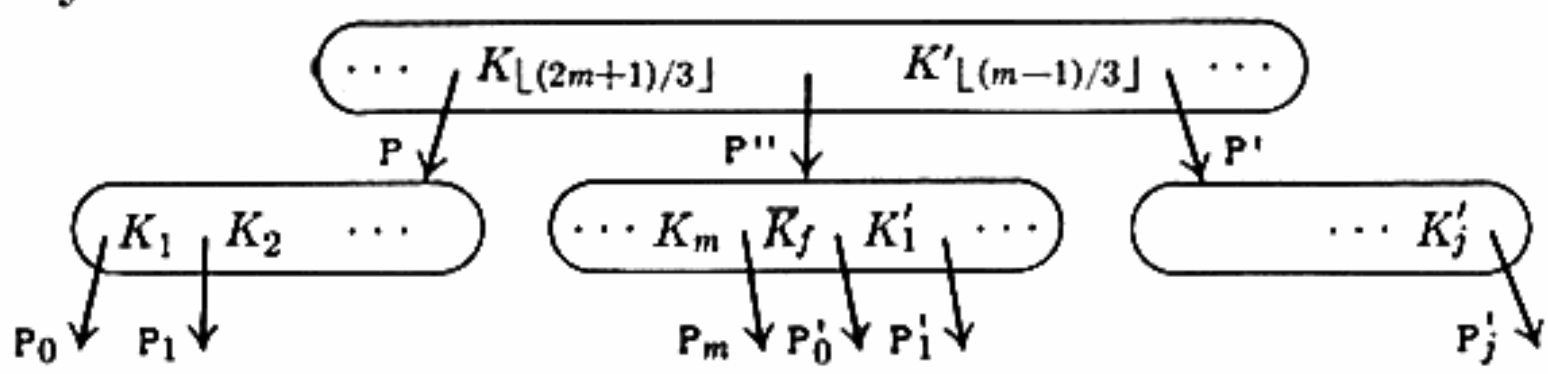
Another important modification to the basic B-tree scheme is the idea of "overflow" introduced by Bayer and McCreight. The idea is to improve the insertion algorithm by resisting its temptation to split nodes so often; a local rotation is used instead. Suppose we have a node that is over-full because it contains m keys and $m + 1$ pointers; instead of splitting it, we can look first at its brother node on the right, which has say j keys and $j + 1$ pointers. In

the father node there is a key $\overline{K}_f$ which separates the keys of the two brothers; schematically,



(8)

If $j < m - 1$, a simple rearrangement makes splitting unnecessary: we leave $\lfloor (m+j)/2 \rfloor$ keys in the left node, we replace $\overline{K}_f$ by $K_{\lfloor (m+j)/2 \rfloor + 1}$ in the father node, and we put the $\lceil (m+j)/2 \rceil$ remaining keys (including $\overline{K}_f$) and the corresponding pointers into the right node. Thus the full node "flows over" into its brother node. On the other hand, if the brother node is already full ($j = m - 1$), we can split *both* of the nodes, making three nodes each about two-thirds full, containing, respectively, $\lfloor (2m-2)/3 \rfloor$, $\lfloor (2m-1)/3 \rfloor$, and $\lfloor 2m/3 \rfloor$ keys:



(9)

If the original node has no right brother, we can look at its left brother in essentially the same way. (If the original node has both a right and a left brother, we could even refrain from splitting off a new node unless *both* left and right brothers are full.) Finally if the original node to be split has no brothers at all, it must be the root; we can change the definition of B-tree, allowing the root to contain as many as $2\lfloor (2m-2)/3 \rfloor$ keys, so that when the root splits it produces two nodes of $\lfloor (2m-2)/3 \rfloor$ keys each.

The effect of all the technicalities in the preceding paragraph is to produce a superior breed of tree, say a B^*-tree of order m, which can be defined as follows:

i) Every node except the root has at most m sons.

ii) Every node, except for the root and the leaves, has $\geq (2m-1)/3$ sons.

iii) The root has at least 2 and at most $2\lfloor (2m-2)/3 \rfloor + 1$ sons.

iv) All leaves appear on the same level.

v) A nonleaf node with k sons contains $k - 1$ keys.

The important change is condition (ii), which asserts that we utilize at least two-thirds of the available space in every node. This change not only uses space more efficiently, it also makes the search process faster, since we may replace "$\lceil m/2 \rceil$" by "$\lceil (2m-1)/3 \rceil$" in (6) and (7).

Perhaps the reader has been skeptical of B-trees because the degree of the root can be as low as 2. Why should we waste a whole disk access on merely a 2-way decision?! A simple buffering scheme, called "least-recently-used page

replacement," overcomes this objection; we can keep several bufferloads of information in the internal memory, so that input commands can be avoided when the corresponding page is already present. Under this scheme, the algorithms for searching or insertion issue "virtual read" commands that are translated into actual input instructions only when the necessary page is not in memory; a subsequent "release" command is issued when the buffer has been read and possibly modified by the algorithm. When an actual read is required, the buffer which has least recently been released is chosen; we write out that buffer, if its contents have changed since they were read in, then we read the desired page into the chosen buffer.

Since the number of levels in the tree is generally small compared to the number of buffers, this paging scheme will ensure that the root page is always present in memory; and if the root has only 2 or 3 sons, the first level pages will probably stay there too. Some special mechanism could be incorporated to ensure that a certain minimum number of pages near the root are always present. Note that the least-recently-used scheme implies that the pages that might need to be split during an insertion are automatically in memory when they are needed.

Some experiments by E. McCreight have shown that this idea is quite successful. For example, he found that with 10 page-buffers and $m = 121$, the process of inserting 100,000 keys in ascending order required only 22 actual read commands, and only 857 actual write commands; thus most of the activity took place in the internal memory. Furthermore the tree contained only 835 nodes, just one higher than the minimum possible value $\lceil 100000/(m-1) \rceil = 834$; thus the storage utilization was nearly 100 percent. For this experiment he used the overflow technique, but with only 2-way node splitting as in (4), not 3-way splitting as in (9). (See exercise 3.)

In another experiment, again with 10 buffers and $m = 121$ and the overflow technique, he inserted 5000 keys into an initially empty tree, in *random* order; this produced a 2-level tree with 48 nodes (87 percent storage utilization), after making 2762 actual reads and 2739 actual writes. Then 1000 random searches required 786 actual reads. The same experiment *without* the overflow feature produced a 2-level tree with 62 nodes (67 percent storage utilization), after making 2743 actual reads and 2800 actual writes; 1000 subsequent random searches required 836 actual reads. This shows not only that the paging scheme is effective but also that it is wise to handle overflows locally before deciding to split a node.

EXERCISES

1. [*10*] What B-tree of order 7 is obtained after the key 613 is inserted into Fig. 30? (Do not use the "overflow" technique.)

2. [*15*] Work exercise 1, but use the overflow technique, with 3-way splitting as in (9).

▸ **3.** [*23*] Suppose we insert the keys 1, 2, 3, . . . in ascending order into an initially empty B-tree of order 101. Which key causes the leaves to be on level 4 for the first time, (a) when we use no overflow; (b) when we use overflow and only 2-way splitting as in (4); (c) when we use a B^*-tree of order 101, with overflow and 3-way splitting as in (9)?

4. [*21*] (Bayer and McCreight.) Explain how to handle insertions into a generalized B-tree so that all nodes except the root and leaves will be guaranteed to have at least $\frac{3}{4}m - \frac{1}{2}$ sons.

▸ **5.** [*21*] Suppose that a node represents 1000 character positions of external memory. If each pointer takes up 5 characters, and if the keys are variable in length, between 5 and 50 characters long, what is the minimum number of character positions occupied in a node after it splits during an insertion? (Consider only a simple splitting procedure analogous to that described in the text for fixed-length-key B-trees, without "overflowing.")

6. [*22*] Can the B-tree idea be used to retrieve items of a linear list by position instead of by key value? (Cf. Algorithm 6.2.3B)

7. [*23*] Design a deletion algorithm for B-trees.

8. [*28*] Design a concatenation algorithm for B-trees (cf. Section 6.2.3).

9. [*30*] Discuss how a large file, organized as a B-tree, can be used for multiple accessing and updating by a large number of simultaneous users, in such a way that users of different pages rarely interfere with each other.

10. [*HM37*] Consider the generalization of tree insertion suggested by Muntz and Uzgalis, where each page can hold M keys. After N random items have been inserted into such a tree, so that there are $N+1$ external nodes, let $b_{Nk}^{(\mathcal{O})}$ be the probability that an unsuccessful search requires k page accesses and that it ends at an external node whose father node belongs to a page containing j keys. If $B_N^{(\mathcal{O})}(z) = \sum b_{Nk}^{(\mathcal{O})} z^k$ is the corresponding generating function, prove that $B_1^{(\mathcal{O})}(z) = \delta_{j1} z$;

$$B_N^{(\mathcal{O})}(z) = \frac{N-j-1}{N+1} B_{N-1}^{(\mathcal{O})}(z) + \frac{j+1}{N+1} B_{N-1}^{(\mathcal{O}-1)}(z), \quad \text{for} \quad 1 < j < M;$$

$$B_N^{(1)}(z) = \frac{N-2}{N+1} B_{N-1}^{(1)}(z) + \frac{2z}{N+1} B_{N-1}^{(M)}(z);$$

$$B_N^{(M)}(z) = \frac{N-1}{N+1} B_{N-1}^{(M)}(z) + \frac{M+1}{N+1} B_{N-1}^{(M-1)}(z).$$

Find the asymptotic behavior of $C'_N = \sum_{1 \le j \le M} B_N^{(\mathcal{O})\prime}(1)$, the average number of page accesses per unsuccessful search. [*Hint:* Express the recurrence in terms of the matrix

$$\mathbf{W}(z) = \begin{pmatrix} -3 & 0 & \dots & 0 & 2z \\ 3 & -4 & \dots & 0 & 0 \\ 0 & 4 & \dots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \dots & -M-1 & 0 \\ 0 & 0 & \dots & M+1 & -2 \end{pmatrix},$$

and relate C'_N to an Nth degree polynomial in $\mathbf{W}(1)$.]

Instead of basing a search method on comparisons between keys, we can make use of their representation as a sequence of digits or alphabetic characters. Consider, for example, the "thumb-index" on a large dictionary; from the first letter of a given word, we can immediately locate the pages which contain all words beginning with that letter.

If we pursue the thumb-index idea to one of its logical conclusions, we come up with a searching scheme based on repeated "subscripting" as illustrated in Table 1. Suppose that we want to test a given search argument to see whether it is one of the 31 most common words of English (cf. Figs. 12 and 13 in Section 6.2.2). The data is represented in Table 1 as a so-called "trie" structure; this name was suggested by E. Fredkin [*CACM* **3** (1960), 490–500] because it is a part of information re*trie*val. A trie is essentially an M-ary tree, whose nodes are M-place vectors with components corresponding to digits or characters. Each node on level l represents the set of all keys that begin with a certain sequence of l characters; the node specifies an M-way branch, depending on the $(l + 1)$st character.

For example, the trie of Table 1 has 12 nodes; node (1) is the root, and we look up the first letter here. If the first letter is, say, N, the table tells us that our word must be NOT (or else it isn't in the table). On the other hand, if the first letter is W, node (1) tells us to go on to node (9), looking up the second letter in the same way; node (9) says that the second letter should be A, H, or I.

The node vectors in Table 1 are arranged according to MIX character code. This means that a trie search will be quite fast, since we are merely fetching words of an array by using the characters of our keys as subscripts. Techniques for making quick multiway decisions by subscripting have been called "Table Look-At" as opposed to "Table Look-Up" [see P. M. Sherman, *CACM* **4** (1961), 172–173, 175].

Algorithm T (*Trie search*). Given a table of records which form an M-ary trie, this algorithm searches for a given argument K. The nodes of the trie are vectors whose subscripts run from 0 to $M - 1$; each component of these vectors is either a key or a link (possibly null).

T1. [Initialize.] Set the link variable P so that it points to the root of the tree.

T2. [Branch.] Set k to the next character of the input argument, K, from left to right. (If the argument has been completely scanned, we set k to a "blank" or end-of-word symbol. The character should be represented as a number in the range $0 \le k < M$.) Let X be table entry number k in NODE(P). If X is a link, go to T3; but if X is a key, go to T4.

T3. [Advance.] If $X \neq \Lambda$, set P $\leftarrow X$ and return to step T2; otherwise the algorithm terminates unsuccessfully.

T4. [Compare.] If $K = X$, the algorithm terminates successfully; otherwise it terminates unsuccessfully. ∎

Table 1

A 'TRIE' FOR THE 31 MOST COMMON ENGLISH WORDS

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
␣	—	A	—	—	—	I	—	—	—	—	HE	—
A	(2)	—	—	—	(10)	—	—	—	WAS	—	—	THAT
B	(3)	—	—	—	—	—	—	—	—	—	—	—
C	—	—	—	—	—	—	—	—	—	—	—	—
D	—	—	—	—	—	—	—	—	—	HAD	—	—
E	—	—	BE	—	(11)	—	—	—	—	—	—	THE
F	(4)	—	—	—	—	—	OF	—	—	—	—	—
G	—	—	—	—	—	—	—	—	—	—	—	—
H	(5)	—	—	—	—	—	—	(12)	WHICH	—	—	—
I	(6)	—	—	—	HIS	—	—	—	WITH	—	—	THIS
Θ	—	—	—	—	—	—	—	—	—	—	—	—
J	—	—	—	—	—	—	—	—	—	—	—	—
K	—	—	—	—	—	—	—	—	—	—	—	—
L	—	—	—	—	—	—	—	—	—	—	—	—
M	—	—	—	—	—	—	—	—	—	—	—	—
N	NOT	AND	—	—	—	IN	ON	—	—	—	—	—
O	(7)	—	—	FOR	—	—	—	TO	—	—	—	—
P	—	—	—	—	—	—	—	—	—	—	—	—
Q	—	—	—	—	—	—	—	—	—	—	—	—
R	—	ARE	—	FROM	—	—	OR	—	—	—	HER	—
Φ	—	—	—	—	—	—	—	—	—	—	—	—
Π	—	—	—	—	—	—	—	—	—	—	—	—
S	—	AS	—	—	—	IS	—	—	—	—	—	—
T	(8)	AT	—	—	—	IT	—	—	—	—	—	—
U	—	—	BUT	—	—	—	—	—	—	—	—	—
V	—	—	—	—	—	—	—	—	—	HAVE	—	—
W	(9)	—	—	—	—	—	—	—	—	—	—	—
X	—	—	—	—	—	—	—	—	—	—	—	—
Y	YOU	—	BY	—	—	—	—	—	—	—	—	—
Z	—	—	—	—	—	—	—	—	—	—	—	—

Note that if the search is unsuccessful, the *longest match* has been found. This property is occasionally useful in applications.

In order to compare the speed of this algorithm to the others in this chapter, we can write a short MIX program assuming that the characters are bytes and that the keys are at most five bytes long.

Program T (*Trie search*). This program assumes that all keys are represented in one MIX word, with blank spaces at the right whenever the key has less than

five characters. Since we use the MIX character code, each byte of the search argument is assumed to contain a number less than 30. Links are represented as negative numbers in the 0:2 field of a node word. rI1 ≡ P, rX ≡ unscanned part of K.

```
01    START   LDX    K            1    T1. Initialize.
02            ENT1   ROOT         1    P ← pointer to root of trie.
03    2H      SLAX   1            C    T2. Branch.
04            STA    *+1(2:2)     C    Extract next character, k.
05            ENT2   0,1          C    Q ← P + k.
06            LD1N   0,2(0:2)     C    P ← LINK(Q).
07            J1P    2B           C    T3. Advance. To T2 if P is a link ≠ Λ.
08            LDA    0,2          1    T4. Compare. rA ← KEY(Q).
09            CMPA   K            1
10            JE     SUCCESS      1    Exit successfully if rA = K.
11    FAILURE EQU    *                 Exit if not in the trie. ∎
```

The running time of this program is $8C + 8$ units, where C is the number of characters examined. Since $C \leq 5$, the search will never take more than 48 units of time.

If we now compare the efficiency of this program (using the trie of Table 1) to Program 6.2.2T (using the *optimum* binary search tree of Fig. 13), we can make the following observations:

1. The trie takes much more memory space; we are using 360 words just to represent 31 keys, while the binary search tree uses only 62 words of memory. (However, exercise 4 shows that, with some fiddling around, we can actually fit the trie of Table 1 into only 55 words.)

2. A successful search takes about 26 units of time for both programs. But an unsuccessful search will go faster in the trie, slower in the binary search tree. For this data the search will be unsuccessful more often than it is successful, so the trie is preferable from the standpoint of speed.

3. If we consider the KWIC indexing application of Fig. 15 instead of the 31 commonest English words, the trie loses its advantage because of the nature of the data. For example, a trie requires 12 iterations to distinguish between COMPUTATION and COMPUTATIONS. (In this case it would be better to build the trie so that words are scanned from right to left instead of from left to right.)

The idea of trie memory was first published by Rene de la Briandais [*Proc. Western Joint Computer Conf.* **15** (1959), 295–298]. He pointed out that we can save memory space at the expense of running time if we use a linked list for each node vector, since most of the entries in the vectors tend to be empty. In effect, this idea amounts to replacing the trie of Table 1 by the forest of trees shown in Fig. 31. Searching in such a forest proceeds by finding the root which matches the first character, then finding the son node of that root which matches the second character, etc.

In his article, de la Briandais did not actually stop the tree branching exactly as shown in Table 1 or Fig. 31; instead, he continued to represent each key, character by character, until reaching the end-of-word delimiter. Thus he

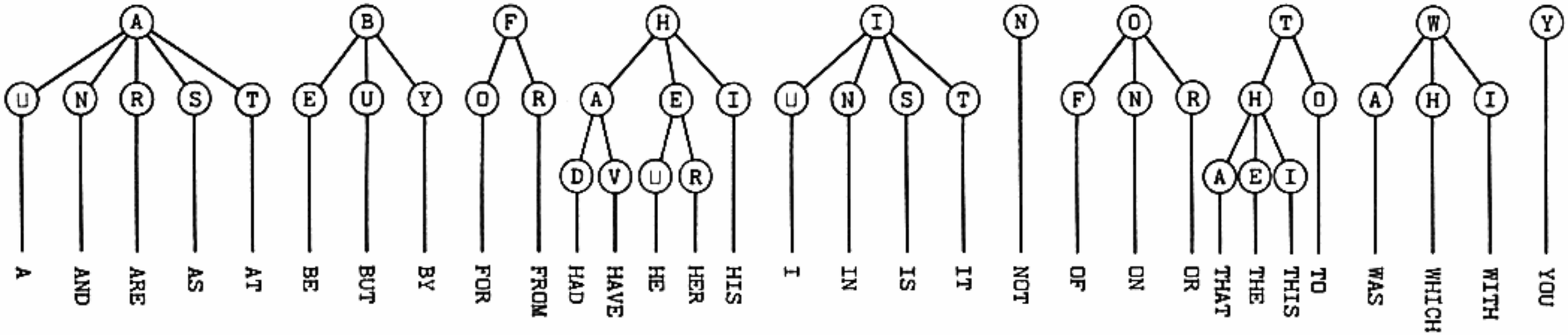
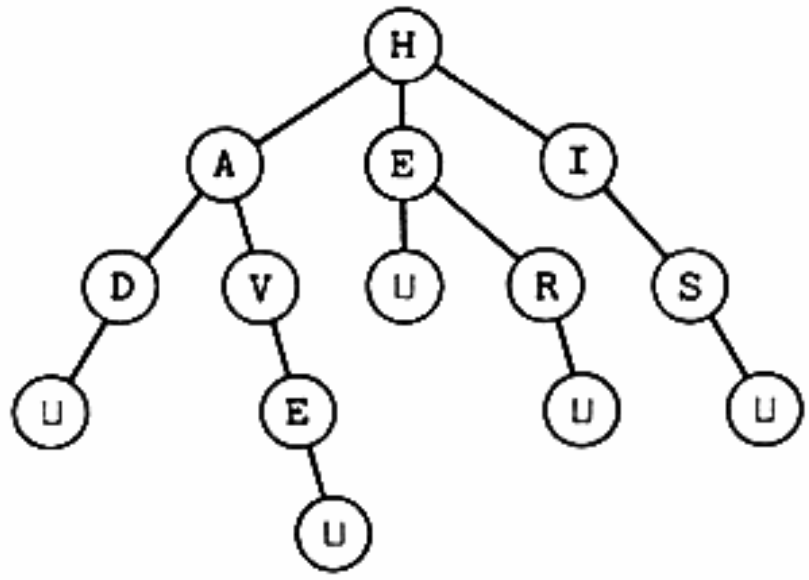


Fig. 31. The trie of Table 1, converted into a "forest."

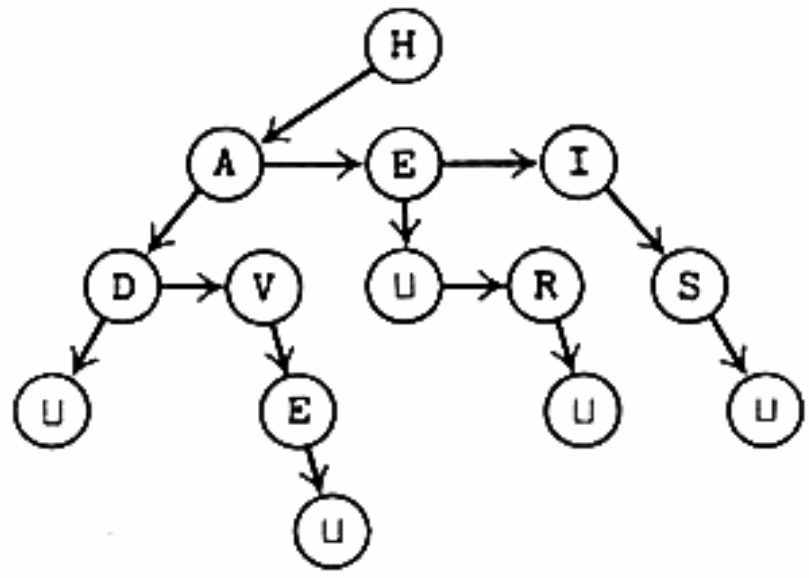
would actually have used



(1)

in place of the "H" tree in Fig. 31. This representation requires more storage, but it makes the processing of variable-length data especially easy. If we use two link fields per character, dynamic insertions and deletions can be handled in a simple manner. Furthermore there are many applications in which the search argument appears in "unpacked form," one character per word, and such a tree makes it unnecessary to pack the data before conducting the search.

If we use the normal way of representing trees as binary trees, (1) becomes the binary tree



(2)

(In the representation of the full forest, Fig. 31, we would also have a pointer leading to the right from H to its neighboring root I.) The search in this binary tree proceeds by comparing a character of the argument to the character in the tree, and following RLINKs until finding a match; then the LLINK is taken and we treat the next character of the argument in the same way.

With such a binary tree, we are more or less doing a search by comparison, with equal-unequal branching instead of less-greater branching. The elementary theory of Section 6.2.1 tells that we must make at least $\log_2 N$ comparisons, on the average, to distinguish between N keys; the average number of tests made when searching a tree like that of Fig. 31 must be at least as many as we make when doing a binary search using the techniques of Section 6.2.

On the other hand, the trie in Table 1 is capable of making an M-way branch all at once; we shall see that the average search time for large N involves only about $\log_M N = \log_2 N/\log_2 M$ iterations, if the input data is random. We shall also see that a "pure" trie scheme like that in Algorithm T requires a total of approximately $N/\ln M$ nodes to distinguish between N random inputs; hence the total amount of space is proportional to $MN/\ln M$.

From these considerations it is clear that the trie idea pays off only in the first few levels of the tree. We can get better performance by mixing two strategies, using a trie for the first few characters and then switching to some other technique. For example, E. H. Sussenguth, Jr. [*CACM* **6** (1963), 272–279] has suggested using a character-by-character scheme until we reach part of the tree where only, say, six or less keys of the file are possible, and then we can sequentially run through the short list of remaining keys. We shall see that this mixed strategy decreases the number of trie nodes by roughly a factor of six, without substantially changing the running time.

An application to English. Many variations on the basic trie and character search strategies suggest themselves. In order to get a feeling for some of these possibilities, let us consider a hypothetical large-scale application: Suppose that we want to store a fairly complete dictionary of the English language in the memory of our computer. For this purpose we will, of course, need a reasonably large internal memory, say 50,000 words. Our goal is to find a compact way to represent the dictionary, yet to keep the searching reasonably fast.

Such a project is obviously no small task; it may be expected to require a good knowledge of the contents of the dictionary as well as considerable programming ingenuity. For the moment, let us try to put ourselves in the position of someone embarking on such a major project.

A typical college dictionary contains over 100,000 words; this is somewhat larger than contemplated here, but a glance through such a dictionary will give us some idea of what to expect. If we try to apply the trie memory approach, we soon notice that important simplifications can be made. For example, suppose that we discount proper names and abbreviations. Then if the first letter of a word is b, the second letter will never be any of the characters b, c, d, f, g, j, k, m, n, p, q, s, t, v, w, x, or z (*except* for the word "bdellium," which we might choose to leave out of our dictionary!). In fact, the same 17 possibilities are excluded as second letters in words starting with c, d, f, g, h, j, k, l, m, n, p, q, r, t, v, w, x, z, except for a few words starting with ct, cz, dw, fj, gn, mn, nt, pf, pn, pt, tc, tm, ts, tz, zw and a fair number of words which begin with kn, ps, tw. One way to make use of this fact is to encode the letters (e.g., to have a 26-word table and to perform the equivalent of "`LDI TABLE,1`"), so that the consonants b, c, d, . . . , z listed above are converted into a special representation greater than the numeric code for the remaining letters a, e, h, i, l, o, r, u, y. In this way, many nodes of a trie can be shortened to a 9-way branch, with another "escape" cell to be used for the rarely occurring exceptional letters. This will save memory space in many parts of a trie for English, not just in the second letter position.

Of the $26^2 = 676$ possible combinations of two letters, only 309 actually occur at the beginning of words in a typical college dictionary; and of these 309 pairs, 88 are the initial letters of 15 or less words. (Typical examples of these 88 rare pairs are aa, ah, aj, ak, ao, aq, ay, az, bd, bh, . . . , xr, yc, yi, yp,

yt, yu, yw, za, zu, zw; most people can't name more than one word from most of these categories.) When one of the rare categories occurs, we can shift from trie memory to some other scheme like a sequential search.

Another way to cut down the storage requirement for a dictionary is to make use of prefixes. For example, if we are looking up a word that begins with re-, pre-, anti-, trans-, dis-, un-, etc., we might wish to detach the prefix and look up the remainder of the word. In this way we can remove many words like reapply, recompute, redecorate, redesign, redeposit, etc.; but we still need to retain words like remainder, requirement, retain, remove, readily, etc., since their meaning is not readily deducible when the prefix "re-" is suppressed. Thus we should first look up the word and then try deleting the prefix only if the first search fails.

The use of suffixes is even more important than the use of prefixes. It would certainly be wasteful to incorporate each noun and verb twice, in both singular and plural form; and there are many other types of suffixes. For example, the following endings may be added to many verb stems, to make a family of related words:

-e	-es	-ing
-ed	-s	-ings
-edly	-able	-ingly
-er	-ible	-ion
-or	-ably	-ions
-ers	-ibly	-ional
-ors	-ability	-ionally
	-abilities	

(Many of these suffixes are themselves composed of suffixes.) If we try to apply these suffixes to the stems

comput-
calculat-
search-
suffix-
translat-
interpret-
confus-

we see that a great many words are formed; this greatly multiplies the capacity of our dictionary. Of course a lot of nonwords are formed also, e.g. "compution"; the stem computat- seems to be necessary as well as comput-. But this causes no harm since such combinations will never appear in the input anyway; and if some author chooses to coin the word "computedly," we will have a ready-made translation of it for him. Note that most people would understand the word "confusability," although it appears in few dictionaries; our dictionary

will give a suitable interpretation. If prefixes and suffixes are correctly handled, our dictionary may even be able to deduce the meaning of "antidisestablishmentarianism," given only the verb stem "establish."

Of course we must be careful that the meaning of each word is properly determined by its stem and suffix; if not, the exceptions should be entered into the dictionary so that they will be found before we attempt to look for a suffix. For example, the analogy between the words "socialism" and "socialist" is not at all the same as between "organism" and "organist"!

These are some of the tricks that can be used to reduce the amount of memory needed. But how shall we represent this hodge-podge of methods compactly in a single system? The answer is to think of the dictionary as a *program* written in a special machine language for a special *interpretive system* (cf. Section 1.4.3); the entries within each node of a trie can be thought of as *instructions*. For example, in Table 1 we have two kinds of "instructions," and Program T uses the sign bit as the "op-code"; a minus sign means branch to another node and advance to the next character for the next instruction, while a plus sign means that the argument is supposed to be compared to a specified key.

We might have the following types of op-codes in the interpretive language for our dictionary application:

- Test n, α, β. "If the next character of the argument has an encoded value $k \le n$, go to location $\alpha + k$ for the next instruction; otherwise go to location β."
- Compare n, α, β. "Compare the remaining characters of the argument to the n words stored in locations α, $\alpha + 1$, . . . , $\alpha + n - 1$. If a match is found in location $\alpha + k$, the search terminates successfully with 'meaning' $\beta + k$, but if no match is found, it terminates unsuccessfully."
- Split α, β. "The word scanned up to this point is a possible prefix or stem. Continue searching by going to location α for the next instruction. If that search is unsuccessful, continue searching by looking up the remaining characters of the argument as if they were a new argument. If this second search is successful, combine the 'meaning' found with the 'meaning' β."

The test operation is essentially the trie search concept, and the compare operation denotes a changeover to sequential searching. The split operation handles both prefixes and suffixes. Several other operations can be envisioned based on further idiosyncrasies of English. It would be possible to save further memory space by eliminating the parameter β from each instruction, since the memory can be arranged so that β is implied by α, n, or the location of the instruction, in each case.

For additional information about dictionary organization, see the interesting articles by Sydney M. Lamb and William H. Jacobsen, Jr., *Mechanical*

Translation **6** (1961), 76–107; Eugene S. Schwartz, *JACM* **10** (1963), 413–439; E. J. Galli and H. Yamada, *IBM Systems J.* **6** (1967), 192–207.

The binary case. Let us now consider the special case $M = 2$, in which we scan the search argument one bit at a time. Two interesting methods have been developed which are especially appropriate for this case.

The first method, which we shall call *digital tree search*, is due to E. G. Coffman and J. Eve [*CACM* **13** (1970), 427–432, 436]. The idea is to store full keys in the nodes just as we did in the tree search algorithm of Section 6.2.2, but to use bits of the argument (instead of results of the comparisons) to govern whether to take the left or right branch at each step. Figure 32 shows the tree constructed by this method when we insert the 31 most common English words in order of decreasing frequency. In order to provide binary data for this illustration, the words have been expressed in MIX character code which was then converted into binary numbers with 5 bits per byte. Thus, the word WHICH is represented as "11010 01000 01001 00011 01000".

To search for this word WHICH in Fig. 32, we compare it first with the word THE at the root of the tree. Since there is no match and since the first bit of WHICH is 1, we move to the right and compare with OF. Since there is no match and since the second bit of WHICH is 1, we move to the right and compare with WITH; and so on.

It is interesting to note the contrast between Fig. 32 and Fig. 12 in Section 6.2.2, since the latter tree was formed in the same way but using comparisons instead of key bits for the branching. If we consider the given frequencies, the digital search tree of Fig. 32 requires an average of 3.42 comparisons per successful search; this is somewhat better than the 4.04 comparisons needed by Fig. 12, although of course the computing time per comparison will probably be different.

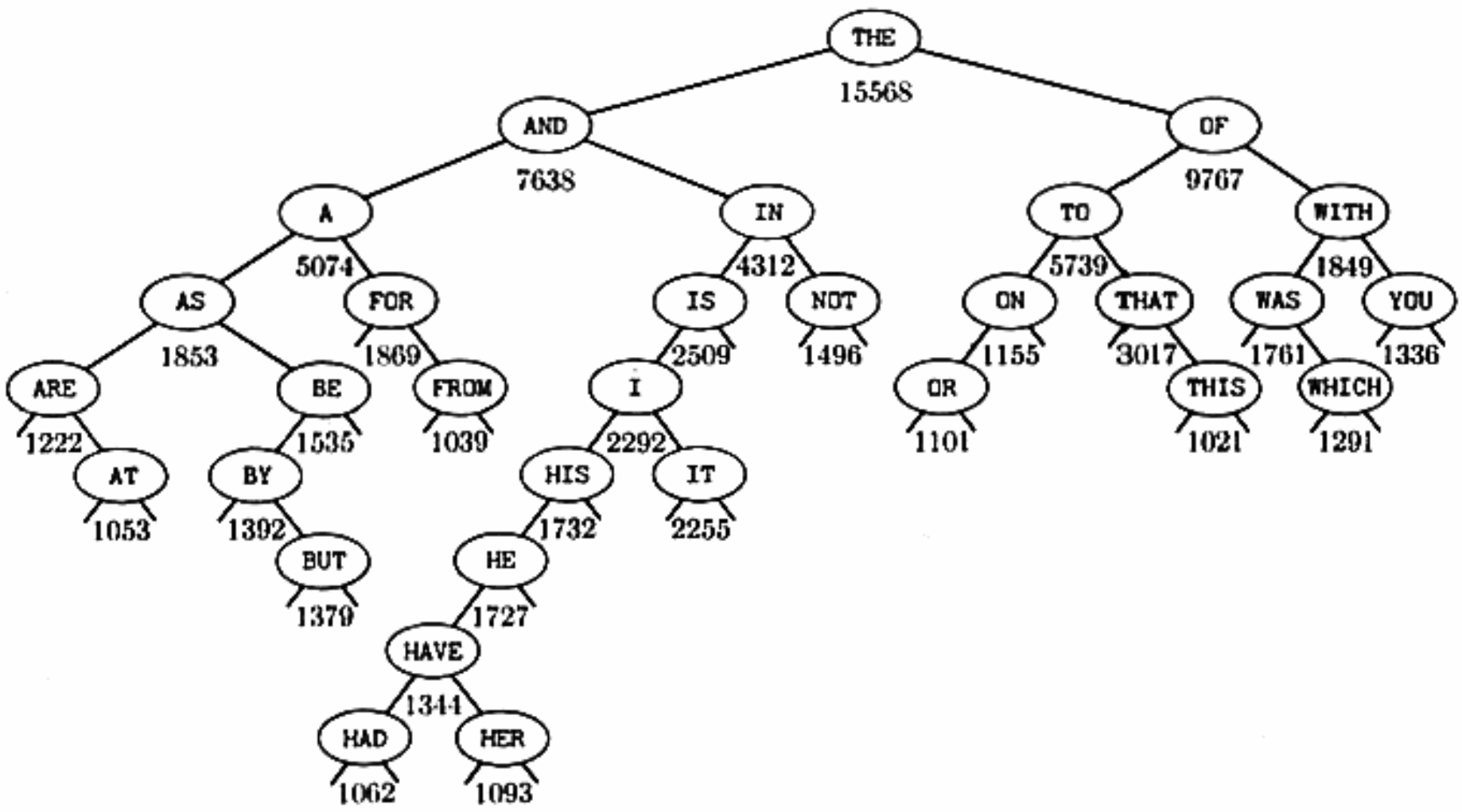


Fig. 32. A digital search tree for the 31 most common English words, inserted in decreasing order of frequency.

Algorithm D (*Digital tree search*). Given a table of records which form a binary tree as described above, this algorithm searches for a given argument K. If K is not in the table, a new node containing K is inserted into the tree in the appropriate place.

This algorithm assumes that the nodes of the tree have KEY, LLINK, and RLINK fields just as in Algorithm 6.2.2T. In fact, the two algorithms are almost identical, as the reader may verify.

D1. [Initialize.] Set P $\leftarrow$ ROOT, and $K1 \leftarrow K$.

D2. [Compare.] If $K =$ KEY(P), the search terminates successfully. Otherwise set b to the leading bit of $K1$, and shift $K1$ left one place (thereby removing that bit and introducing a 0 at the right). If $b = 0$, go to D3, otherwise go to D4.

D3. [Move left.] If LLINK(P) $\neq \Lambda$, set P $\leftarrow$ LLINK(P) and go back to D2. Otherwise go to D5.

D4. [Move right.] If RLINK(P) $\neq \Lambda$, set P $\leftarrow$ RLINK(P) and go back to D2.

D5. [Insert into tree.] Set Q $\Leftarrow$ AVAIL, KEY(Q) $\leftarrow K$, LLINK(Q) $\leftarrow$ RLINK(Q) $\leftarrow \Lambda$. If $b = 0$ set LLINK(P) $\leftarrow$ Q, otherwise set RLINK(P) $\leftarrow$ Q. ∎

Although the tree search of Algorithm 6.2.2T is inherently binary, it is not difficult to see that the present algorithm could be extended to an M-ary digital search for any $M \geq 2$ (see exercise 13).

Donald R. Morrison [*JACM* **15** (1968), 514–534] has discovered a very pretty way to form N-node search trees based on the binary representation of keys, *without* storing keys in the nodes. His method, called "Patricia" (Practical Algorithm To Retrieve Information Coded In Alphanumeric), is especially suitable for dealing with extremely long, variable-length keys such as titles or phrases stored within a large bulk file.

Patricia's basic idea is to build a binary trie, but to avoid one-way branching by including in each node the number of bits to skip over before making the next test. There are several ways to exploit this idea; perhaps the simplest to explain is illustrated in Fig. 33. We have a TEXT array of bits, which is usually quite long; it may be stored as an external direct-access file, since each search accesses TEXT only once. Each key to be stored in our table is specified by a starting place in the text, and it can be imagined to go from this starting place all the way to the end of the text. (Patricia does not search for strict equality between key and argument, rather it will determine whether or not there exists a key *beginning* with the argument.)

The situation depicted in Fig. 33 involves seven keys, one starting at each word, namely "THIS IS THE HOUSE THAT JACK BUILT." and "IS THE HOUSE THAT JACK BUILT." and . . . and "BUILT.". There is one important restriction, namely that *no one key may be a prefix of another*; this restriction can be met if we end the text with a unique end-of-text code (in this case ".") that appears

```
          1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
TEXT:     T H I S   I S   T H E   H O U S E   T H A T   J A C K   B U I L T .
```

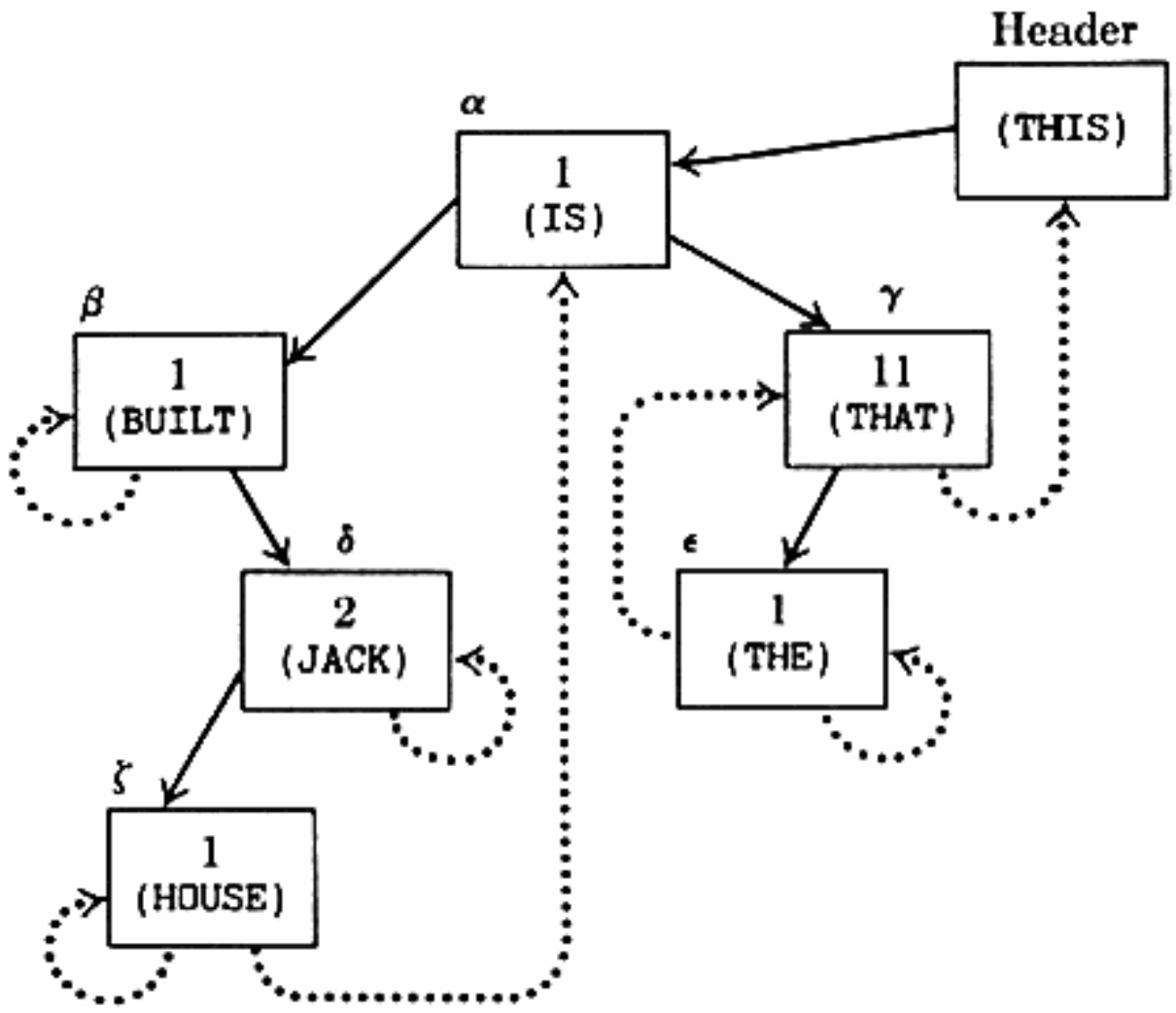


Fig. 33. An example of Patricia's tree and TEXT.

nowhere else. The same restriction was implicit in the trie scheme of Algorithm T, where "␣" was the termination code.

The tree which Patricia uses for searching should be contained in random-access memory, or it should be arranged on pages as suggested in Section 6.2.4. It consists of a header and $N - 1$ nodes, where the nodes contain several fields:

KEY, a pointer to the text. This field must be at least $\log_2 C$ bits long, if the text contains C characters. In Fig. 33 the words shown within each node would really be represented by pointers to the text, e.g. instead of "(JACK)" the node contains the number 24 (the starting place of "JACK BUILT.").

LLINK and RLINK, pointers within the tree. These fields must be at least $\log_2 N$ bits long.

LTAG and RTAG, one-bit fields which tell whether or not LLINK and RLINK, respectively, are pointers to sons or to ancestors of the node. The dotted lines in Fig. 33 correspond to pointers whose TAG bit is 1.

SKIP, a number which tells how many bits to skip when searching, as explained below. This field should be large enough to hold the largest number k such that identical k-bit substrings occur in two different keys; in practice, we may usually assume that k isn't too large, and an error indication can be given if the size of the SKIP field is exceeded. The SKIP fields are shown as numbers within each node of Fig. 33.

The header contains only KEY, LLINK, and LTAG fields.

A search in Patricia's tree is carried out as follows: Suppose we are looking up the word THAT (bit pattern 10111 01000 00001 10111). We start by looking at the SKIP field of the root node, which tells us to examine the first bit of the argument. It is 1, so we move to the right. The SKIP field in the next node tells us to look at the $1 + 11 = 12$th bit of the argument. It is 0, so we move to the left. The SKIP field of the next node tells us to look at the $(12 + 1)$st bit, which is 0; now we find LTAG = 1, so we go to node γ which refers us to the TEXT. The search path we have taken would occur for any argument whose bit pattern is 1xxxx xxxxx x00 . . . , and we must check to see if it matches the unique key which begins with that pattern.

Suppose, on the other hand, that we are looking for any or all keys starting with TH. The search process begins as above, but it eventually tries to look at the (nonexistent) 12th bit of the 10-bit argument. At this point we compare the argument to the TEXT at the point specified in the current node (in this case node γ); if it does not match, the argument is not the beginning of any key, but if it does match, the argument is the beginning of every key represented by dotted links in node γ and its descendants.

This process can be spelled out more precisely as follows.

Algorithm P (*Patricia*). Given a TEXT array and a tree with KEY, LLINK, RLINK, LTAG, RTAG, and SKIP fields as described above, this algorithm determines whether or not there is a key in the TEXT which begins with a specified argument K. (If r such keys exist, for $r \geq 1$, it is subsequently possible to locate them all in $O(r)$ steps; see exercise 14.) We assume that at least one key is present.

P1. [Initialize.] Set P ← HEAD and $j \leftarrow 0$. (Variable P is a pointer which will move down the tree, and j is a counter which will designate bit positions of the argument.) Set $n \leftarrow$ number of bits in K.

P2. [Move left.] Set Q ← P and P ← LLINK(Q). If LTAG(Q) = 1, go to P6.

P3. [Skip bits.] (At this point we know that if the first j bits of K match any key whatsoever, they match the key which starts at KEY(P).) Set $j \leftarrow j +$ SKIP(P). If $j > n$, go to P6.

P4. [Test bit.] (At this point we know that if the first $j - 1$ bits of K match any key, they match the key starting at KEY(P).) If the jth bit of K is 0, go to P2, otherwise go to P5.

P5. [Move right.] Set Q ← P and P ← RLINK(Q). If RTAG(Q) = 0, go to P3.

P6. [Compare.] (At this point we know that if K matches any key, it matches the key starting at KEY(P).) Compare K to the key starting at position KEY(P) in the TEXT array. If they are equal (up to n bits, the length of K), the algorithm terminates successfully; if unequal, it terminates unsuccessfully. ∎

Exercise 15 shows how Patricia's tree can be built in the first place. We can also add to the text and insert new keys, provided that the new text material always ends with a unique delimiter (e.g., an end-of-text symbol followed by a serial number).

Patricia is a little tricky, and she requires careful scrutiny before all of her beauties are revealed.

Analyses of the algorithms. We shall conclude this section by making a mathematical study of tries, digital search trees, and Patricia. A summary of the main consequences of these analyses appears at the very end.

Let us consider first the case of binary tries, i.e., tries with $M = 2$. Figure 34 shows the binary trie that is formed when the sixteen keys from the sorting examples of Chapter 5 are treated as 10-bit binary numbers. (The keys are shown in octal notation, so that for example *1144* represents the 10-bit number $(1001100100)_2$.) As in Algorithm T, we use the trie to store information about the leading bits of the keys until we get to the first point where the key is uniquely identified; then the key is recorded in full.

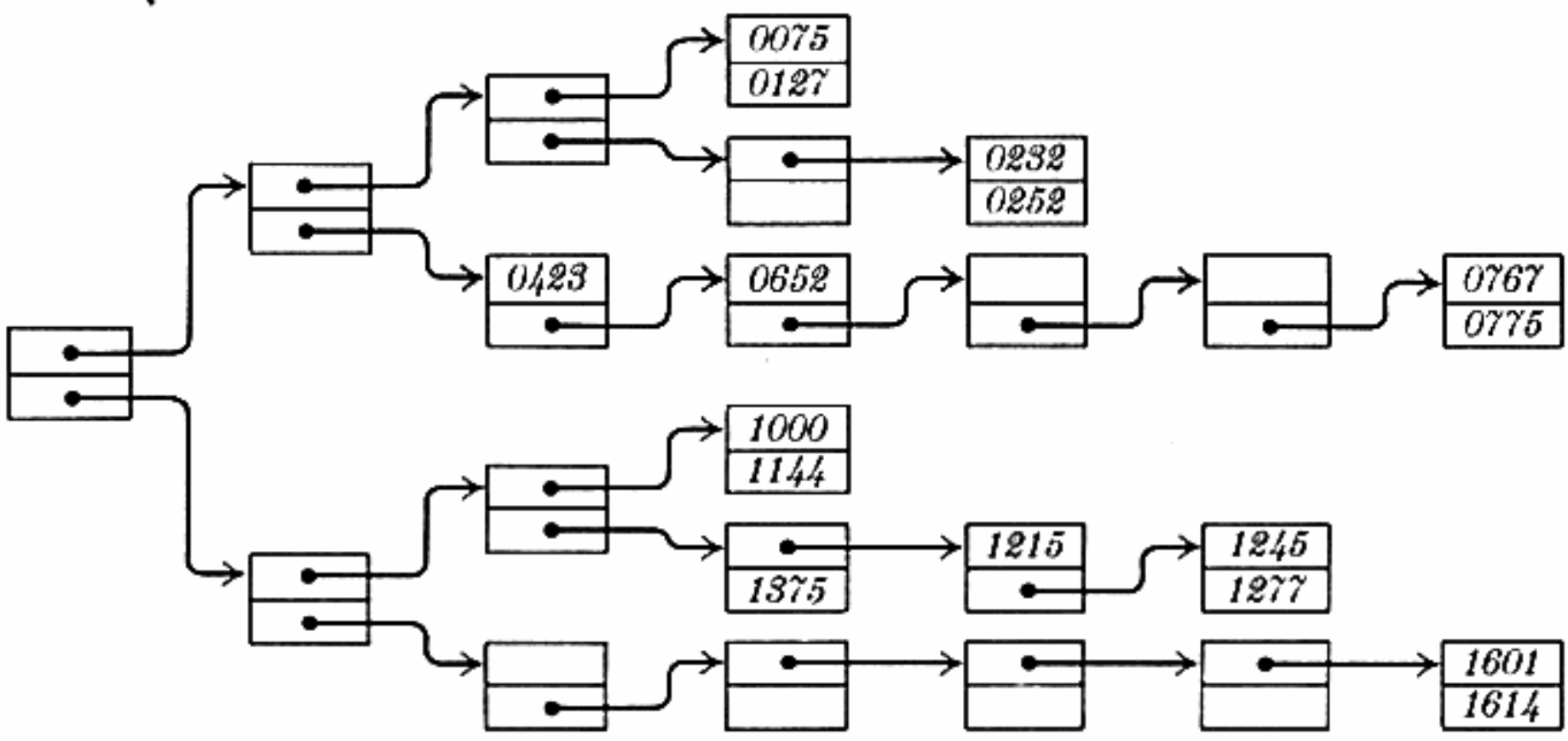


Fig. 34. Example of a random binary trie.

If Fig. 34 is compared to Table 5.2.2–3, an amazing relationship between trie memory and radix-exchange sorting is revealed. (Then again, perhaps this relationship is obvious.) The 22 nodes of Fig. 34 correspond precisely to the 22 partitioning stages in Table 5.2.2–3, with the pth node in preorder corresponding to Stage p. The number of bit inspections in a partitioning stage is equal to the number of keys within the corresponding node and its subtries; consequently we may state the following result.

Theorem T. *If N distinct binary numbers are put into a binary trie as described above, then* (i) *the number of nodes of the trie is equal to the number of partitioning stages required if these numbers are sorted by radix-exchange; and* (ii) *the average number of bit inspections required to retrieve a key by means of Algorithm T is $1/N$ times the number of bit inspections required by the radix-exchange sort.* ▮

Because of this theorem, we can make use of all the mathematical machinery that was developed for radix exchange in Section 5.2.2. For example, if we assume that our keys are infinite-precision random uniformly distributed real numbers between 0 and 1, the number of bit inspections needed for retrieval will be $\log_2 N + \gamma/(\ln 2) + 1/2 + f(N) + O(N^{-1})$, and the number of trie nodes will be $N/(\ln 2) + Ng(N) + O(1)$. Here $f(N)$ and $g(N)$ are complicated functions which may be neglected since their value is always less than 10^{-6} (see exercises 5.2.2–38, 48).

Of course there is still more work to be done, since we need to generalize from binary tries to M-ary tries. We shall describe only the starting point of the investigations here, leaving the instructive details as exercises.

Let A_N be the average number of nodes in a random M-ary search trie that contains N keys. Then $A_0 = A_1 = 0$, and for $N \geq 2$ we have

$$A_N = 1 + \sum_{k_1+\cdots+k_M=N} \left(\frac{N!}{k_1! \ldots k_M!} M^{-N}\right) (A_{k_1} + \cdots + A_{k_M}), \qquad (3)$$

since $N!M^{-N}/k_1! \ldots k_M!$ is the probability that k_1 of the keys are in the first subtrie, . . . , k_M in the Mth. This equation can be rewritten

$$A_N = 1 + M^{1-N} \sum_{k_1+\cdots+k_M=N} \left(\frac{N!}{k_1! \ldots k_M!}\right) A_{k_1}$$

$$= 1 + M^{1-N} \sum_k \binom{N}{k} (M-1)^{N-k} A_k, \qquad \text{for} \qquad N \geq 2, \qquad (4)$$

by using symmetry and then summing over $k_2, \ldots, k_M$. Similarly, if C_N denotes the average total number of bit inspections needed to look up all N keys in the trie, we find $C_0 = C_1 = 0$ and

$$C_N = N + M^{1-N} \sum_k \binom{N}{k} (M-1)^{N-k} C_k \qquad \text{for} \qquad N \geq 2. \qquad (5)$$

Exercise 17 shows how to deal with general recurrences of this type, and exercises 18–25 work out the corresponding theory of random tries. [The analysis of A_N was first approached from another point of view by L. R. Johnson and M. H. McAndrew, *IBM J. Res. and Devel.* **8** (1964), 189–193, in connection with an equivalent hardware-oriented sorting algorithm.]

If we now turn to a study of digital search trees, we find that the formulas are similar, yet different enough that it is not easy to see how to deduce the asymptotic behavior. For example, if $\overline{C}_N$ denotes the average total number of bit inspections made when looking up all N keys in a binary digital search tree, it is not difficult to deduce as above that $\overline{C}_0 = \overline{C}_1 = 0$, and

$$\overline{C}_{N+1} = N + M^{1-N} \sum_k \binom{N}{k} (M-1)^{N-k} \overline{C}_k \qquad \text{for} \qquad N \geq 0. \qquad (6)$$

This is almost identical to Eq. (5); but the appearance of $N + 1$ instead of N on the left-hand side of this equation is enough to change the entire character of the recurrence, and so the methods we have used to study (5) are wiped out.

Let's consider the binary case first. Figure 35 shows the digital search tree corresponding to the sixteen example keys of Fig. 34, when they have been inserted in the order used in the examples of Chapter 5. If we want to determine the average number of bit inspections made in a random successful search, this is just the internal path length of the tree divided by N, since we need l bit inspections to find a node on level l. Note, however, that the average number of bit inspections made in a random *unsuccessful* search is *not* simply related to the external path length of the tree, since unsuccessful searches are more likely to occur at external nodes near the root; thus, the probability of reaching the left sub-branch of node *0075* in Fig. 35 is $\frac{1}{8}$ (assuming infinitely precise keys), and the left sub-branch of node *0232* will be encountered with probability only $\frac{1}{32}$. (For this reason, digital search trees tend to stay better balanced than the binary search trees of Algorithm 6.2.2T, when the keys are uniformly distributed.)

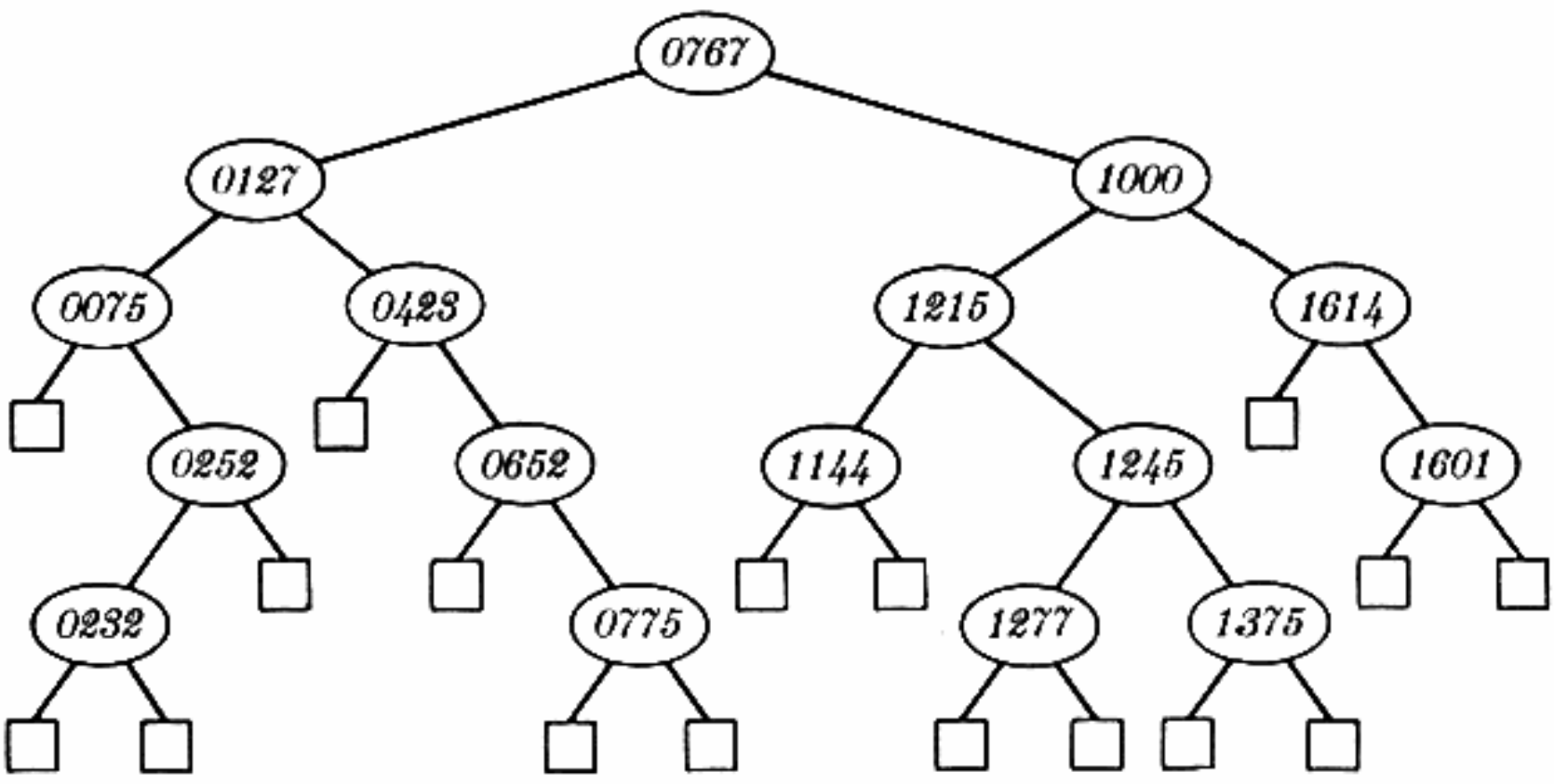


Fig. 35. A random digital search tree constructed by Algorithm D.

We can use a generating function to describe the pertinent characteristics of a digital search tree. If there are a_l internal nodes on level l, consider the generating function $a(z) = \sum a_l z^l$; for example, the generating function corresponding to Fig. 35 is $a(z) = 1 + 2z + 4z^2 + 5z^3 + 4z^4$. If there are b_l external nodes on level l, and if $b(z) = \sum b_l z^l$, we have

$$b(z) = 1 + (2z - 1)a(z) \tag{7}$$

by exercise 6.2.1–25. For example, $1 + (2z - 1)(1 + 2z + 4z^2 + 5z^3 + 4z^4) = 3z^3 + 6z^4 + 8z^5$. The average number of bit inspections made in a random successful search is $a'(1)/a(1)$, since $a'(1)$ is the internal path length of the tree and $a(1)$ is the number of internal nodes. The average number of bit inspections

made in a random *unsuccessful* search is $\sum l b_l 2^{-l} = \frac{1}{2}b'(\frac{1}{2}) = a(\frac{1}{2})$, since we end up at a given external node on level l with probability 2^{-l}. The number of comparisons is the same as the number of bit inspections, plus one in a successful search. For example, in Fig. 35, a successful search will take $2\frac{9}{16}$ bit inspections and $3\frac{9}{16}$ comparisons, on the average; an unsuccessful search will take $3\frac{7}{8}$ of each.

Now let $g_N(z)$ be the "average" $a(z)$ for trees with N nodes; in other words, $g_N(z)$ is the sum $\sum p_T a_T(z)$ over all binary digital search trees T with N internal nodes, where $a_T(z)$ is the generating function for the internal nodes of T and p_T is the probability that T occurs when N random numbers are inserted using Algorithm D. Then the average number of bit inspections will be $g'_N(1)/N$ in a successful search, $g_N(\frac{1}{2})$ in an unsuccessful search.

We can compute $g_N(z)$ by mimicking the tree construction process, as follows. If $a(z)$ is the generating function for a tree of N nodes, we can form $N+1$ trees from it by making the next insertion into any one of the external node positions. The insertion goes into a given external node on level l with probability 2^{-l}; hence the sum of the generating functions for the $N+1$ new trees, multiplied by the probability of occurrence, is $a(z) + b(\frac{1}{2}z) = a(z) + 1 + (z-1)a(\frac{1}{2}z)$. Averaging over all trees for N nodes, it follows that

$$g_{N+1}(z) = g_N(z) + 1 + (z-1)g_N(\tfrac{1}{2}z); \qquad g_0(z) = 0. \tag{8}$$

The corresponding generating function for external nodes, $h_N(z) = 1 + (2z-1)g_N(z)$, is somewhat easier to work with, because (8) is equivalent to the formula

$$h_{N+1}(z) = h_N(z) + (2z-1)h_N(\tfrac{1}{2}z); \qquad h_0(z) = 1. \tag{9}$$

Applying this rule repeatedly, we find that

$$\begin{aligned} h_{N+1}(z) &= h_{N-1}(z) + 2(2z-1)h_{N-1}(\tfrac{1}{2}z) + (2z-1)(z-1)h_{N-1}(\tfrac{1}{4}z) \\ &= h_{N-2}(z) + 3(2z-1)h_{N-2}(\tfrac{1}{2}z) + 3(2z-1)(z-1)h_{N-2}(\tfrac{1}{4}z) \\ &\quad + (2z-1)(z-1)(\tfrac{1}{2}z-1)h_{N-2}(\tfrac{1}{8}z) \end{aligned}$$

and so on, so that eventually we have

$$h_N(z) = \sum_k \binom{N}{k} \prod_{0\le j<k} (2^{1-j}z - 1); \tag{10}$$

$$g_N(z) = \sum_{k\ge 0} \binom{N}{k+1} \prod_{0\le j<k} (2^{-j}z - 1). \tag{11}$$

For example, $g_4(z) = 4 + 6(z-1) + 4(z-1)(\frac{1}{2}z-1) + (z-1)(\frac{1}{2}z-1)(\frac{1}{4}z-1)$. These formulas make it possible to express the quantities we are looking for as

sums of products:

$$\overline{C}_N = g'_N(1) = \sum_{k\ge 0} \binom{N}{k+2} \prod_{1\le j\le k} (2^{-j} - 1); \tag{12}$$

$$g_N(\tfrac{1}{2}) = \sum_{k\ge 0} \binom{N}{k+1} \prod_{1\le j\le k} (2^{-j} - 1) = \overline{C}_{N+1} - \overline{C}_N. \tag{13}$$

It is not at all obvious that this formula for $\overline{C}_N$ satisfies (6)!

Unfortunately, these expressions are not suitable for calculation or for finding an asymptotic expansion, since $2^{-j} - 1$ is negative; we get large terms and a lot of cancellation. A more useful formula for $\overline{C}_N$ can be obtained by applying the partition identities of exercise 5.1.1–16. We have

$$\begin{aligned}
\overline{C}_N &= \Big(\prod_{j\ge 1} (1 - 2^{-j})\Big) \sum_{k\ge 0} \binom{N}{k+2} (-1)^k \prod_{l\ge 0} (1 - 2^{-l-k-1})^{-1} \\
&= \Big(\prod_{j\ge 1} (1 - 2^{-j})\Big) \sum_{k\ge 0} \binom{N}{k+2} (-1)^k \sum_{m\ge 0} (2^{-k-1})^m \prod_{1\le r\le m} (1 - 2^{-r})^{-1} \\
&= \sum_{m\ge 0} 2^m \Big(\sum_k \binom{N}{k} (-2^{-m})^k - 1 + 2^{-m}N\Big) \prod_{j\ge 0} (1 - 2^{-j-m-1}) \\
&= \sum_{m\ge 0} 2^m ((1 - 2^{-m})^N - 1 + 2^{-m}N) \sum_{n\ge 0} (-2^{-m-1})^n 2^{-n(n-1)/2} \\
&\qquad\qquad \times \prod_{1\le r\le n} (1 - 2^{-r})^{-1}.
\end{aligned} \tag{14}$$

This may not seem at first glance to be an improvement over Eq. (12), but it has the great advantage that it converges rapidly for each fixed n. A precisely analogous situation occurred for the trie case in Eq. 5.2.2–38, 39; in fact, if we consider only the $n = 0$ terms of (14), we have exactly $N - 1$ plus the number of bit inspections in a binary trie. We can now proceed to get the asymptotic value in essentially the same way as before; see exercise 27. [The above derivation is largely based on an approach suggested by A. J. Konheim and D. J. Newman, *Discrete Mathematics* (to appear).]

Finally let us take a mathematical look at Patricia. In her case the binary tree is like the corresponding binary trie on the same keys, but squashed together (because the SKIP fields eliminate 1-way branching), so that there are $N - 1$ internal nodes and N external nodes. Figure 36 shows the Patrician tree corresponding to the sixteen keys in the trie of Fig. 34. The number shown in each branch node is the amount of SKIP; the keys are indicated with the external nodes, although the external node is not explicitly present (there is actually a tagged link to an internal node which references the TEXT, in place

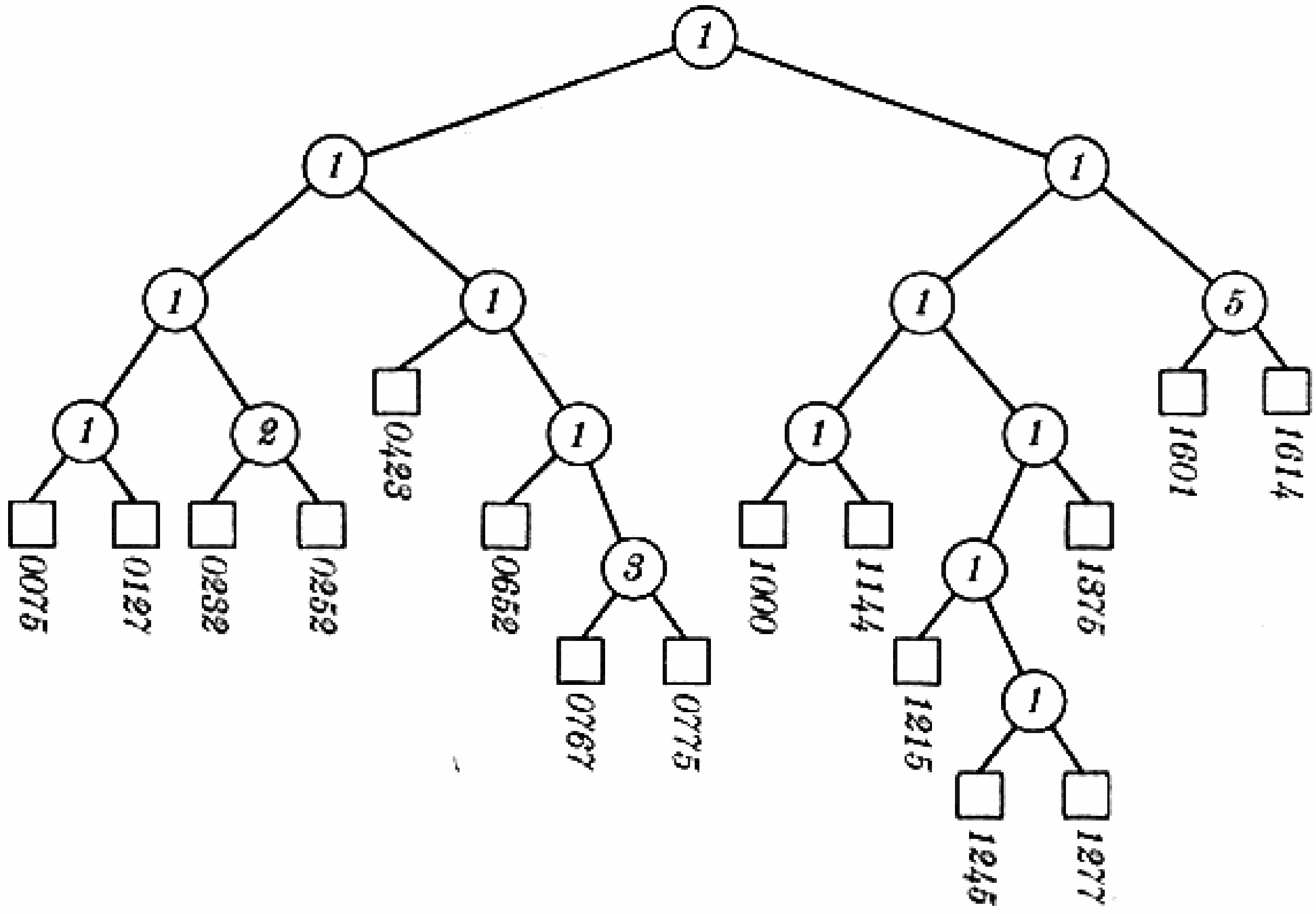


Fig. 36. Patricia constructs this tree instead of Fig. 34.

of each external node.) For the purposes of analysis, we may assume that external nodes exist as shown.

Since successful searches with Patricia end at external nodes, the average number of bit inspections made in a random successful search will be the external path length, divided by N. If we form the generating function $b(z)$ for external nodes as above, this will be $b'(1)/b(1)$. An *unsuccessful* search with Patricia also ends at an external node, but weighted with probability 2^{-l} for external nodes on level l, so the average number of bit inspections is $\frac{1}{2}b'(\frac{1}{2})$. For example, in Fig. 36 we have $b(z) = 3z^3 + 8z^4 + 3z^5 + 2z^6$; there are $4\frac{1}{4}$ bit inspections per successful search and $3\frac{25}{32}$ per unsuccessful search, on the average.

Let $h_N(z)$ be the "average" $b(z)$ for a Patrician tree constructed with N external nodes, using uniformly distributed keys. The recurrence relation

$$h_n(z) = 2^{1-n} \sum_k \binom{n}{k} h_k(z)\bigl(z + \delta_{kn}(1 - z)\bigr), \quad h_0(z) = 0, \quad h_1(z) = 1 \tag{15}$$

appears to have no simple solution. But fortunately, there is a simple recurrence for the average external path length $h_n'(1)$, since

$$\begin{aligned} h_n'(1) &= 2^{1-n} \sum_k \binom{n}{k} h_k'(1) + 2^{1-n} \sum_k \binom{n}{k} k(1 - \delta_{kn}) \\ &= n - 2^{n-1}n + 2^{1-n} \sum_k \binom{n}{k} h_k'(1). \end{aligned} \tag{16}$$

Since this has the form of (6), we can use the methods already developed to solve for $h_n'(1)$, which turns out to be exactly n less than the corresponding number of bit inspections in a random binary trie. Thus, the SKIP fields save us

about one bit inspection per successful search, on random data. (See exercise 31.) The redundancy of typical real data will lead to greater savings.

When we try to find the average number of bit inspections for a random *un*successful search by Patricia, we obtain the recurrence

$$a_n = 1 + \frac{1}{2^n - 2} \sum_{k<n} \binom{n}{k} a_k, \qquad \text{for} \qquad n \geq 2; \qquad a_0 = a_1 = 0. \tag{17}$$

Here $a_n = \frac{1}{2}h_n'(\frac{1}{2})$. This does *not* have the form of any recurrences we have studied, nor is it easily transformed into such a recurrence. In fact, it turns out that the solution involves the Bernoulli numbers:

$$\frac{na_{n-1}}{2} - n + 2 = \sum_{2\leq k<n} \binom{n}{k} \frac{B_k}{2^{k-1} - 1}. \tag{18}$$

This formula is probably the hardest asymptotic nut we have yet had to crack; the solution in exercise 34 is an instructive review of many things we have done before, with some slightly different twists.

Summary of the analyses. As a result of all the complicated mathematics in this section, the following facts are perhaps the most noteworthy:

(a) The number of nodes needed to store N random keys in an M-ary trie, with the trie branching terminated for subfiles of $\leq s$ keys, is approximately $N/(s \ln M)$. (This approximation is valid for large N, small s, and small M.) Since a trie node involves M link fields, we will need only about $N/\ln M$ link fields if we choose $s = M$.

(b) The number of digits or characters examined during a random search is approximately $\log_M N$ for all methods considered. When $M = 2$, the various analyses give us the following more accurate approximations to the number of bit inspections:

	Successful	Unsuccessful
Trie search	$\log_2 N + 1.33275$	$\log_2 N - 0.10995$
Digital tree search	$\log_2 N - 1.71665$	$\log_2 N - 0.27395$
Patricia	$\log_2 N + 0.33275$	$\log_2 N - 0.31875$

(These approximations can all be expressed in terms of fundamental mathematical constants, e.g. 0.31875 stands for $(\ln \pi - \gamma)/(\ln 2) - 1/2$.)

(c) "Random" data here means that the M-ary digits are uniformly distributed, as if the keys were real numbers between 0 and 1 expressed in M-ary notation. Digital search methods are insensitive to the order in which keys are entered into the file (except for Algorithm D, which is only slightly sensitive to the order); but they are very sensitive to the distribution of digits. For example, if 0 bits are much more common than 1 bits, the trees will become much more skewed than they would be for random data as considered in the analyses cited above. (Exercise 5.2.2–53 works out one example of what happens when the data is biased in this way.)

EXERCISES

1. [*00*] If a tree has leaves, what does a trie have?

2. [*20*] Design an algorithm for the insertion of a new key into an M-ary trie, using the conventions of Algorithm T.

3. [*21*] Design an algorithm for the deletion of a key from an M-ary trie, using the conventions of Algorithm T.

▶ 4. [*21*] Most of the 360 entries in Table 1 are blank (null links). But we can compress the table into only 55 entries, by overlapping nonblank entries with blank ones, as follows:

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Entry	A	---	THAT	(26)	---	WAS	THE	(15)	(2)	I	THIS	HIS	WHICH	WITH	HE	AND	TO	OF	BE	ARE	(1)	(14)	AS	AT	IN	(21)	ON	(3)

Position	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55
Entry	(10)	HAD	OR	IS	IT	HER	NOT	(12)	FOR	BUT	---	FROM	---	BY	(1)	---	---	(5)	---	YOU	---	---	HAVE	---	---	---	---

(Nodes (1), (2), . . . , (12) of Table 1 begin, respectively, at positions 20, 1, 14, 21, 3, 10, 12, 1, 5, 26, 15, 2 within this compressed table.)

Show that if the compressed table is substituted for Table 1, Program T will still work, but not quite as fast.

▶ 5. [*M26*] (Y. N. Patt.) The trees of Fig. 31 have their letters arranged in alphabetic order within each family. This order is not necessary, and if we rearrange the order of nodes within the families before constructing binary tree representations such as (2) we may get a faster search. What rearrangement of Fig. 31 is optimum from this standpoint? (Use the frequency assumptions of Fig. 32, and find the forest which minimizes the successful search time when it has been represented as a binary tree.)

6. [*15*] What digital search tree is obtained if the fifteen 4-bit binary keys 0001, 0010, 0011, . . . , 1111 are inserted in increasing order by Algorithm D? (Start with 0001 at the root and then do fourteen insertions.)

▶ 7. [*M26*] If the fifteen keys of exercise 6 are inserted in a different order, we may get a different tree. Of all the 15! possible permutations of these keys, which is the *worst*, in the sense that it produces a tree with the greatest internal path length?

8. [*20*] Consider the following changes to Algorithm D, which have the effect of eliminating variable $K1$: Change "$K1$" to "K" in both places in step D2, and delete the operation "$K1 \leftarrow K$" from step D1. Will the resulting algorithm still be valid for searching and insertion?

9. [*21*] Write a MIX program for Algorithm D, and compare it to Program 6.2.2T. You may use binary operations such as SLB (shift left AX binary), JAE (jump if A even), etc.; and you may also use the idea of exercise 8 if it helps.

10. [*23*] Given a file in which all the keys are n-bit binary numbers, and given a search argument $K = b_1 b_2 \ldots b_n$, suppose we want to find the maximum value of k such that there is a key in the file beginning with the bit pattern $b_1 b_2 \ldots b_k$. How can we

do this efficiently if the file is represented as (a) a binary search tree (cf. Algorithm 6.2.2T); (b) a binary trie (cf. Algorithm T); (c) a binary digital search tree (cf. Algorithm D)?

11. [*21*] Can Algorithm 6.2.2D be used without change to delete a node from a digital search tree?

12. [*25*] After a random element is deleted from a random digital search tree constructed by Algorithm D, is the resulting tree still random? (Cf. exercise 11 and Theorem 6.2.2H.)

13. [*20*] (*M-ary digital searching.*) Explain how Algorithms T and D can be combined into a generalized algorithm that is essentially the same as Algorithm D when $M = 2$. What changes would be made to Table 1, if your algorithm is used for $M = 30$?

▶**14.** [*25*] Design an efficient algorithm that can be performed just after Algorithm P has terminated successfully, to locate *all* places where K appears in the TEXT.

15. [*28*] Design an efficient algorithm that can be used to construct the tree used by Patricia, or to insert new TEXT references into an existing tree. Your insertion algorithm should refer to the TEXT array at most twice.

16. [*22*] Why is it desirable for Patricia to make the restriction that no key is a prefix of another?

17. [*M25*] Find a way to express the solution of the recurrence

$$x_0 = x_1 = 0; \qquad x_n = a_n + m^{1-n} \sum_k \binom{n}{k} (m-1)^{n-k} x_k, \qquad n \geq 2,$$

in terms of binomial transforms, by generalizing the technique of exercise 5.2.2–36.

18. [*M21*] Use the result of exercise 17 to express the solutions to (4) and (5) in terms of functions U_n and V_n analogous to those defined in exercise 5.2.2–38.

19. [*HM23*] Find the asymptotic value of the function

$$K(n, s, m) = \sum_{k \geq 2} \binom{n}{k} \binom{k}{s} \frac{(-1)^k}{m^{k-1} - 1}$$

to $O(1)$ as $n \to \infty$, for fixed $s \geq 0$ and $m > 1$. [The case $s = 0$ has already been solved in exercise 5.2.2–50, and the case $s = 1$, $m = 2$ has been solved in exercise 5.2.2–48.]

▶**20.** [*M30*] Consider M-ary trie memory in which we use a sequential search whenever reaching a subfile of s or less keys. (Algorithm T is the special case $s = 1$.) Apply the results of the preceding exercises to analyze (a) the average number of trie nodes; (b) the average number of digit or character inspections in a successful search; and (c) the average number of comparisons made in a successful search. State your answers as asymptotic formulas as $N \to \infty$, for fixed M and s; the answer for (a) should be correct to within $O(1)$, and the answers for (b) and (c) should be correct to within $O(N^{-1})$. [When $M = 2$, this analysis applies also to the modified radix-exchange sort, in which subfiles of size $\leq s$ are sorted by insertion.]

21. [*M25*] How many of the nodes, in a random M-ary trie containing N keys, have a null pointer in table entry 0? (For example, 9 of the 12 nodes in Table 1 have a null

pointer in the "␣" position. "Random" in this exercise means as usual that the characters of the keys are uniformly distributed between 0 and $M - 1$.)

22. [*M25*] How many trie nodes are on level l of a random M-ary trie containing N keys, for $l = 0, 1, 2, \ldots$?

23. [*M26*] How many digit inspections are made on the average during an *un*successful search in an M-ary trie containing N random keys?

24. [*M30*] Consider an M-ary trie which has been represented as a forest (cf. Fig. 31). Find exact and asymptotic expressions for (a) the average number of nodes in the forest, and (b) the average number of times "P ← RLINK(P)" is performed during a random successful search.

▸**25.** [*M24*] The mathematical derivations of asymptotic values in this section have been quite difficult, involving complex variable theory, because it is desirable to get more than just the leading term of the asymptotic behavior (and the second term is intrinsically complicated). The purpose of this exercise is to show that elementary methods are good enough to deduce some of the results in weaker form. (a) Prove by induction that the solution to (4) satisfies $A_N \leq M(N-1)/(M-1)$ for $N \geq 1$. (b) Let $D_N = C_N - NH_{N-1}/(\ln M)$, where C_N is defined by (5). Prove that $D_N = O(N)$; hence $C_N = N \log_M N + O(N)$. [*Hint:* Use (a) and Theorem 1.2.7A.]

26. [*23*] Determine the value of the infinite product

$$(1 - \tfrac{1}{2})(1 - \tfrac{1}{4})(1 - \tfrac{1}{8})(1 - \tfrac{1}{16}) \ldots$$

correct to five decimal places, by hand calculation. [*Hint:* Cf. exercise 5.1.1–16.]

27. [*HM31*] What is the asymptotic value of $\overline{C}_N$, as given by (14), to within $O(1)$?

28. [*HM26*] Find the asymptotic average number of digit inspections when searching in a random M-ary digital search tree, for general $M \geq 2$. Consider both successful and unsuccessful search, and give your answer to within $O(N^{-1})$.

29. [*M46*] What is the asymptotic average number of nodes, in an M-ary digital search tree, for which all M links are null? (We might save memory space by eliminating such nodes, cf. exercise 13.)

30. [*M24*] Show that the Patrician generating function $h_n(z)$ defined in (15) can be expressed in the rather horrible form

$$n \sum_{m \geq 1} z^m \left(\sum_{\substack{a_1 + \cdots + a_m = n-1 \\ a_1, \ldots, a_m \geq 1}} \binom{n-1}{a_1, \ldots, a_m} \frac{1}{(2^{a_1} - 1)(2^{a_1 + a_2} - 1) \ldots (2^{a_1 + \cdots + a_m} - 1)} \right).$$

[Thus, if there is a simple formula for $h_n(z)$, we will be able to simplify this rather ungainly expression.]

31. [*M21*] Solve the recurrence (16).

32. [*M21*] What is the average value of the sum of all SKIP fields in a random Patrician tree with $N - 1$ internal nodes?

33. [*M30*] Prove that (18) is a solution to the recurrence (17). [*Hint:* Consider the generating function $A(z) = \sum_{n \geq 0} a_n z^n / n!$.]

34. [*HM40*] The purpose of this exercise is to find the asymptotic behavior of (18).

(a) Prove that

$$\frac{1}{n}\sum_{2\le k<n}\binom{n}{k}\frac{B_k}{2^{k-1}-1}=\sum_{j\ge 1}\left(\frac{1^{n-1}+2^{n-1}+\cdots+(2^j-1)^{n-1}}{2^{j(n-1)}}-\frac{2^j}{n}+\frac{1}{2}\right).$$

(b) Show that the summand in (a) can be approximated by $1/(e^x-1)-1/x+1/2$, where $x=n/2^j$; the resulting sum equals the original sum $+\,O(n^{-1})$. (c) Show that

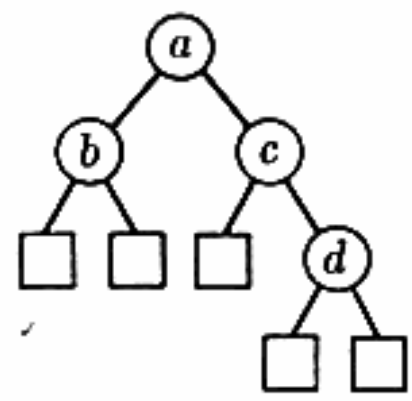
$$\frac{1}{e^x-1}-\frac{1}{x}+\frac{1}{2}=\frac{1}{2\pi i}\int_{-\frac{1}{2}-i\infty}^{-\frac{1}{2}+i\infty}\zeta(z)\Gamma(z)x^{-z}\,dz,\qquad\text{for real } x>0.$$

(d) Therefore the sum equals

$$\frac{1}{2\pi i}\int_{-\frac{1}{2}-i\infty}^{-\frac{1}{2}+i\infty}\frac{\zeta(z)\Gamma(z)n^{-z}}{2^{-z}-1}\,dz+O(n^{-1});$$

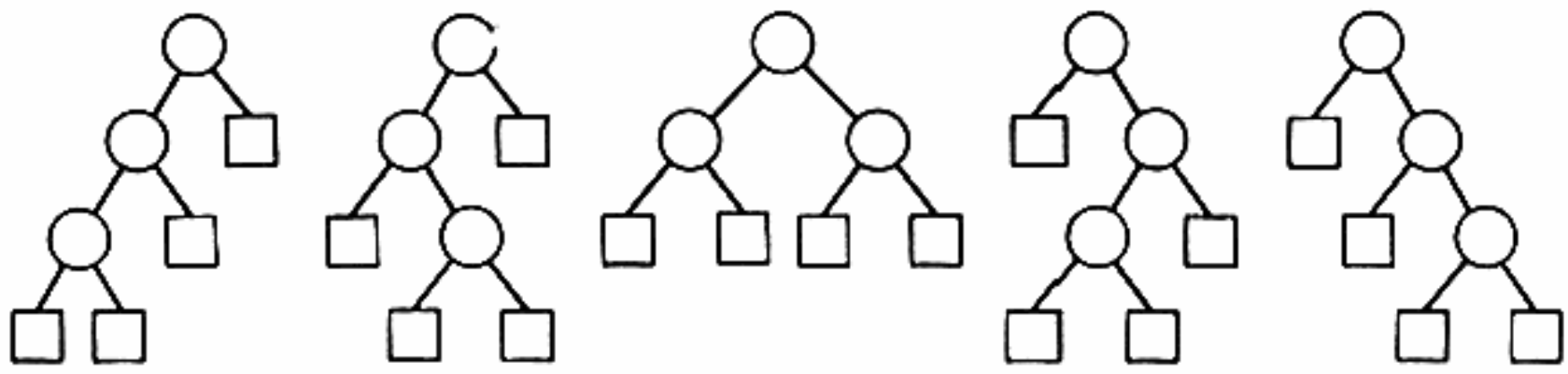
evaluate this integral.

▸**35.** [*M20*] What is the probability that Patricia's tree on five keys will be



with the SKIP fields a, b, c, d as shown? (Assume that the keys have independent random bits, and give your answer as a function of a, b, c and d.)

36. [*M25*] There are five binary trees with three internal nodes. If we consider how frequently each particular one of these occurs as the search tree in various algorithms, for random data, we find the following different probabilities:



Tree search (Algorithm 6.2.2T)	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{6}$
Digital tree search (Algorithm D)	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{2}$	$\frac{1}{8}$	$\frac{1}{8}$
Patricia (Algorithm P)	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{3}{7}$	$\frac{1}{7}$	$\frac{1}{7}$

(Note that the digital search tree tends to be balanced more often than the others.) In exercise 6.2.2–5 we found that the probability of a tree in the tree search algorithm was $\Pi(1/s(x))$, where the product is over all internal nodes x, and $s(x)$ is the number of internal nodes in the subtree rooted at x. Find similar formulas for the probability of a tree in the case of (a) Algorithm D; (b) Algorithm P.

▸**37.** [*M22*] Consider a binary tree with b_l external nodes on level l. The text observes that the running time for unsuccessful searching in digital search trees is not directly related to the external path length $\sum lb_l$, but instead it is essentially proportional to the *modified external path length* $\sum lb_l 2^{-l}$. Prove or disprove: The smallest modified external path length, over all trees with N external nodes, occurs when all of the external nodes appear on at most two adjacent levels. [Cf. exercise 5.3.1–20.]

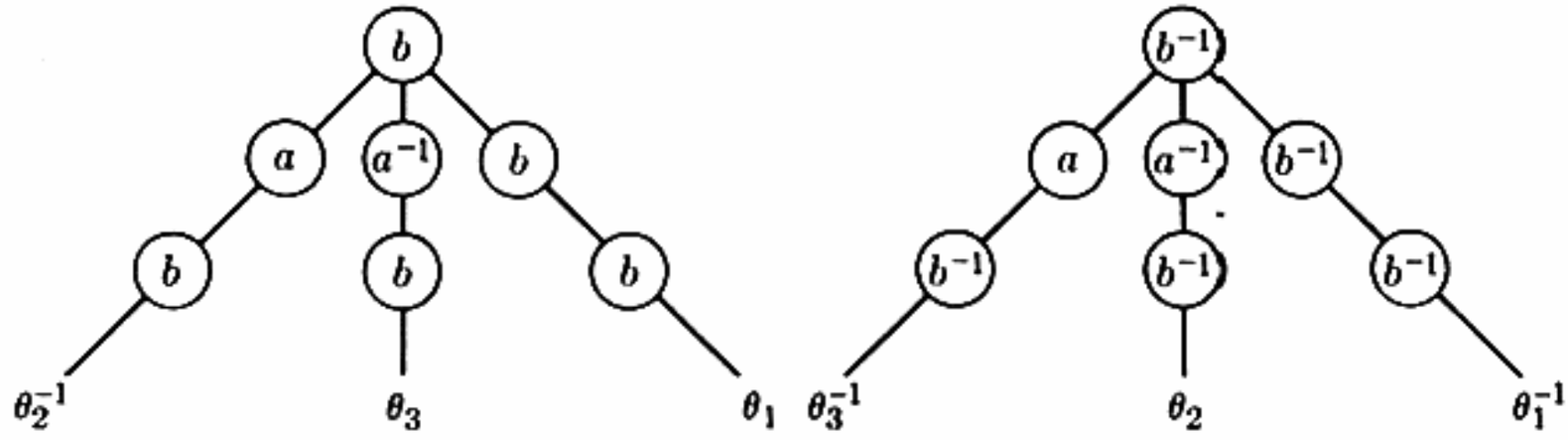
38. [*M40*] Develop an algorithm to find the n-node tree having the minimum value of $\alpha \cdot$ (internal path length) $+ \beta \cdot$ (modified external path length), given α and β. (Cf. exercise 37.)

39. [*M47*] Develop an algorithm to find optimum digital search trees, analogous to the optimum binary search trees considered in Section 6.2.2.

40. [*25*] Let $a_0a_1a_2\ldots$ be a periodic binary sequence with $a_{N+k} = a_k$ for all $k \geq 0$. Show that there is a way to represent any fixed sequence of this type in $O(N)$ memory locations, so that the following operation can be done in only $O(n)$ steps: Given any binary pattern $b_0b_1\ldots b_{n-1}$, determine how often the pattern occurs in the period (i.e., find how many values of p exist with $0 \leq p < N$ and $b_k = a_{p+k}$ for $0 \leq k < n$). (The length n of the pattern is variable as well as the pattern itself. Assume that each memory location is big enough to hold arbitrary integers between 0 and N).

41. [*HM28*] This is an application to group theory. Let G be the free group on the letters $\{a_1, \ldots, a_n\}$, i.e., the set of all strings $\alpha = b_1\ldots b_r$, where each b_i is one of the a_j or a_j^{-1} and no adjacent pair $a_ja_j^{-1}$ or $a_j^{-1}a_j$ occurs. The inverse of α is $b_r^{-1}\ldots b_1^{-1}$, and we multiply two such strings by concatenating them and cancelling adjacent inverse pairs. Let H be the subgroup of G generated by the strings $\{\beta_1, \ldots, \beta_p\}$, i.e., the set of all elements of G which can be written as products of the β's and their inverses. It can be shown (see Marshall Hall, *The Theory of Groups* (New York: Macmillan, 1959), Chapter 7) that we may always find generators $\theta_1, \ldots, \theta_m$ of H, with $m \leq p$, satisfying the "Nielsen property," which states that the middle character of θ_i (or at least one of the two central characters of θ_i if it has even length) is never cancelled in the expressions $\theta_i\theta_j^e$ or $\theta_j^e\theta_i$, $e = \pm 1$, unless $j = i$ and $e = -1$. This property implies that there is a simple algorithm for testing whether an arbitrary element of G is in H: Record the $2m$ keys $\theta_1, \ldots, \theta_m, \theta_1^{-1}, \ldots, \theta_m^{-1}$ in a character-oriented search tree, using the $2n$ letters $a_1, \ldots, a_n, a_1^{-1}, \ldots, a_n^{-1}$. Let $\alpha = b_1\ldots b_r$ be a given element of G; if $r = 0$, α is obviously in H. Otherwise look up α, finding the longest prefix $b_1\ldots b_k$ that matches a key. If there is more than one key beginning with $b_1\ldots b_k$, α is not in H; otherwise let the unique such key be $b_1\ldots b_kc_1\ldots c_l = \theta_i^e$, and replace α by $\theta_i^{-e}\alpha = c_l^{-1}\ldots c_1^{-1}b_{k+1}\ldots b_r$. If this new value of α is longer than the old (i.e., if $l > k$), α is not in H; otherwise repeat the process on the new value of α. The Nielsen property implies that this algorithm will always terminate. If α is eventually reduced to the null string, we can reconstruct the representation of the original α as a product of θ's.

For example, let $\{\theta_1, \theta_2, \theta_3\} = \{bbb, b^{-1}a^{-1}b^{-1}, ba^{-1}b\}$ and $\alpha = bbabaab$. The forest



can be used with the above algorithm to deduce that $\alpha = \theta_1\theta_3^{-1}\theta_1\theta_3^{-1}\theta_2^{-1}$. Implement this algorithm, given the θ's as input to your program.

42. [*23*] (*Front and rear compression.*) When a set of binary keys is being used as an index, to partition a larger file, we need not store the full keys. For example, if the sixteen keys of Fig. 34 are used, they can be truncated at the right, as soon as enough digits have been given to uniquely identify them: 0000, 0001, 00100, 00101, 010, . . . , 1110001. These truncated keys can be used to partition a file into seventeen parts, where for example the fifth part consists of all keys beginning with 0011 or 010, and the last part contains all keys beginning with 111001, 11101, or 1111. The truncated keys can be represented more compactly if we suppress all leading digits common to the previous key: 0000, ***1, **100, ****1, *10, . . . , ******1. The bit following a * is always 1, so it may be suppressed. A large file will have many *'s, and we need store only the number of *'s and the values of the following bits. (This compression technique was shown to the author by A. Heller and R. L. Johnsen.)

Show that the total number of bits in the compressed file, excluding *'s and the following 1 bits, is always equal to the number of nodes in the binary trie for the keys.

(Consequently the average total number of such bits in the entire index is about $N/(\ln 2)$, only 1.44 bits per key. Still further compression is possible, since we need only represent the trie structure; cf. Theorem 2.3.1 A.)

So far we have considered search methods based on comparing the given argument K to the keys in the table, or using its digits to govern a branching process. A third possibility is to avoid all this rummaging around by doing some arithmetical calculation on K, computing a function $f(K)$ which is the location of K and the associated data in the table.

For example, let's consider again the set of 31 English words which we have subjected to various search strategies in Section 6.2.2 and 6.3. Table 1 shows a short MIX program which transforms each of the 31 keys into a unique number $f(K)$ between -10 and 30. If we compare this method to the MIX programs for the other methods we have considered (e.g., binary search, optimal tree search, trie memory, digital tree search), we find that it is superior from the standpoint of both space and speed, except that binary search uses slightly less space. In fact, the average time for a successful search, using the program of Table 1 with the frequency data of Fig. 12, is only about $17.8u$, and only 41 table locations are needed to store the 31 keys.

Unfortunately it isn't very easy to discover such functions $f(K)$. There are $41^{31} \approx 10^{50}$ possible functions from a 31-element set into a 41-element set, and only $41 \cdot 40 \cdot \cdots \cdot 11 = 41!/10! \approx 10^{43}$ of them will give distinct values for each argument; thus only about one of every 10 million functions will be suitable.

Functions which avoid duplicate values are surprisingly rare, even with a fairly large table. For example, the famous "birthday paradox" asserts that if 23 or more people are present in a room, chances are good that two of them

Table 1

TRANSFORMING A SET OF KEYS INTO UNIQUE ADDRESSES

	Instruction		A	AND	ARE	AS	AT	BE	BUT	BY	FOR	FROM	HAD	HAVE	HE	HER
	LD1N	K(1:1)	−1	−1	−1	−1	−1	−2	−2	−2	−6	−6	−8	−8	−8	−8
	LD2	K(2:2)	−1	−1	−1	−1	−1	−2	−2	−2	−6	−6	−8	−8	−8	−8
	INC1	−8,2	−9	6	10	13	14	−5	14	18	2	5	−15	−15	−11	−11
	J1P	*+2	−9	6	10	13	14	−5	14	18	2	5	−15	−15	−11	−11
	INC1	16,2	7	.	.	.	.	16	.	.	.	.	2	2	10	10
	LD2	K(3:3)	7	6	10	13	14	16	14	18	2	5	2	2	10	10
	J2Z	9F	7	6	10	13	14	16	14	18	2	5	2	2	10	10
	INC1	−28,2	.	−18	−13	.	.	.	9	.	−7	−7	−22	−1	.	1
	J1P	9F	.	−18	−13	.	.	.	9	.	−7	−7	−22	−1	.	1
	INC1	11,2	.	−3	3	.	.	.	.	.	23	20	−7	35	.	.
	LDA	K(4:4)	.	−3	3	.	.	.	.	.	23	20	−7	35	.	.
	JAZ	9F	.	−3	3	.	.	.	.	.	23	20	−7	35	.	.
	DEC1	−5,2	.	.	.	.	.	.	.	.	.	9	.	15	.	.
	J1N	9F	.	.	.	.	.	.	.	.	.	9	.	15	.	.
	INC1	10	.	.	.	.	.	.	.	.	.	19	.	25	.	.
9H	LDA	K	7	−3	3	13	14	16	9	18	23	19	−7	25	10	1
	CMPA	TABLE,1	7	−3	3	13	14	16	9	18	23	19	−7	25	10	1
	JNE	FAILURE	7	−3	3	13	14	16	9	18	23	19	−7	25	10	1

will have the same month and day of birth! In other words, if we select a random function which maps 23 keys into a table of size 365, the probability that no two keys map into the same location is only 0.4927 (less than one-half). Skeptics who doubt this result should try to find the birthday mates at the next large parties they attend. [The birthday paradox originated in work of R. von Mises, *İstanbul Üniversitesi Fen Fakültesi Mecmuasi* **4** (1939), 145–163, and W. Feller, *An Introduction to Probability Theory* (New York: Wiley, 1950), Section 2.3.]

On the other hand, the approach used in Table 1 is fairly flexible [cf. M. Greniewski and W. Turski, *CACM* **6** (1963), 322–323], and for a medium-sized table a suitable function can be found after about a day's work. In fact it is rather amusing to solve a puzzle like this.

Of course this method has a serious flaw, since the contents of the table must be known in advance; adding one more key will probably ruin everything, making it necessary to start over almost from scratch. We can obtain a much more versatile method if we give up the idea of uniqueness, permitting different keys to yield the same value $f(K)$, and using a special method to resolve any ambiguity after $f(K)$ has been computed.

These considerations lead to a popular class of search methods commonly known as *hashing* or *scatter storage* techniques. The verb "to hash" means to chop something up or to make a mess out of it; the idea in hashing is to chop off some aspects of the key and to use this partial information as the basis for searching. We compute a *hash function* $h(K)$ and use this value as the address where the search begins.

The birthday paradox tells us that there will probably be distinct keys $K_i \neq K_j$ which hash to the same value $h(K_i) = h(K_j)$. Such an occurrence is

HIS	I	IN	IS	IT	NOT	OF	ON	OR	THAT	THE	THIS	TO	WAS	WHICH	WITH	YOU
Contents of r11 after executing the instruction, given a particular key K																
−8	−9	−9	−9	−9	−15	−16	−16	−16	−23	−23	−23	−23	−26	−26	−26	−28
−8	−9	−9	−9	−9	−15	−16	−16	−16	−23	−23	−23	−23	−26	−26	−26	−28
−7	−17	−2	5	6	−7	−18	−9	−5	−23	−23	−23	−15	−33	−26	−25	−20
−7	−17	−2	5	6	−7	−18	−9	−5	−23	−23	−23	−15	−33	−26	−25	−20
18	−1	29	.	.	25	4	22	30	1	1	1	17	−16	−2	0	12
18	−1	29	5	6	25	4	22	30	1	1	1	17	−16	−2	0	12
18	−1	29	5	6	25	4	22	30	1	1	1	17	−16	−2	0	12
12	.	.	.	.	20	.	.	.	−26	−22	−18	.	−22	−21	−5	8
12	.	.	.	.	20	.	.	.	−26	−22	−18	.	−22	−21	−5	8
.	.	.	.	.	.	.	.	.	−14	−6	2	.	11	−1	29	.
.	.	.	.	.	.	.	.	.	−14	−6	2	.	11	−1	29	.
.	.	.	.	.	.	.	.	.	−14	−6	2	.	11	−1	29	.
.	.	.	.	.	.	.	.	.	−10	.	−2	.	.	−5	11	.
.	.	.	.	.	.	.	.	.	−10	.	−2	.	.	−5	11	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	21	.
12	−1	29	5	6	20	4	22	30	−10	−6	−2	17	11	−5	21	8
12	−1	29	5	6	20	4	22	30	−10	−6	−2	17	11	−5	21	8
12	−1	29	5	6	20	4	22	30	−10	−6	−2	17	11	−5	21	8

called a *collision*, and several interesting approaches have been devised to handle the collision problem. In order to use a scatter table, a programmer must make two almost independent decisions: He must choose a hash function $h(K)$, and he must select a method for collision resolution. We shall now consider these two aspects of the problem in turn.

Hash functions. To make things more explicit, let us assume throughout this section that our hash function h takes on at most M different values, with

$$0 \le h(K) < M, \tag{1}$$

for all keys K. The keys in actual files that arise in practice usually have a great deal of redundancy; we must be careful to find a hash function that breaks up clusters of almost identical keys, in order to reduce the number of collisions.

It is theoretically impossible to define a hash function that creates random data from the nonrandom data in actual files. But in practice it is not difficult to produce a pretty good imitation of random data, by using simple arithmetic as we have discussed in Chapter 3. And in fact we can often do even better, by exploiting the nonrandom properties of actual data to construct a hash function that leads to fewer collisions than truly random keys would produce.

Consider, for example, the case of 10-digit keys on a decimal computer. One hash function that suggests itself is to let $M = 1000$, say, and to let $h(K)$ be three digits chosen from somewhere near the middle of the 20-digit product $K \times K$. This would seem to yield a fairly good spread of values between 000 and 999, with low probability of collisions. Experiments with actual data show, in fact, that this "middle square" method isn't bad, provided that the keys do not have a lot of leading or trailing zeros; but it turns out that there are safer and saner ways to proceed, just as we found in Chapter 3 that the middle square method is not an especially good random number generator.

Extensive tests on typical files have shown that two major types of hash functions work quite well. One of these is based on division, and the other is based on multiplication.

The division method is particularly easy; we simply use the remainder modulo M:

$$h(K) = K \bmod M. \tag{2}$$

In this case, some values of M are obviously much better than others. For example, if M is an even number, $h(K)$ will be even when K is even and odd when K is odd, and this will lead to a substantial bias in many files. It would be even worse to let M be a power of the radix of the computer, since $K \bmod M$ would then be simply the least significant digits of K (independent of the other digits). Similarly we can argue that M probably shouldn't be a multiple of 3 either; for if the keys are alphabetic, two keys which differ from each other only by permutation of letters would then differ in numeric value by a multiple of 3. (This occurs because $10^n \bmod 3 = 4^n \bmod 3 = 1$.) In general, we want to avoid values of M which divide $r^k \pm a$, where k and a are small numbers and r is the radix of the alphabetic character set (usually $r = 64$, 256, or 100),

since a remainder modulo such a value of M tends to be largely a simple superposition of the key digits. Such considerations suggest that we *choose M to be a prime number* such that $r^k \not\equiv \pm a$ (modulo M) for small k and a. This choice has been found to be quite satisfactory in virtually all cases.

For example, on the MIX computer we could choose $M = 1009$, computing $h(K)$ by the sequence

```
LDX    K          rX ← K.
ENTA   0          rA ← 0.                    (3)
DIV    =1009=     rX ← K mod 1009.
```

The multiplicative hashing scheme is equally easy to do, but it is slightly harder to describe because we must imagine ourselves working with fractions instead of with integers. Let w be the word size of the computer, so that w is usually 10^{10} or 2^{30} for MIX; we can regard an integer A as the fraction A/w if we imagine the radix point to be at the left of the word. The method is to choose some integer constant A relatively prime to w, and to let

$$h(K) = \left\lfloor M \left(\left(\frac{A}{w} K \right) \bmod 1 \right) \right\rfloor. \tag{4}$$

In this case we usually let M be a power of 2 on a binary computer, so that $h(K)$ consists of the leading bits of the least significant half of the product AK.

In MIX code, if we let $M = 2^m$ and assume a binary radix, the multiplicative hash function is

```
LDA    K          rA ← K.
MUL    A          rAX ← AK.
ENTA   0          rAX ← AK mod w.            (5)
SLB    m          Shift rAX m bits to the left.
```

Now $h(K)$ appears in register A. Since MIX has rather slow multiplication and shift instructions, this sequence takes exactly as long to compute as (3); but on many machines multiplication is significantly faster than division.

In a sense this method can be regarded as a generalization of (3), since we could for example take A to be an approximation to $w/1009$; multiplying by the reciprocal of a constant is often faster than dividing by that constant. Note that (5) is almost a "middle square" method, but there is one important difference: We shall see that multiplication by a suitable constant has demonstrably good properties.

One of the nice features of the multiplicative scheme is that no information was lost in (5); we would determine K again, given only the contents of rAX after (5) has finished. The reason is that A is relatively prime to w, so Euclid's algorithm can be used to find a constant A' with $AA' \bmod w = 1$; this implies that $K = (A'(AK \bmod w)) \bmod w$. In other words, if $f(K)$ denotes the contents of register X just before the SLB instruction in (5), then

$$K_1 \neq K_2 \quad \text{implies} \quad f(K_1) \neq f(K_2). \tag{6}$$

Of course $f(K)$ takes on values in the range 0 to $w - 1$, so it isn't any good as a hash function, but it can be very useful as a *scrambling function*, namely a function satisfying (6) and tending to randomize the keys. Such a function can be very useful in connection with the tree search algorithms of Section 6.2.2, if the order of keys is unimportant, since it removes the danger of degeneracy when keys enter the tree in increasing order. A scrambling function is also useful in connection with the digital tree search algorithm of Section 6.3, if the bits of the actual keys are biased.

Another feature of the multiplicative hash method is that it makes good use of the nonrandomness found in many files. Actual sets of keys often have a preponderance of arithmetic progressions, where $\{K, K + d, K + 2d, \ldots, K + td\}$ all appear in the file; for example, consider alphabetic names like {PART1, PART2, PART3} or {TYPEA, TYPEB, TYPEC}. The multiplicative hash method converts an arithmetic progression into an approximate arithmetic progression $h(K)$, $h(K + d)$, $h(K + 2d)$, ... of distinct hash values, reducing the number of collisions from what we would expect in a random situation. The division method has this same property.

Figure 37 illustrates this aspect of multiplicative hashing in a particularly interesting case. Suppose that A/w is approximately the golden ratio $\phi^{-1} = (\sqrt{5} - 1)/2 = 0.6180339887$; then the behavior of successive values $h(K)$, $h(K + 1)$, $h(K + 2)$, ... can be studied by considering the behavior of the successive values $h(0)$, $h(1)$, $h(2)$, ... This suggests the following experiment: Starting with the line segment [0, 1], we successively mark off the points $\{\phi^{-1}\}$, $\{2\phi^{-1}\}$, $\{3\phi^{-1}\}$, ..., where $\{x\}$ denotes the fractional part of x (namely $x - \lfloor x \rfloor = x \bmod 1$). As shown in Fig. 37, these points stay very well separated from each other; in fact, each newly added point falls into one of the largest remaining intervals, and divides it in the golden ratio! [This phenomenon was

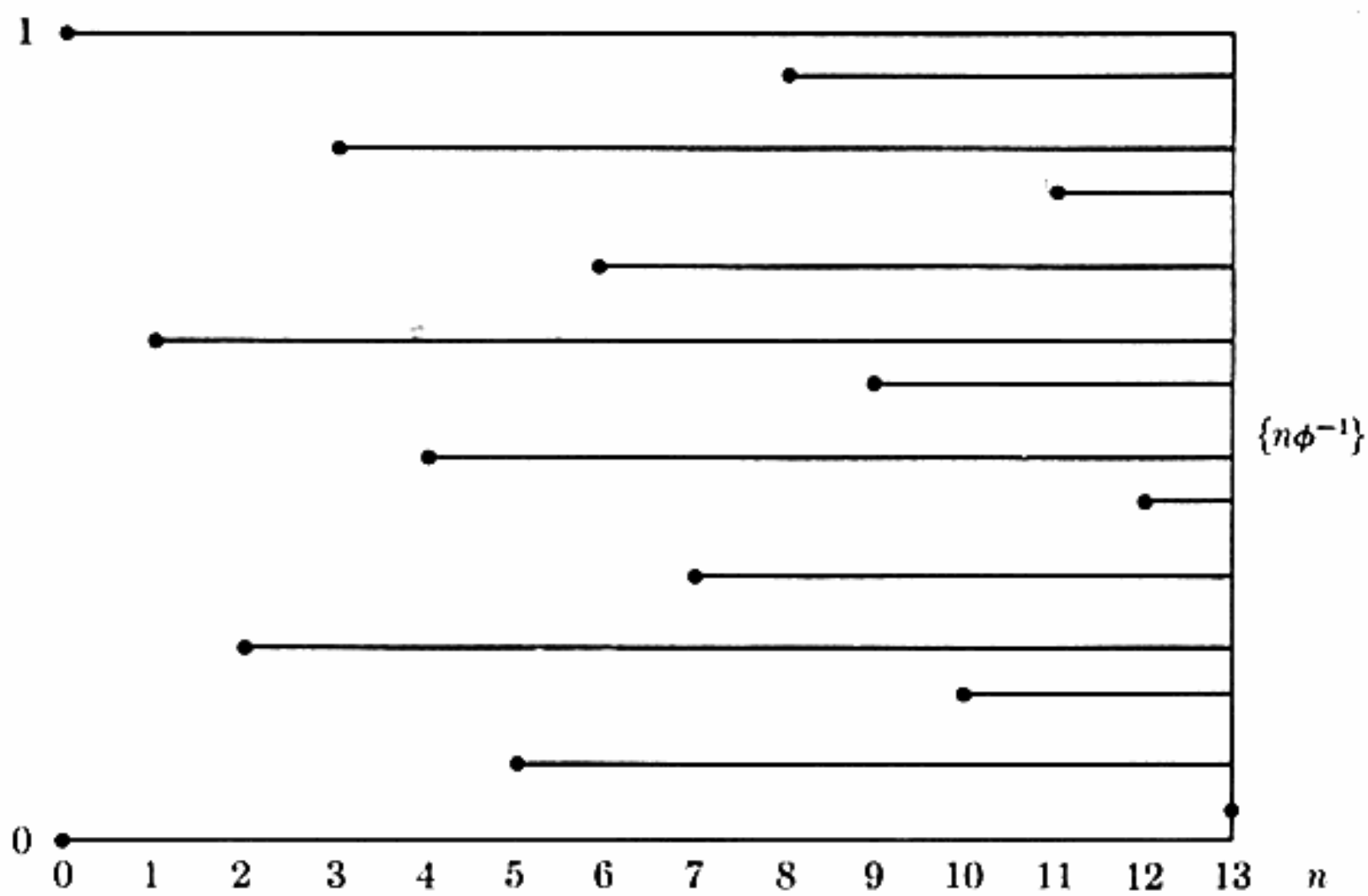


Fig. 37. Fibonacci hashing.

first conjectured by J. Oderfeld and proved by S. Świerczkowski, *Fundamenta Math.* **46** (1958), 187–189. Fibonacci numbers play an important rôle in the proof.]

This remarkable property of the golden ratio is actually just a special case of a very general result, originally conjectured by Hugo Steinhaus and first proved by Vera Turán Sós [*Acta Math. Acad. Sci. Hung.* **8** (1957), 461–471; *Ann. Univ. Sci. Budapest. Eötvös Sect. Math.* **1** (1958), 127–134]:

Theorem S. *Let θ be any irrational number. When the points $\{\theta\}$, $\{2\theta\}$, ..., $\{n\theta\}$ are placed in the line segment* [0, 1], *the $n+1$ line segments formed have at most three different lengths. Moreover, the next point $\{(n+1)\theta\}$ will fall in one of the largest existing segments.* ▮

Thus, the points $\{\theta\}$, $\{2\theta\}$, ..., $\{n\theta\}$ are spread out very evenly between 0 and 1. If θ is rational, the same theorem holds if we give a suitable interpretation to the segments of length 0 that appear when n is greater than or equal to the denominator of θ. A proof of Theorem S, together with a detailed analysis of the underlying structure of the situation, appears in exercise 8; it turns out that the segments of a given length are created and destroyed in a first-in-first-out manner. Of course, some θ's are better than others, since for example a value that is near 0 or 1 will start out with many small segments and one large segment. Exercise 9 shows that the two numbers ϕ^{-1} and $\phi^{-2} = 1 - \phi^{-1}$ lead to the "most uniformly distributed" sequences, among all numbers θ between 0 and 1.

The above theory suggests *Fibonacci hashing*, where we choose the constant A to be the nearest integer to $\phi^{-1}w$ that is relatively prime to w. For example if `MIX` were a decimal computer we would take

$$A = \boxed{+}\boxed{61}\boxed{80}\boxed{33}\boxed{98}\boxed{87}\ . \tag{7}$$

This multiplier will spread out alphabetic keys like `LIST1`, `LIST2`, `LIST3` very nicely. But notice what happens when we have an arithmetic series in the fourth character position, as in the keys `SUM1␣`, `SUM2␣`, `SUM3␣`: The effect is as if Theorem S were being used with $\theta = \{100A/w\} = .80339887$ instead of $\theta = .6180339887 = A/w$. The resulting behavior is still all right, in spite of the fact that this value of θ is not quite as good as ϕ^{-1}. On the other hand, if the progression occurs in the second character position, as in `A1␣␣␣`, `A2␣␣␣`, `A3␣␣␣`, the effective θ is .9887, and this is probably too close to 1.

Therefore we might do better with a multiplier like

$$A = \boxed{+}\boxed{61}\boxed{61}\boxed{61}\boxed{61}\boxed{61} \tag{8}$$

in place of (7); such a multiplier will separate out consecutive sequences of keys differing in *any* character position. Unfortunately this choice suffers from another problem analogous to the difficulty of dividing by $r^k \pm 1$: Keys such as `XY` and `YX` will tend to hash to the same location! One way out of this diffi-

culty is to look more closely at the structure underlying Theorem S; for short progressions of keys, only the first few partial quotients of the continued fraction representation of θ are relevant, and small partial quotients correspond to good distribution properties. Therefore we find that the best values of θ lie in the ranges

$$\tfrac{1}{4} < \theta < \tfrac{3}{10}, \qquad \tfrac{1}{3} < \theta < \tfrac{3}{7}, \qquad \tfrac{4}{7} < \theta < \tfrac{2}{3}, \qquad \tfrac{7}{10} < \theta < \tfrac{3}{4}.$$

A value of A can be found so that each of its bytes lies in a good range and is not too close to the values of the other bytes or their complements, e.g.,

$$A = \boxed{+}\boxed{61}\boxed{25}\boxed{42}\boxed{33}\boxed{71}. \tag{9}$$

Such a multiplier can be recommended. (These ideas about multiplicative hashing are due largely to R. W. Floyd.)

A good hash function should satisfy two requirements:

a) Its computation should be very fast.
b) It should minimize collisions.

Property (a) is somewhat machine-dependent, and property (b) is data-dependent. If the keys were truly random, we could simply extract a few bits from them and use these bits for the hash function; but in practice we nearly always need to have a hash function that depends on all bits of the key in order to satisfy (b).

So far we have considered how to hash one-word keys. Multiword or variable-length keys can be handled by multiple-precision extensions of the above methods, but it is generally adequate to speed things up by combining the individual words together into a single word, then doing a single multiplication or division as above. The combination can be done by addition mod w, or by "exclusive or" on a binary computer; both of these operations have the advantage that they are invertible, i.e., that they depend on all bits of both arguments, and exclusive-or is sometimes preferable because it avoids arithmetic overflow. Note that both of these operations are commutative, so that (X, Y) and (Y, X) will hash to the same address; G. D. Knott has suggested avoiding this problem by doing a cyclic shift just before adding or exclusive-oring.

Many more methods for hashing have been suggested, but none of these has proved to be superior to the simple division and multiplication methods described above. For a survey of some other methods together with detailed statistics on their performance with actual files, see the article by V. Y. Lum, P. S. T. Yuen, and M. Dodd, *CACM* **14** (1971), 228–239.

Of all the other hash methods that have been tried, perhaps the most interesting is a technique based on algebraic coding theory; the idea is analogous to the division method above, but we divide by a polynomial modulo 2 instead of dividing by an integer. (As observed in Section 4.6, this operation is analogous to division, just as addition is analogous to exclusive-or.) For this method, M

should be a power of 2, say $M = 2^m$, and we make use of an mth degree polynomial $P(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_0$. An n-digit binary key $K = (k_{n-1} \ldots k_1 k_0)_2$ can be regarded as the polynomial $K(x) = k_{n-1}x^{n-1} + \cdots + k_1 x + k_0$, and we compute the remainder

$$K(x) \bmod P(x) = h_{m-1}x^{m-1} + \cdots + h_1 x + h_0$$

using polynomial arithmetic modulo 2; then $h(K) = (h_{m-1} \ldots h_1 h_0)_2$. If $P(x)$ is chosen properly, this hash function can be guaranteed to avoid collisions between nearly-equal keys. For example if $n = 15$, $m = 10$, and

$$P(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1, \tag{10}$$

it can be shown that $h(K_1)$ will be unequal to $h(K_2)$ whenever K_1 and K_2 are distinct keys that differ in fewer than seven bit positions. (See exercise 7 for further information about this scheme; it is, of course, more suitable for hardware or microprogramming implementation than for software.)

It has been found convenient to use the constant hash function $h(K) = 0$ when debugging a program, since all keys will be stored together; an efficient $h(K)$ can be substituted later.

Collision resolution by "chaining." We have observed that some hash addresses will probably be burdened with more than their share of keys. Perhaps the most obvious way to solve this problem is to maintain M linked lists, one for each possible hash code. A `LINK` field should be included in each record, and there will also be M list heads, numbered say from 1 through M. After hashing the key, we simply do a sequential search in list number $h(K) + 1$. (Cf. exercise 6.1–2. The situation is very similar to multiple-list-insertion sorting, Program 5.2.1M.)

Figure 38 illustrates this simple chaining scheme when $M = 9$, for the sequence of seven keys

$$K = \texttt{EN, TO, TRE, FIRE, FEM, SEKS, SYV} \tag{11}$$

(i.e., the numbers 1 through 7 in Norwegian), having the respective hash codes

$$h(K) + 1 = \ 3, \quad 1, \quad 4, \quad 1, \quad 5, \quad 9, \quad 2. \tag{12}$$

The first list has two elements, and three of the lists are empty.

Chaining is quite fast, because the lists are short. If 365 people are gathered together in one room, there will probably be many pairs having the same birthday, but the average number of people with any given birthday will be only 1! In general, if there are N keys and M lists, the average list size is N/M; thus hashing decreases the amount of work needed for sequential searching by roughly a factor of M.

This method is a straightforward combination of techniques we have discussed before, so we do not need to formulate a detailed algorithm for chained scatter tables. It is often a good idea to keep the individual lists in order by key, so that insertions and unsuccessful searches go faster. Thus if we choose

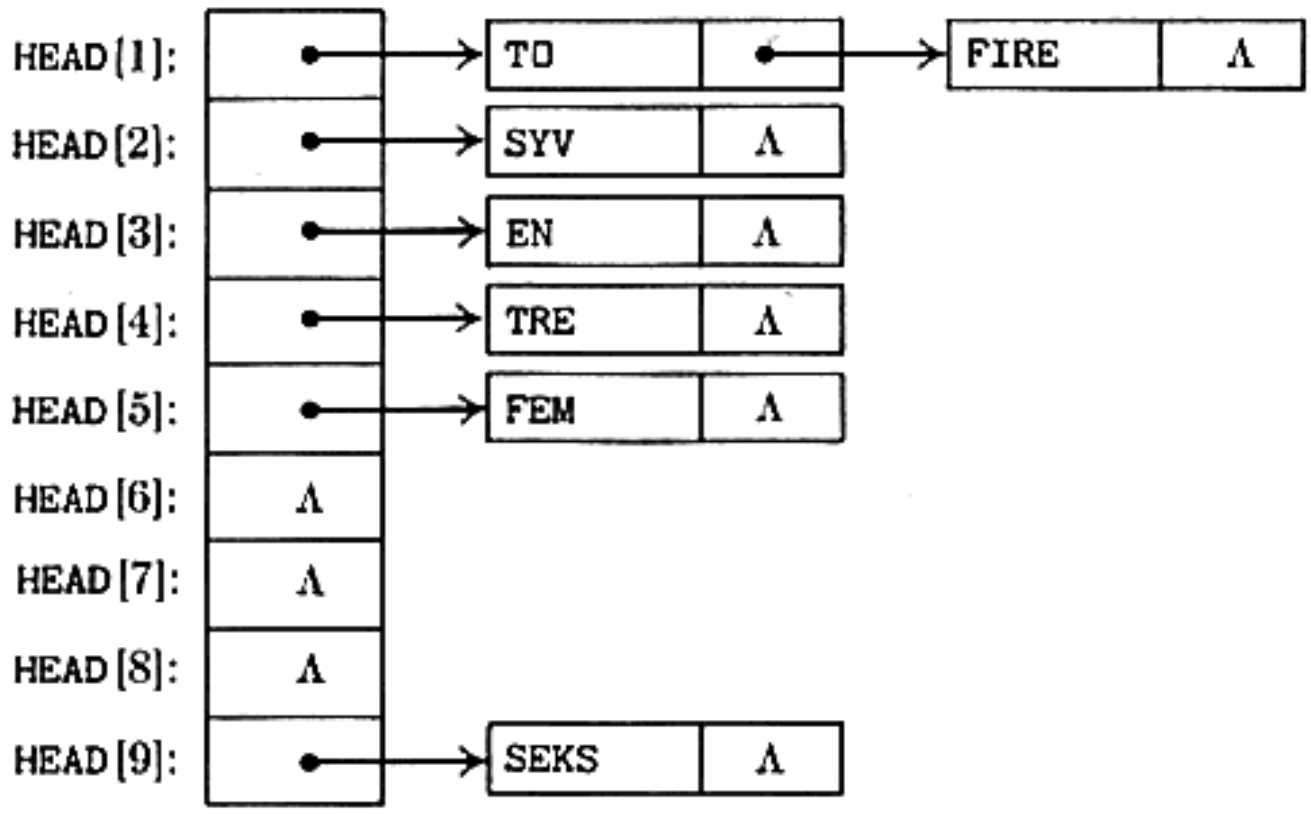


Fig. 38. Separate chaining.

to make the lists ascending, the TO and FIRE nodes of Fig. 38 would be interchanged, and all the Λ links would be replaced by pointers to a dummy record whose key is ∞. (Cf. Algorithm 6.1T.) Alternatively we can make use of the "self-organizing file" concept discussed in Section 6.1; instead of keeping the lists in order by key, they may be kept in order according to the time of most recent occurrence.

For the sake of speed we would like to make M rather large. But when M is large, many of the lists will be empty and much of the space for the M list heads will be wasted. This suggests another approach, when the records are small: We can overlap the record storage with the list heads, making room for a total of M records and M links instead of for N records and $M + N$ links. Sometimes it is possible to make one pass over all the data to find out which list heads will be used, then to make another pass inserting all the "overflow" records into the empty slots. But this is often impractical or impossible, and we would rather have a technique that processes each record only once when it first enters the system. The following algorithm, due to F. A. Williams [*CACM* **2**, 6 (June 1959), 21–24], is a convenient way to solve the problem.

Algorithm C (*Chained scatter table search and insertion*). This algorithm searches an M-node table, looking for a given key K. If K is not in the table and the table is not full, K is inserted.

The nodes of the table are denoted by TABLE[i], for $0 \le i \le M$, and they are of two distinguishable types, *empty* and *occupied*. An occupied node contains a key field KEY[i], a link field LINK[i], and possibly other fields.

The algorithm makes use of a hash function $h(K)$. An auxiliary variable R is also used, to help find empty spaces; when the table is empty, we have R $= M + 1$, and as insertions are made it will always be true that TABLE[j] is occupied for all j in the range R $\le j \le M$. By convention, TABLE[0] will always be empty.

C1. [Hash.] Set $i \leftarrow h(K) + 1$. (Now $1 \le i \le M$.)

C2. [Is there a list?] If TABLE[i] is empty, go to C6. (Otherwise TABLE[i] is occupied; we will look at the list of occupied nodes which starts here.)

C3. [Compare.] If $K =$ KEY[i], the algorithm terminates successfully.

C4. [Advance to next.] If LINK[i] $\neq 0$, set $i \leftarrow$ LINK[i] and go back to step C3.

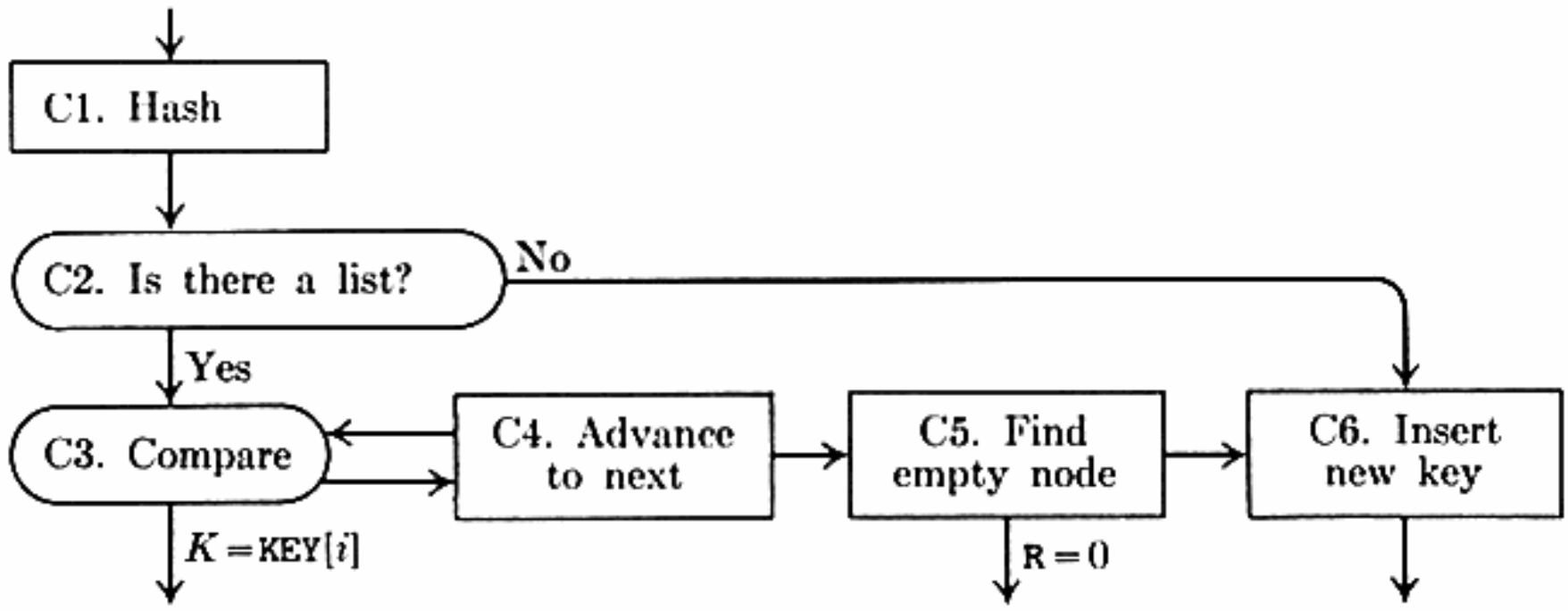


Fig. 39. Chained scatter table search and insertion.

C5. [Find empty node.] (The search was unsuccessful, and we want to find an empty position in the table.) Decrease R one or more times until finding a value such that TABLE[R] is empty. If R = 0, the algorithm terminates with overflow (there are no empty nodes left); otherwise set LINK[i] $\leftarrow$ R, $i \leftarrow$ R.

C6. [Insert new key.] Mark TABLE[i] as an occupied node, with KEY[i] $\leftarrow K$ and LINK[i] $\leftarrow 0$. ∎

This algorithm allows several lists to coalesce, so that records need not be moved after they have been inserted into the table. For example, see Fig. 40, where SEKS appears in the list containing TO and FIRE since the latter had already been inserted into position 9.

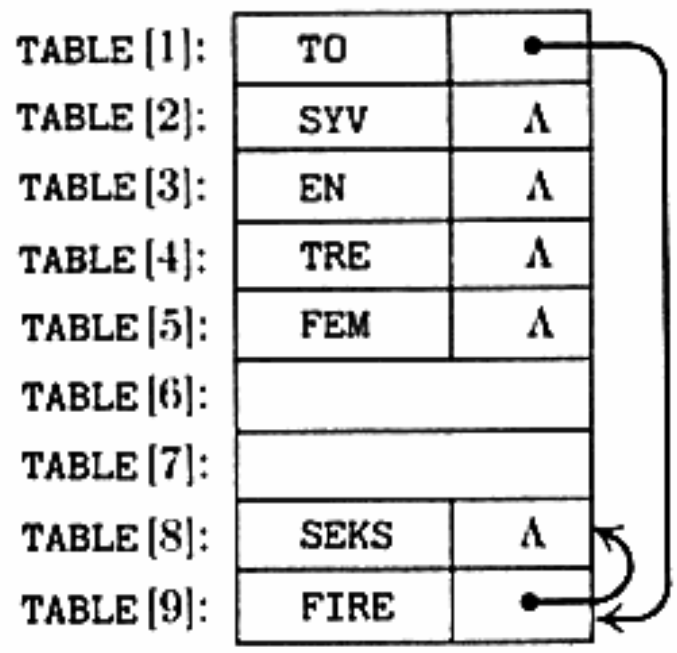


Fig. 40. Coalesced chaining.

In order to see how this algorithm compares with others in this chapter, we can write the following MIX program. The analysis worked out below indicates that the lists of occupied cells tend to be short, and the program has been designed with this fact in mind.

Program C (*Chained scatter table search and insertion*). For convenience, the keys are assumed to be only three bytes long, and nodes are represented as follows:

empty	—	1	0	0	0	0
occupied	+	LINK		KEY		

(13)

The table size M is assumed to be prime; TABLE[i] is stored in location TABLE$+i$. rI1 $\equiv i$, rA $\equiv K$.

```
01  KEY     EQU   3:5
02  LINK    EQU   0:2
03  START   LDX   K                    1          C1. Hash.
04          ENTA  0                    1
05          DIV   =M=                  1
06          STX   *+1(0:2)             1
07          ENT1  *                    1          i ← h(k)
08          INC1  1                    1                  + 1.
09          LDA   K                    1
10          LD2   TABLE,1(LINK)        1          C2. Is there a list?
11          J2N   6F                   1          To C6 if TABLE[i] empty.
12          CMPA  TABLE,1(KEY)         A          C3. Compare.
13          JE    SUCCESS              A          Exit if K = KEY[i].
14          J2Z   5F                   A − S1     To C5 if LINK[i] = 0.
15  4H      ENT1  0,2                  C − 1      C4. Advance to next.
16          CMPA  TABLE,1(KEY)         C − 1      C3. Compare.
17          JE    SUCCESS              C − 1      Exit if K = KEY[i].
18          LD2   TABLE,1(LINK)        C − 1 − S2
19          J2NZ  4B                   C − 1 − S2 Advance if LINK[i] ≠ 0.
20  5H      LD2   R                      A − S    C5. Find empty node.
21          DEC2  1                      T        R ← R − 1.
22          LDX   TABLE,2                T
23          JXNN  *-2                    T        Repeat until TABLE[R] empty.
24          J2Z   OVERFLOW             A − S      Exit if no empty nodes left.
25          ST2   TABLE,1(LINK)        A − S      LINK[i] ← R.
26          ENT1  0,2                  A − S      i ← R.
27          ST2   R                    A − S      Update R in memory.
28  6H      STZ   TABLE,1(LINK)        1 − S      C6. Insert new key.
29          STA   TABLE,1(KEY)         1 − S      KEY[i] ← K. ∎
```

The running time of this program depends on

C = number of table entries probed while searching;

$A = 1$ if the initial probe found an occupied node;

$S = 1$ if successful, 0 if unsuccessful;

$T =$ number of table entries probed while looking for an empty space.

Here $S = S1 + S2$, where $S1 = 1$ if successful on the first try. The total running time for the searching phase of Program C is $(7C + 4A + 17 - 3S + 2S1)u$, and the insertion of a new key when $S = 0$ takes an additional $(8A + 4T + 4)u$.

Suppose there are N keys in the table at the start of this program, and let

$$\alpha = N/M = \text{load factor of the table.} \tag{14}$$

Then the average value of A in an unsuccessful search is obviously α, if the hash function is random; and exercise 39 proves that the average value of C in an unsuccessful search is

$$C'_N = 1 + \frac{1}{4}\left(\left(1 + \frac{2}{M}\right)^N - 1 - \frac{2N}{M}\right) \approx 1 + \tfrac{1}{4}(e^{2\alpha} - 1 - 2\alpha). \tag{15}$$

Thus when the table is half full, the average number of probes made in an unsuccessful search is about $\frac{1}{4}(e + 2) \approx 1.18$; and even when the table gets completely full, the average number of probes made just before inserting the final item will be only about $\frac{1}{4}(e^2 + 1) \approx 2.10$. The standard deviation is also small, as shown in exercise 40. These statistics prove that *the lists stay short even though the algorithm occasionally allows them to coalesce*, when the hash function is random. Of course C can be as high as N, if the hash function is bad or if we are extremely unlucky.

In a successful search, we always have $A = 1$. The average number of probes during a successful search may be computed by summing the quantity $C + A$ over the first N unsuccessful searches and dividing by N, if we assume that each key is equally likely. Thus we obtain

$$C_N = \frac{1}{N}\sum_{0\le k<N}\left(C'_k + \frac{k}{M}\right) = 1 + \frac{1}{8}\,\frac{M}{N}\left(\left(1 + \frac{2}{M}\right)^N - 1 - \frac{2N}{M}\right) + \frac{1}{4}\,\frac{N-1}{M}$$

$$\approx 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \tfrac{1}{4}\alpha \tag{16}$$

as the average number of probes in a random successful search. Even a full table will require only about 1.80 probes, on the average, to find an item! Similarly (see exercise 42), the average value of $S1$ turns out to be

$$S1_N = 1 - \tfrac{1}{2}((N-1)/M) \approx 1 - \tfrac{1}{2}\alpha. \tag{17}$$

At first glance it may appear that step C5 is inefficient, since it has to search sequentially for an empty position. But actually the total number of table probes made in step C5 as a table is being built will never exceed the number of items in the table; so we make an average of at most one of these probes per insertion! Exercise 41 proves that T is approximately αe^{α} in a random unsuccessful search.

It would be possible to modify Algorithm C so that no two lists coalesce, but then it would become necessary to move records around. For example,

consider the situation in Fig. 40 just before we wanted to insert SEKS into position 9; in order to keep the lists separate, it would be necessary to move FIRE, and for this purpose it would be necessary to discover which node points to FIRE. We could solve this problem without providing two-way linkage by using circular lists, as suggested by Allen Newell in 1962, since the lists are short; but that would probably slow down the main search loop because step C4 would be more complicated. Exercise 34 shows that the average number of probes, when lists aren't coalesced, is reduced to

$$C'_N = \left(1 - \frac{1}{M}\right)^N + \frac{N}{M} \approx e^{-\alpha} + \alpha \qquad \text{(unsuccessful search)}; \qquad (18)$$

$$C_N = \quad 1 + \frac{N-1}{2M} \quad \approx 1 + \tfrac{1}{2}\alpha \qquad \text{(successful search)}. \qquad (19)$$

This is not enough of an improvement over (15) and (16) to warrant changing the algorithm.

On the other hand, Butler Lampson has observed that most of the space actually needed for links can actually be saved in the chaining method, if we avoid coalescing the lists. This leads to an interesting algorithm which is discussed in exercise 13.

Note that chaining can be used when $N > M$, so overflow is not a serious problem. If separate lists are used, formulas (18) and (19) are valid for $\alpha > 1$. When the lists coalesce as in Algorithm C, we can link extra items into an auxiliary storage pool; L. Guibas has proved that the average number of probes to insert the $(M + L + 1)$st item is then $(L/2M + \frac{1}{4})((1 + 2/M)^M - 1) + \frac{1}{2}$.

Collision resolution by "open addressing." Another way to resolve the problem of collisions is to do away with the links entirely, simply looking at various entries of the table one by one until either finding the key K or finding an empty position. The idea is to formulate some rule by which every key K determines a "probe sequence," namely a sequence of table positions which are to be inspected whenever K is inserted or looked up. If we encounter an open position while searching for K, using the probe sequence determined by K, we can conclude that K is not in the table, since the same sequence of probes will be made every time K is processed. This general class of methods was named *open addressing* by W. W. Peterson [*IBM J. Research & Development* **1** (1957), 130–146].

The simplest open addressing scheme, known as *linear probing*, uses the cyclic probe sequence

$$h(K),\ h(K) - 1,\ \ldots,\ 0,\ M - 1,\ M - 2,\ \ldots,\ h(K) + 1 \qquad (20)$$

as in the following algorithm.

Algorithm L (*Open scatter table search and insertion*). This algorithm searches an M-node table, looking for a given key K. If K is not in the table and the table is not full, K is inserted.

The nodes of the table are denoted by TABLE[i], for $0 \le i < M$, and they are of two distinguishable types, *empty* and *occupied*. An occupied node contains a key, called KEY[i], and possibly other fields. An auxiliary variable N is used to keep track of how many nodes are occupied; this variable is considered to be part of the table, and it is increased by 1 whenever a new key is inserted.

This algorithm makes use of a hash function $h(K)$, and it uses the linear probing sequence (20) to address the table. Modifications of that sequence are discussed below.

L1. [Hash.] Set $i \leftarrow h(K)$. (Now $0 \le i < M$.)

L2. [Compare.] If TABLE[i] is empty, go to L4. Otherwise if KEY[i] $= K$, the algorithm terminates successfully.

L3. [Advance to next.] Set $i \leftarrow i - 1$; if now $i < 0$, set $i \leftarrow i + M$. Go back to step L2.

L4. [Insert.] (The search was unsuccessful.) If N $= M - 1$, the algorithm terminates with overflow. (This algorithm considers the table to be full when N $= M - 1$, not when N $= M$; see exercise 15.) Otherwise set N $\leftarrow$ N $+ 1$, mark TABLE[i] occupied, and set KEY[i] $\leftarrow K$. ∎

Figure 41 shows what happens when the seven example keys (11) are inserted by Algorithm L, using the respective hash codes 2, 7, 1, 8, 2, 8, 1: The last three keys, FEM, SEKS, and SYV, have been displaced from their initial locations $h(K)$.

0	FEM
1	TRE
2	EN
3	
4	
5	SYV
6	SEKS
7	TO
8	FIRE

Fig. 41. Linear open addressing.

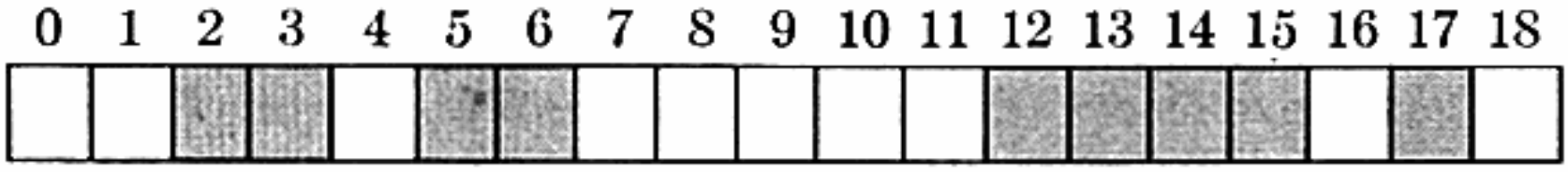
Program L (*Open scatter table search and insertion*). This program deals with full-word keys; but a key of 0 is not allowed, since 0 is used to signal an empty position in the table. (Alternatively, we could require the keys to be nonnegative, letting empty positions contain -1.) The table size M is assumed to be prime, and TABLE[i] is stored in location TABLE $+ i$ for $0 \le i < M$. For speed in the inner loop, location TABLE $- 1$ is assumed to contain 0. Location VACANCIES is assumed to contain the value $M - 1 -$ N; and rA $\equiv K$, rI1 $\equiv i$.

In order to speed up the inner loop of this program, the test "$i < 0$" has been removed from the loop so that only the essential parts of steps L2 and L3 remain. The total running time for the searching phase comes to $(7C + 9E + 21 - 4S)u$, and the insertion after an unsuccessful search adds an extra $9u$.

```
01  START  LDX   K              1          L1. Hash.
02         ENTA  0              1
03         DIV   =M=            1
04         STX   *+1(0:2)       1
05         ENT1  *              1          i ← h(K).
06         LDA   K              1
07         JMP   2F             1
08  8H     INC1  M+1            E          L3. Advance to next.
09  3H     DEC1  1          C + E − 1      i ← i − 1.
10  2H     CMPA  TABLE,1        C + E      L2. Compare.,
11         JE    SUCCESS        C + E      Exit if K = KEY[i].
12         LDX   TABLE,1        C + E − S
13         JXNZ  3B             C + E − S  To L3 if TABLE[i] empty.
14         J1N   8B             E + 1 − S  To L3 with i ← M if i = −1.
15  4H     LDX   VACANCIES      1 − S      L4. Insert.
16         JXZ   OVERFLOW       1 − S      Exit with overflow if N = M − 1.
17         DECX  1              1 − S
18         STX   VACANCIES      1 − S      Increase N by 1.
19         STA   TABLE,1        1 − S      TABLE[i] ← K. ∎
```

As in Program C, the variable C denotes the number of probes, and S tells whether or not the search was successful. We may ignore the variable E, which is 1 only if a spurious probe of TABLE[−1] has been made, since its average value is $(C - 1)/M$.

Experience with linear probing shows that the algorithm works fine until the table begins to get full; but eventually the process slows down, with long drawn-out searches becoming increasingly frequent. The reason for this behavior can be understood by considering the following hypothetical scatter table with $M = 19$, $N = 9$:



(21)

Shaded squares represent occupied positions. The next key K to be inserted into the table will go into one of the ten empty spaces, but these are not equally likely; in fact, K will be inserted into position 11 if $11 \le h(K) \le 15$, while it will fall into position 8 only if $h(K) = 8$. Therefore position 11 is five times as likely as position 8; long lists tend to grow even longer.

This phenomenon isn't enough by itself to account for the relatively poor behavior of linear probing, since a similar thing occurs in Algorithm C. (A list of length 4 is four times as likely to grow in Algorithm C as a list of length 1.) The real problem occurs when a cell like 4 or 16 is filled in (21); then two separate lists are combined, while the lists in Algorithm C never grow by more than one step at a time. Consequently the performance of linear probing degrades rapidly when N approaches M.

We shall prove later in this section that the average number of probes needed by Algorithm L is approximately

$$C'_N \approx \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right) \qquad \text{(unsuccessful search)}; \tag{22}$$

$$C_N \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \text{(successful search)}, \tag{23}$$

where $\alpha = N/M$ is the load factor of the table. So Program L is almost as fast as Program C, when the table is less than 75 percent full, in spite of the fact that Program C deals with unrealistically short keys. On the other hand, when α approaches 1 the best thing we can say about Program L is that it works, slowly but surely. In fact, when $N = M - 1$, there is only one vacant space in the table, so the average number of probes in an unsuccessful search is $(M + 1)/2$; we shall also prove that the average number of probes in a successful search is approximately $\sqrt{\pi M/8}$ when the table is full.

The pileup phenomenon which makes linear probing costly on a nearly full table is aggravated by the use of division hashing, if consecutive key values $\{K, K + 1, K + 2, \ldots\}$ are likely to occur, since these keys will have consecutive hash codes. Multiplicative hashing will break up these clusters satisfactorily. Note that multiplicative hashing is slightly awkward for the chained method, since we generally would want to use at least $(m + 1)$-bit link fields when $M = 2^m$ in order to distinguish rapidly between Λ and a valid link. This wasted bit position is no problem with open addressing since there are no links.

Another way to protect against the consecutive hash code problem is to set $i \leftarrow i - c$ in step L3, instead of $i \leftarrow i - 1$. Any positive value of c will do, so long as it is *relatively prime* to M, since the probe sequence will still examine every position of the table in this case. Such a change will make Program L somewhat slower. It doesn't alter the pileup phenomenon, since groups of c-apart records will still be formed; equations (22) and (23) will still apply, but the appearance of consecutive keys $\{K, K + 1, K + 2, \ldots\}$ will now actually be a help instead of a hindrance.

Although a fixed value of c does not reduce the pileup phenomenon, we can improve the situation nicely by letting c depend on K! This idea leads to an important modification of Algorithm L, first discovered by Guy de Balbine [Ph.D. thesis, Calif. Inst. of Technology (1968), 149–150]:

Algorithm D (*Open addressing with double hashing*). This algorithm is almost identical to Algorithm L, but it probes the table in a slightly different fashion by making use of two hash functions $h_1(K)$ and $h_2(K)$. As usual $h_1(K)$ produces a value between 0 and $M - 1$, inclusive; but $h_2(K)$ must produce a value between 1 and $M - 1$ that is *relatively prime* to M. (For example, if M is prime, $h_2(K)$ can be *any* value between 1 and $M - 1$ inclusive; or if $M = 2^m$, $h_2(K)$ can be any *odd* value between 1 and $2^m - 1$.)

D1. [First hash.] Set $i \leftarrow h_1(K)$.

D2. [First probe.] If TABLE[i] is empty, go to D6. Otherwise if KEY[i] $= K$, the algorithm terminates successfully.

D3. [Second hash.] Set $c \leftarrow h_2(K)$.

D4. [Advance to next.] Set $i \leftarrow i - c$; if now $i < 0$, set $i \leftarrow i + M$.

D5. [Compare.] If TABLE[i] is empty, go to D6. Otherwise if KEY[i] $= K$, the algorithm terminates successfully. Otherwise go back to D4.

D6. [Insert.] If N $= M - 1$, the algorithm terminates with overflow. Otherwise set N $\leftarrow$ N $+ 1$, mark TABLE[i] occupied, and set KEY[i] $\leftarrow K$. ∎

Several possibilities have been suggested for computing $h_2(K)$. If M is prime and $h_1(K) = K \bmod M$, we might let $h_2(K) = 1 + (K \bmod (M - 1))$; but since $M - 1$ is even, it would be better to let $h_2(K) = 1 + (K \bmod (M - 2))$. This suggests choosing M so that M and $M - 2$ are "twin primes" like 1021 and 1019. Alternatively, we could set $h_2(K) = 1 + (\lfloor K/M \rfloor \bmod (M - 2))$, since the quotient $\lfloor K/M \rfloor$ might be available in a register as a by-product of the computation of $h_1(K)$.

If $M = 2^m$ and we are using multiplicative hashing, $h_2(K)$ can be computed simply by shifting left m more bits and "oring in" a 1, so that the coding sequence (5) would be followed by

```
ENTA   0        Clear rA.
SLB    m        Shift rAX m bits left.          (24)
ORR    =1=      rA ← rA ∨ 1.
```

This is faster than the division method.

In each of the techniques suggested above, $h_1(K)$ and $h_2(K)$ are "independent," in the sense that different keys will have the same value for both h_1 and h_2 with probability $O(1/M^2)$ instead of $O(1/M)$. Empirical tests show that the behavior of Algorithm D with independent hash functions is essentially indistinguishable from the number of probes which would be required if the keys were inserted at random into the table; there is practically no "piling up" or "clustering" as in Algorithm L.

It is also possible to let $h_2(K)$ depend on $h_1(K)$, as suggested by Gary Knott in 1968; for example, if M is prime we could let

$$h_2(K) = \begin{cases} 1, & \text{if} \quad h_1(K) = 0; \\ M - h_1(K), & \text{if} \quad h_1(K) > 0. \end{cases} \tag{25}$$

This would be faster than doing another division, but we shall see that it does cause a certain amount of *secondary clustering*, requiring slightly more probes because of the increased chance that two or more keys will follow the same path. The formulas derived below can be used to determine whether the gain in hashing time outweighs the loss of probing time.

Algorithms L and D are very similar, yet there are enough differences that it is instructive to compare the running time of the corresponding MIX programs.

Program D (*Open addressing with double hashing*). Since this program is substantially like Program L, it is presented without comments. rI2 $\equiv c - 1$.

```
01    START  LDX    K              1
02           ENTA   0              1
03           DIV    =M=            1
04           STX    *+1(0:2)       1
05           ENT1   *              1
06           LDX    TABLE,1        1
07           CMPX   K              1
08           JE     SUCCESS        1
09           JXZ    4F             1 - S1
10           SRAX   5              A - S1
11           DIV    =M-2=          A - S1
12           STX    *+1(0:2)       A - S1
13           ENT2   *              A - S1
14           LDA    K              A - S1
15     3H    DEC1   1,2            C - 1
16           J1NN   *+2            C - 1
17           INC1   M              B
18           CMPA   TABLE,1        C - 1
19           JE     SUCCESS        C - 1
20           LDX    TABLE,1        C - 1 - S2
21           JXNZ   3B             C - 1 - S2
22     4H    LDX    VACANCIES      1 - S
23           JXZ    OVERFLOW       1 - S
24           DECX   1              1 - S
25           STX    VACANCIES      1 - S
26           LDA    K              1 - S
27           STA    TABLE, 1       1 - S
```

The frequency counts A, C, $S1$, $S2$ in this program have a similar interpretation to those in Program C above. The other variable B will be about $\frac{1}{2}(C - 1)$ on the average. (If we restricted the range of $h_2(K)$ to, say, $1 \le h_2(K) \le \frac{1}{2}M$, B would be only about $\frac{1}{4}(C - 1)$; this increase of speed will probably *not* be offset by a noticeable increase in the number of probes.) When there are $N = \alpha M$ keys in the table, the average value of A is, of course, α in an unsuccessful search, and $A = 1$ in a successful search. As in Algorithm C, the average value of $S1$ in a successful search is $1 - \frac{1}{2}((N - 1)/M) \approx 1 - \frac{1}{2}\alpha$. The average number of probes is difficult to determine exactly, but empirical tests show good agreement with formulas derived below for "uniform probing," namely

$$C'_N = \frac{M+1}{M+1-N} \approx (1-\alpha)^{-1} \quad \text{(unsuccessful search)}, \tag{26}$$

$$C_N = \frac{M+1}{N}(H_{M+1} - H_{M+1-N}) \approx -\alpha^{-1}\ln(1-\alpha) \quad \text{(successful search)}, \tag{27}$$

when $h_1(K)$ and $h_2(K)$ are independent. When $h_2(K)$ depends on $h_1(K)$ as in (25), the secondary clustering causes these formulas to be increased to

$$C'_N = \frac{M+1}{M+1-N} - \frac{N}{M+1} + H_{M+1} - H_{M+1-N} + O(M^{-1}) \approx (1-\alpha)^{-1} - \alpha - \ln(1-\alpha); \tag{28}$$

$$C_N = 1 + H_{M+1} - H_{M+1-N} - \frac{N}{2(M+1)} - (H_{M+1} - H_{M+1-N})/N + O(N^{-1}) \approx 1 - \ln(1-\alpha) - \tfrac{1}{2}\alpha. \tag{29}$$

(See exercise 44.) Note that as the table gets full, these values of C_N approach $H_{M+1} - 1$ and $H_{M+1} - \frac{1}{2}$, respectively, when $N = M$; this is much better than we observed in Algorithm L, but not as good as in the chaining methods.

Since each probe takes slightly less time in Algorithm L, double hashing is advantageous only when the table gets full. Figure 42 compares the average running time of Program L, Program D, and a modified Program D which involves secondary clustering, replacing the rather slow calculation of $h_2(K)$ in lines 10–13 by the three instructions

```
ENN2 -M-1,1          c ← M − i.
J1NZ *+2                                   (30)
ENT2 0               If i = 0, c ← 1.
```

In this case, secondary clustering is preferable to independent double hashing.

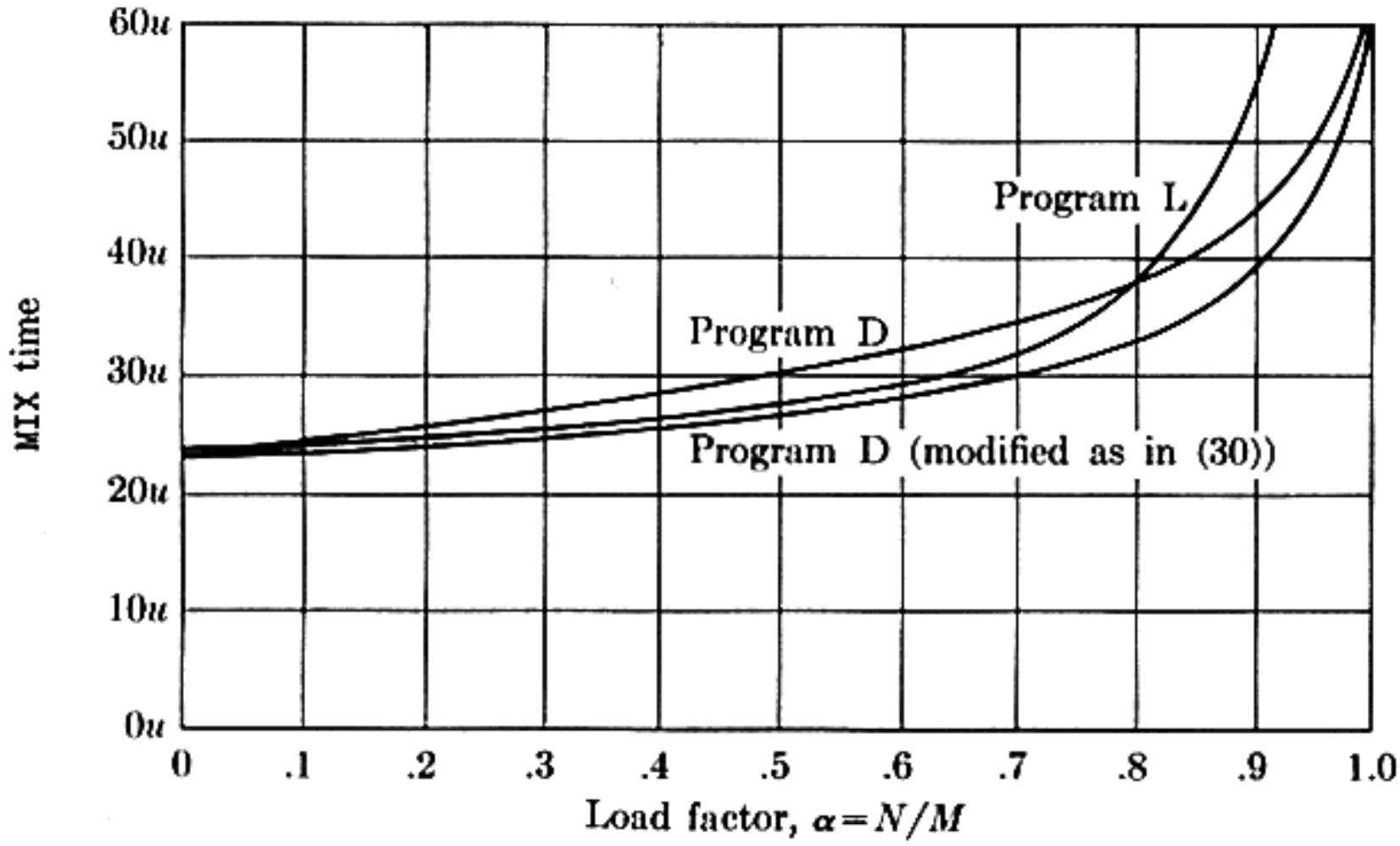


Fig. 42. The running time for successful searching by three open addressing schemes.

On a binary computer, we could speed up the computation of $h_2(K)$ in another way, replacing lines 10–13 by, say,

```
AND  =511=           rA ← rA mod 512.
STA  *+1(0:2)                              (31)
ENT2 *               c ← rA + 1.
```

if M is a prime greater than 512. This idea (suggested by Bell and Kaman, *CACM* **13** (1970), 675–677, who discovered Algorithm D independently) avoids secondary clustering without the expense of another division.

Many other probe sequences have been proposed as improvements on Algorithm L, but none seem to be superior to Algorithm D except possibly the method described in exercise 20.

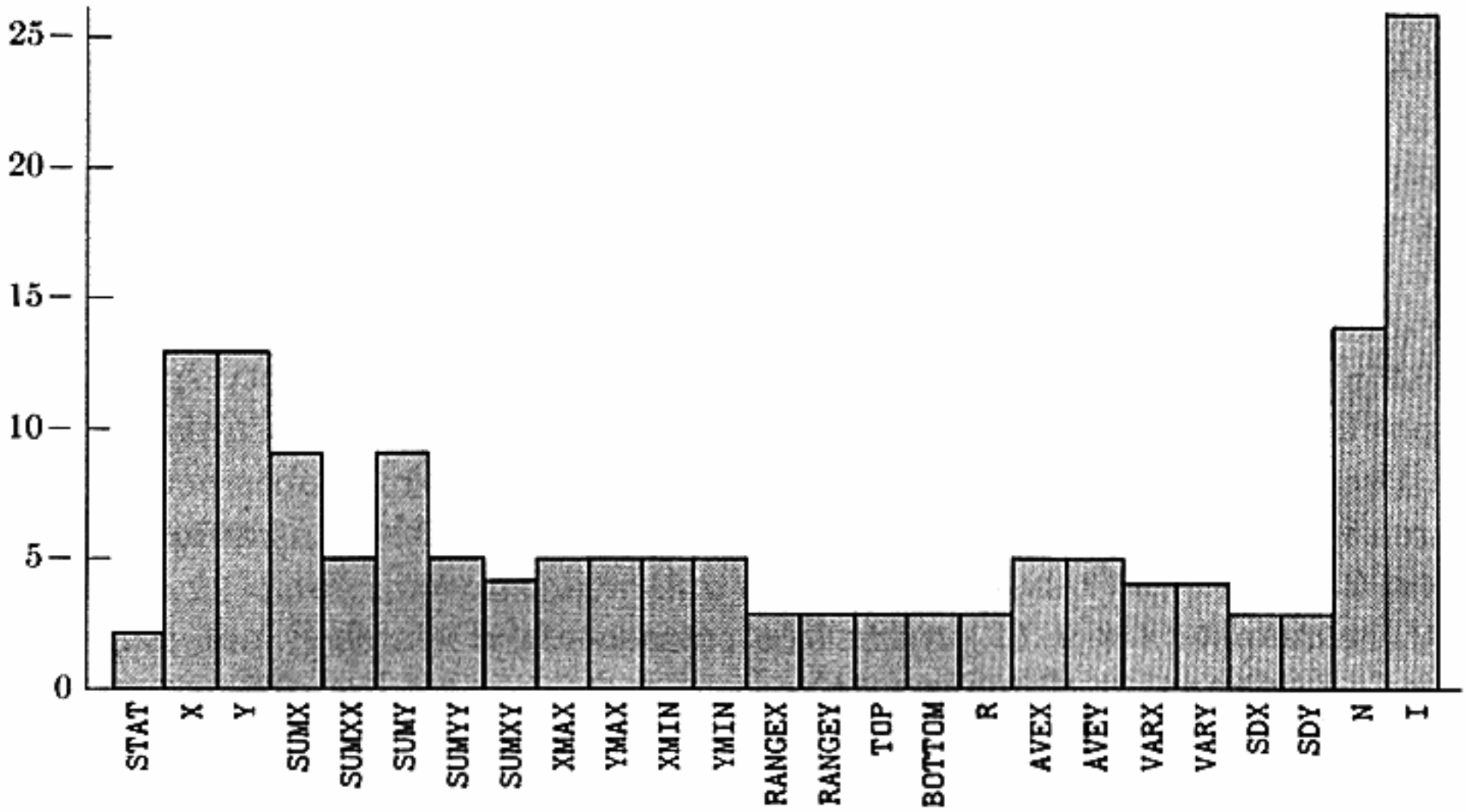


Fig. 43. The number of times a compiler typically searches for variable names. The names are listed from left to right in order of their first appearance.

Brent's Variation. Richard P. Brent has discovered a way to modify Algorithm D so that the average successful search time remains bounded as the table gets full. His method is based on the fact that successful searches are much more common than insertions, in many applications; therefore he proposes doing more work when inserting an item, moving records in order to reduce the expected retrieval time. [*CACM* **15** (1972), to appear.]

For example, Fig. 43 shows the number of times each identifier was actually found to appear, in a typical PL/I procedure. This data indicates that a PL/I compiler which uses a hash table to keep track of variable names will be looking up many of the names five or more times but inserting them only once. Similarly, Bell and Kaman found that a COBOL compiler used its symbol table algorithm 10988 times while compiling a program, but made only 735 insertions into the table; this is an average of about 14 successful searches per unsuccessful search. Sometimes a table is actually created only once (for example, a table of symbolic opcodes in an assembler), and it is used thereafter purely for retrieval.

Brent's idea is to change the insertion process in Algorithm D as follows. Suppose an unsuccessful search has probed locations

$$p_0, p_1, \cdots, p_{t-1}, p_t,$$

where $p_j = \big(h_1(K) - jh_2(K)\big) \bmod M$ and TABLE$[p_t]$ is empty. If $t \leq 1$, we insert K in position p_t as usual; but if $t \geq 2$, we compute $c_0 = h_2(K_0)$, where $K_0 =$ KEY$[p_0]$, and see if TABLE$[(p_0 - c_0) \bmod M]$ is empty. If it is, we set it to

TABLE[p_0] and then insert K in position p_0. This increases the retrieval time for K_0 by one step, but it decreases the retrieval time for K by $t \geq 2$ steps, so it results in a net improvement. Similarly, if TABLE[$(p_0 - c_0) \bmod M$] is occupied and $t \geq 3$, we try TABLE[$(p_0 - 2c_0) \bmod M$]; if that is full too, we compute $c_1 = h_2(\text{KEY}[p_1])$ and try TABLE[$(p_1 - c_1) \bmod M$]; etc. In general, let $c_j = h_2(\text{KEY}[p_j])$ and $p_{j,k} = (p_j - kc_j) \bmod M$; if we have found TABLE[$p_{j,k}$] occupied for all indices j, k such that $j + k < r$, and if $t \geq r + 1$, we look at TABLE[$p_{0,r}$], TABLE[$p_{1,r-1}$], . . . , TABLE[$p_{r-1,1}$]. If the first empty space occurs at position $p_{j,r-j}$ we set TABLE[$p_{j,r-j}$] $\leftarrow$ TABLE[p_j] and insert K in position p_j.

Brent's analysis indicates that the average number of probes per successful search is reduced as shown in Fig. 44, on page 539, with a maximum value of about 2.49.

The number $t + 1$ of probes in an unsuccessful search is not reduced by Brent's variation; it remains at the level indicated by Eq. (26), approaching $\frac{1}{2}(M + 1)$ as the table gets full. The average number of times h_2 needs to be computed per insertion is $\alpha^2 + \alpha^5 + \frac{1}{3}\alpha^6 + \cdots$, according to Brent's analysis, eventually approaching the order of $\sqrt{M}$; and the number of additional table positions probed while deciding how to make the insertion is about $\alpha^2 + \alpha^4 + \frac{4}{3}\alpha^5 + \alpha^6 + \cdots$.

Deletions. Many computer programmers have great faith in algorithms, and they are surprised to find that *the obvious way to delete records from a scatter table doesn't work*. For example, if we try to delete the key EN from Fig. 41, we can't simply mark that table position empty, because another key FEM would suddenly be forgotten! (Recall that EN and FEM both hashed to the same location. When looking up FEM, we would find an empty place, indicating an unsuccessful search.) A similar problem occurs with Algorithm C, due to the coalescing of lists; imagine the deletion of both TO and FIRE from Fig. 40.

In general, we can handle deletions by putting a special code value in the corresponding cell, so that there are three kinds of table entries: empty, occupied, and deleted. When searching for a key, we should skip over deleted cells, as if they were occupied. If the search is unsuccessful, the key can be inserted in place of the first deleted or empty position that was encountered.

But this idea is workable only when deletions are very rare, because the entries of the table never become empty again once they have been occupied. After a long sequence of repeated insertions and deletions, all of the empty spaces will eventually disappear, and every unsuccessful search will take M probes! Furthermore the time per probe will be increased, since we will have to test whether i has returned to its starting value in step D4; and the number of probes in a successful search will drift upward from C_N to C'_N.

When linear probing is being used (i.e., Algorithm L), it is possible to do deletions in a way that avoids such a sorry state of affairs, if we are willing to do some extra work for the deletion.

Algorithm R (*Deletion with linear probing*). Assuming that an open scatter table has been constructed by Algorithm L, this algorithm deletes the record from a given position TABLE[i].

R1. [Empty a cell.] Mark TABLE[i] empty, and set $j \leftarrow i$.

R2. [Decrease i.] Set $i \leftarrow i - 1$, and if this makes i negative set $i \leftarrow i + M$.

R3. [Inspect TABLE[i].] If TABLE[i] is empty, the algorithm terminates. Otherwise set $r \leftarrow h(\text{KEY}[i])$, the original hash address of the key now stored at position i. If $i \le r < j$ or if $r < j < i$ or $j < i \le r$ (in other words, if r lies cyclically between i and j), go back to R2.

R4. [Move a record.] Set TABLE[j] $\leftarrow$ TABLE[i], and return to step R1. ▮

Exercise 22 shows that this algorithm causes no degradation in performance, i.e., the average number of probes predicted in Eqs. (22) and (23) remains the same. (A similar result for tree insertion was proved in Theorem 6.2.2H.) But the validity of Algorithm R depends heavily on the fact that linear probing is involved, and no analogous deletion procedure for use with Algorithm D is possible.

Of course when chaining is used with separate lists for each possible hash value, deletion causes no problems since it is simply a deletion from a linked linear list. Deletion with Algorithm C is discussed in exercise 23.

***Analysis of the algorithms.** It is especially important to know the average behavior of a hashing method, because we are committed to trusting in the laws of probability whenever we hash. The worst case of these algorithms is almost unthinkably bad, so we need to be reassured that the average behavior is very good.

Before we get into the analysis of linear probing, etc., let us consider a very approximate model of the situation, which may be called *uniform hashing* (cf. W. W. Peterson, *IBM J. Research & Development* **1** (1957), 135–136). In this model, we assume that the keys go into random locations of the table, so that each of the $\binom{M}{N}$ possible configurations of N occupied cells and $M - N$ empty cells is equally likely. This model ignores any effect of primary or secondary clustering; the occupancy of each cell in the table is essentially independent of the others. For this model the probability that exactly r probes are needed to insert the $(N + 1)$st item is the number of configurations in which $r - 1$ given cells are occupied and another is empty, divided by $\binom{M}{N}$, namely

$$P_r = \binom{M - r}{N - r + 1} \Big/ \binom{M}{N};$$

therefore the average number of probes for uniform hashing is

$$C'_N = \sum_{1 \le r \le M} rP_r = M + 1 - \sum_{1 \le r \le M} (M + 1 - r)P_r$$

$$= M + 1 - \sum_{1 \le r \le M} (M + 1 - r) \binom{M - r}{M - N - 1} \Big/ \binom{M}{N}$$

$$= M + 1 - \sum_{1 \le r \le M} (M - N) \binom{M+1-r}{M-N} \Big/ \binom{M}{N} \tag{32}$$

$$= M + 1 - (M - N) \binom{M+1}{M-N+1} \Big/ \binom{M}{N}$$

$$= M + 1 - (M - N) \frac{M+1}{M-N+1} = \frac{M+1}{M-N+1}, \quad \text{for} \quad 1 \le N < M.$$

(We have already solved essentially the same problem in connection with random sampling, in exercise 3.4.2–5.) Setting $\alpha = N/M$, this exact formula for C'_N is approximately equal to

$$\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \cdots, \tag{33}$$

a series which has a rough intuitive interpretation: With probability α we need more than one probe, with probability α^2 we need more than two, etc. The corresponding average number of probes for a successful search is

$$\begin{aligned} C_N = \frac{1}{N} \sum_{0 \le k < N} C'_k &= \frac{M+1}{N} \left(\frac{1}{M+1} + \frac{1}{M} + \cdots + \frac{1}{M-N+2} \right) \\ &= \frac{M+1}{N} (H_{M+1} - H_{M-N+1}) \approx \frac{1}{\alpha} \log \frac{1}{1-\alpha}. \end{aligned} \tag{34}$$

As remarked above, extensive tests show that Algorithm D with two independent hash functions behaves essentially like uniform hashing, for all practical purposes.

This completes the analysis of uniform hashing. In order to study linear probing and other types of collision resolution, we need to set up the theory in a different, more realistic way. The probabilistic model we shall use for this purpose assumes that each of the M^N possible "hash sequences"

$$a_1 \, a_2 \ldots a_N, \qquad 0 \le a_j < M, \tag{35}$$

is equally likely, where a_j denotes the initial hash address of the jth key inserted into the table. The average number of probes in a successful search, given any particular searching algorithm, will be denoted by C_N as above; this is assumed to be the average number of probes needed to find the kth key, averaged over $1 \le k \le N$ with each key equally likely, and averaged over all hash sequences (35) with each sequence equally likely. Similarly, the average number of probes needed when the Nth key is inserted, considering all sequences (35) to be equally likely, will be denoted by C'_{N-1}; this is the average number of probes in an unsuccessful search starting with $N - 1$ elements in the table. When open addressing is used,

$$C_N = \frac{1}{N} \sum_{0 \le k < N} C'_k, \tag{36}$$

so that we can deduce one quantity from the other as we have done in (34).

Strictly speaking, there are two defects even in this more accurate model. In the first place, the different hash sequences aren't all equally probable, because the keys themselves are distinct. This makes the probability that $a_1 = a_2$ slightly less than $1/M$; but the difference is usually negligible since the set of all possible keys is typically very large compared to M. (See exercise 24.) Furthermore a good hash function will exploit the nonrandomness of typical data, making it even less likely that $a_1 = a_2$; as a result, our estimates for the number of probes will be pessimistic. Another inaccuracy in the model is indicated in Fig. 43: Keys that occur earlier are (with some exceptions) more likely to be looked up than keys that occur later. Therefore our estimate of C_N tends to be doubly pessimistic, and the algorithms should perform slightly better in practice than our analysis predicts.

With these precautions, we are ready to make an "exact" analysis of linear probing.* Let $f(M, N)$ be the number of hash sequences (35) such that position 0 of the table will be empty after the keys have been inserted by Algorithm L. The circular symmetry of linear probing implies that position 0 is empty just as often as any other position, so it is empty with probability $1 - N/M$; in other words

$$f(M, N) = \left(1 - \frac{N}{M}\right) M^N. \tag{37}$$

Now let $g(M, N, k)$ be the number of hash sequences (35) such that the algorithm leaves position 0 empty, positions 1 through k occupied, and position $k + 1$ empty. We have

$$g(M, N, k) = \binom{N}{k} f(k + 1, k)\ f(M - k - 1, N - k), \tag{38}$$

because all such hash sequences are composed of two subsequences, one (containing k elements $a_i \le k$) that leaves position 0 empty and positions 1 through k occupied and one (containing $N - k$ elements $a_j \ge k + 1$) that leaves position $k + 1$ empty; there are $f(k + 1, k)$ subsequences of the former type and $f(M - k - 1, N - k)$ of the latter, and there are $\binom{N}{k}$ ways to intersperse two such subsequences. Finally let P_k be the probability that exactly $k + 1$ probes will be needed when the $(N + 1)$st key is inserted; it follows (see exercise 25) that

$$P_k = M^{-N}(g(M, N, k) + g(M, N, k + 1) + \cdots + g(M, N, N)). \tag{39}$$

Now $C'_N = \sum_{0 \le k \le N} (k + 1)P_k$; putting this equation together with (36)–(39) and simplifying yields the following result.

---

* The author cannot resist inserting a biographical note at this point: I first formulated the following derivation in 1962, shortly after beginning work on *The Art of Computer Programming*. Since this was the first nontrivial algorithm I had ever analyzed satisfactorily, it has had a strong influence on the structure of these books. Little did I know that more than ten years would go by before this derivation got into print!

Theorem K. *The average number of probes needed by Algorithm L, assuming that all M^N hash sequences (35) are equally likely, is*

$$C_N = \tfrac{1}{2}(1 + Q_0(M, N-1)) \qquad \text{(successful search)}, \tag{40}$$

$$C'_N = \tfrac{1}{2}(1 + Q_1(M, N)) \qquad \text{(unsuccessful search)}, \tag{41}$$

where

$$Q_r(M, N) = \binom{r}{0} + \binom{r+1}{1}\frac{N}{M} + \binom{r+2}{2}\frac{N(N-1)}{M^2} + \cdots$$

$$= \sum_{k\ge 0}\binom{r+k}{k}\frac{N}{M}\,\frac{N-1}{M}\cdots\frac{N-k+1}{M}. \tag{42}$$

Proof. Details of the calculation are worked out in exercise 27. ∎

The rather strange-looking function $Q_r(M, N)$ which appears in this theorem is really not hard to deal with. We have

$$N^k - \binom{k}{2}N^{k-1} \le N(N-1)\cdots(N-k+1) \le N^k;$$

hence if $N/M = \alpha$,

$$\sum_{k\ge 0}\binom{r+k}{k}\left(N^k - \binom{k}{2}N^{k-1}\right)\Big/ M^k \le Q_r(M, N) \le \sum_{k\ge 0}\binom{r+k}{k}N^k/M^k,$$

$$\sum_{k\ge 0}\binom{r+k}{k}\alpha^k - \frac{\alpha}{M}\sum_{k\ge 0}\binom{r+k}{k}\binom{k}{2}\alpha^{k-2} \le Q_r(M, \alpha M) \le \sum_{k\ge 0}\binom{r+k}{k}\alpha^k,$$

i.e.,

$$\frac{1}{(1-\alpha)^{r+1}} - \frac{1}{M}\binom{r+2}{2}\frac{\alpha}{(1-\alpha)^{r+3}} \le Q_r(M, \alpha M) \le \frac{1}{(1-\alpha)^{r+1}}. \tag{43}$$

This relation gives us a good estimate of $Q_r(M, N)$ when M is large and α is not too close to 1. (The lower bound is a better approximation than the upper bound.) When α approaches 1, these formulas become useless, but fortunately $Q_0(M, M-1)$ is the function $Q(M)$ whose asymptotic behavior was studied in great detail in Section 1.2.11.3; and $Q_1(M, M-1)$ is simply equal to M (see exercise 50).

Another approach to the analysis of linear probing has been taken by G. Schay, Jr. and W. G. Spruth [*CACM* **5** (1962), 459–462]. Although their method yields only an approximation to the exact formulas in Theorem K, it sheds further light on the algorithm, so we shall sketch it briefly here. First let us consider a surprising property of linear probing which was first noticed by W. W. Peterson in 1957:

Theorem P. *The average number of probes in a successful search by Algorithm L is independent of the order in which the keys were inserted; it depends only on the number of keys which hash to each address.*

In other words, any rearrangement of a hash sequence $a_1\, a_2 \ldots a_N$ yields a hash sequence with the same average displacement of keys from their hash addresses. (We are assuming, as stated earlier, that all keys in the table have equal importance. If some keys are more frequently accessed than others, the proof can be extended to show that an optimal arrangement occurs if we insert them in decreasing order of frequency, by the method of Theorem 6.1S.)

Proof. It suffices to show that the total number of probes needed to insert keys for the hash sequence $a_1\, a_2 \ldots a_N$ is the same as the total number needed for $a_1 \ldots a_{i-1} a_{i+1} a_i a_{i+2} \ldots a_N$, $1 \le i < N$. There is clearly no difference unless the $(i+1)$st key in the second sequence falls into the position occupied by the ith in the first sequence. But then the ith and $(i+1)$st merely exchange places, so the number of probes for the $(i+1)$st is decreased by the same amount that the number for the ith is increased. ∎

Theorem P tells us that the average search length for a hash sequence $a_1\, a_2 \ldots a_N$ can be determined from the numbers $b_0\, b_1 \ldots b_{M-1}$, where b_j is the number of a's that equal j. From this sequence we can determine the "carry sequence" $c_0\, c_1 \ldots c_{M-1}$, where c_j is the number of keys for which both locations j and $j-1$ are probed as the key is inserted. This sequence is determined by the rule

$$c_j = \begin{cases} 0, & \text{if } \quad b_j = c_{(j+1) \bmod M} = 0; \\ b_j + c_{(j+1) \bmod M} - 1, & \text{otherwise.} \end{cases} \tag{44}$$

For example, let $M = 10$, $N = 8$, and $b_0 \ldots b_9 = 0\ 3\ 2\ 0\ 1\ 0\ 0\ 0\ 2$; then $c_0 \ldots c_9 = 2\ 3\ 1\ 0\ 0\ 0\ 0\ 1\ 2\ 3$, since one key needs to be "carried over" from position 2 to position 1, three from position 1 to position 0, two of these from position 0 to position 9, etc. We have $b_0 + b_1 + \cdots + b_{M-1} = N$, and the average number of probes needed for retrieval of the N keys is

$$1 + (c_0 + c_1 + \cdots + c_{M-1})/N. \tag{45}$$

Rule (44) seems to be a circular definition of the c's in terms of themselves, but actually there is a unique solution to the stated equations whenever $N < M$ (see exercise 32).

Schay and Spruth used this idea to determine the probability q_k that $c_j = k$, in terms of the probability p_k that $b_j = k$. (These probabilities are independent of j.) Thus

$$\begin{aligned} q_0 &= p_0 q_0 + p_1 q_0 + p_0 q_1, \\ q_1 &= p_2 q_0 + p_1 q_1 + p_0 q_2, \\ q_2 &= p_3 q_0 + p_2 q_1 + p_1 q_2 + p_0 q_3, \end{aligned} \tag{46}$$

etc., since, for example, the probability that $c_j = 2$ is the probability that $b_j + c_{(j+1) \bmod M} = 3$. Let $B(z) = \sum p_k z^k$ and $C(z) = \sum q_k z^k$ be the generating functions for these probability distributions; the equations (46) are equivalent to

$$B(z)C(z) = p_0 q_0 + (q_0 - p_0 q_0)z + q_1 z^2 + \cdots = p_0 q_0 (1 - z) + zC(z).$$

Since $B(1) = 1$, we may write $B(z) = 1 + (z - 1)D(z)$, and it follows that

$$C(z) = \frac{p_0 q_0}{1 - D(z)} = \frac{1 - D(1)}{1 - D(z)}, \tag{47}$$

since $C(1) = 1$. The average number of probes needed for retrieval, according to (45), will therefore be

$$1 + \frac{M}{N} C'(1) = 1 + \frac{M}{N} \frac{D'(1)}{1 - D(1)} = 1 + \frac{M}{2N} \frac{B''(1)}{1 - B'(1)}. \tag{48}$$

Since we are assuming that each hash sequence $a_1 \ldots a_N$ is equally likely we, have

$$\begin{aligned} p_k &= \Pr(\text{exactly } k \text{ of the } a_i \text{ are equal to } j, \text{ for fixed } j) \\ &= \binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}; \end{aligned} \tag{49}$$

hence

$$B(z) = \left(1 + \frac{z - 1}{M}\right)^N, \qquad B'(1) = \frac{N}{M}, \qquad B''(1) = \frac{N(N - 1)}{M^2}, \tag{50}$$

and the average number of probes according to (48) will be

$$C_N = \frac{1}{2}\left(1 + \frac{M - 1}{M - N}\right). \tag{51}$$

Can the reader see why this answer is different from the result in Theorem K? (Cf. exercise 33.)

***Optimality considerations.** We have seen several examples of probe sequences for open addressing, and it is natural to ask for one that can be proved "best possible" in some meaningful sense. This problem has been set up in the following interesting way by J. D. Ullman [*JACM* **19** (1972), 569–575]: Instead of computing a "hash address" $h(K)$, we map each key K into an entire permutation of $\{0, 1, \ldots, M - 1\}$, which represents the probe sequence to use for K. Each of the $M!$ permutations is assigned a probability, and the generalized hash function is supposed to select each permutation with that probability. The question is, "What assignment of probabilities to permutations gives the best performance, in the sense that the corresponding average number of probes C_N or C'_N is minimized?"

For example, if we assign the probability $1/M!$ to each permutation, it is easy to see that we have exactly the behavior of *uniform hashing* which we have analyzed above in (32), (34). However, Ullman has found an example with $M = 4$, $N = 2$ for which C'_N is smaller than the value $\frac{5}{3}$ obtained with uniform hashing. His construction assigns zero probability to all but the following six permutations:

Permutation	Probability	Permutation	Probability	
0 1 2 3	$(1 + 2\epsilon)/6$	1 0 3 2	$(1 + 2\epsilon)/6$	
2 0 1 3	$(1 - \epsilon)/6$	2 1 0 3	$(1 - \epsilon)/6$	(52)
3 0 1 2	$(1 - \epsilon)/6$	3 1 0 2	$(1 - \epsilon)/6$	

Roughly speaking, the first choice favors 2 and 3, but the second choice is 0 or 1. The average number of probes needed to insert the third item turns out to be $\frac{5}{3} - \frac{1}{9}\epsilon + O(\epsilon^2)$, so we can improve on uniform hashing by taking ϵ to be a small positive value.

However, the corresponding value of C'_1 for these probabilities is $\frac{23}{18} + O(\epsilon)$, which is larger than $\frac{5}{4}$ (the uniform hashing value). Ullman has proved that any assignment of probabilities such that $C'_N < (M + 1)/(M + 1 - N)$ for some N always implies that $C'_n > (M + 1)/(M + 1 - n)$ for some $n < N$; you can't win all the time over uniform hashing.

Actually the number of probes C_N for a *successful* search is a better measure than C'_N. The permutations in (52) do not lead to an improved value of C_N for any N, and indeed it seems reasonable to conjecture that no assignment of probabilities will be able to make C_N less than the uniform value $((M + 1)/N) \times (H_{M+1} - H_{M+1-N})$.

This conjecture appears to be very difficult to prove, especially because there are many ways to assign probabilities to achieve the effect of uniform hashing; we do not need to assign $1/M!$ to each permutation. For example, the following assignment for $M = 4$ is equivalent to uniform hashing:

Permutation	Probability	Permutation	Probability	
0 1 2 3	1/6	0 2 1 3	1/12	
1 2 3 0	1/6	1 3 2 0	1/12	(53)
2 3 0 1	1/6	2 0 3 1	1/12	
3 0 1 2	1/6	3 1 0 2	1/12	

with zero probability assigned to the other 16 permutations.

The following theorem characterizes *all* assignments that produce the behavior of uniform hashing.

Theorem U. *An assignment of probabilities to permutations will make each of the* $\binom{M}{N}$ *configurations of empty and occupied cells equally likely after* N *insertions, for* $0 < N < M$, *if and only if the sum of probabilities assigned to all permutations whose first* N *elements are the members of a given* N*-element set is* $1/\binom{M}{N}$, *for all* N *and all* N*-element sets.*

For example, the sum of probabilities assigned to each of the $3!(M - 3)!$ permutations beginning with the numbers $\{0, 1, 2\}$ in some order must be $1/\binom{M}{3} = 3!(M - 3)!/M!$. Observe that the condition of this theorem holds in (53).

Proof. Let $A \le \{0, 1, \ldots, M-1\}$, and let $\prod(A)$ be the set of all permutations whose first $\|A\|$ elements are members of A, and let $S(A)$ be the sum of the probabilities assigned to these permutations. Let $P_k(A)$ be the probability that the first $\|A\|$ insertions of the open addressing procedure occupy the locations specified by A, and that the last insertion required exactly k probes; and let $P(A) = P_1(A) + P_2(A) + \cdots$. The proof is by induction on $N \ge 1$, assuming that

$$P(A) = S(A) = 1 \Big/ \binom{M}{n}$$

for all sets A with $\|A\| = n < N$. Let B be any N-element set. Then

$$P_k(B) = \sum_{\substack{A \subseteq B \\ \|A\|=k}} \sum_{\pi \in \prod(A)} \Pr(\pi) P(B \backslash \{\pi_k\}),$$

where $\Pr(\pi)$ is the probability assigned to permutation π and π_k is its kth element. By induction

$$P_k(B) = \sum_{\substack{A \subseteq B \\ \|A\|=k}} \frac{1}{\binom{M}{N-1}} \sum_{\pi \in \prod(A)} \Pr(\pi),$$

which equals

$$\binom{N}{k} \Big/ \binom{M}{N-1}\binom{M}{k}, \qquad \text{if} \qquad k < N;$$

hence

$$P(B) = \frac{1}{\binom{M}{N-1}} \left(S(B) + \sum_{1 \le k < N} \frac{\binom{N}{k}}{\binom{M}{k}} \right),$$

and this can be equal to $1/\binom{M}{N}$ if and only if $S(B)$ has the correct value. ∎

External searching. Hashing techniques lend themselves well to external searching on direct-access storage devices like disks or drums. For such applications, as in Section 6.2.4, we want to minimize the number of accesses to the file, and this has two major effects on the choice of algorithms:

1) It is reasonable to spend more time computing the hash function, since the penalty for bad hashing is much greater than the cost of the extra time needed to do a careful job.
2) The records are usually grouped into *buckets*, so that several records are fetched from the external memory each time.

The file is usually divided into M buckets containing b records each. Collisions now cause no problem unless more than b keys have the same hash address. The following three approaches to collision resolution seem to be best:

A) *Chaining with separate lists.* If more than b records fall into the same bucket, a link to an "overflow" record can be inserted at the end of the first bucket. These overflow records are kept in a special overflow area. There is usually no advantage in having buckets in the overflow area, since comparatively few overflows occur; thus, the extra records are usually linked together so that the $(b + k)$th record of a list requires $1 + k$ accesses. It is usually a good idea to leave some room for overflows on each "cylinder" of a disk file, so that most accesses are to the same cylinder.

Although this method of handling overflows seems inefficient, the number of overflows is statistically small enough that the average search time is very good. See Tables 2 and 3, which show the average number of accesses required as a function of the load factor

$$\alpha = N/Mb, \tag{54}$$

for fixed α as $M, N \to \infty$. Curiously when $\alpha = 1$ the asymptotic number of accesses for an unsuccessful search increases with increasing b.

Table 2

AVERAGE ACCESSES IN AN UNSUCCESSFUL SEARCH BY SEPARATE CHAINING

Bucket size, b	Load factor, α 10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0048	1.0187	1.0408	1.0703	1.1065	1.1488	1.197	1.249	1.307	1.3
2	1.0012	1.0088	1.0269	1.0581	1.1036	1.1638	1.238	1.327	1.428	1.5
3	1.0003	1.0038	1.0162	1.0433	1.0898	1.1588	1.252	1.369	1.509	1.6
4	1.0001	1.0016	1.0095	1.0314	1.0751	1.1476	1.253	1.394	1.571	1.7
5	1.0000	1.0007	1.0056	1.0225	1.0619	1.1346	1.249	1.410	1.620	1.7
10	1.0000	1.0000	1.0004	1.0041	1.0222	1.0773	1.201	1.426	1.773	2.0
20	1.0000	1.0000	1.0000	1.0001	1.0028	1.0234	1.113	1.367	1.898	2.3
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0007	1.018	1.182	1.920	2.7

Table 3

AVERAGE ACCESSES IN A SUCCESSFUL SEARCH BY SEPARATE CHAINING

Bucket size, b	Load factor, α 10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0500	1.1000	1.1500	1.2000	1.2500	1.3000	1.350	1.400	1.450	1.5
2	1.0063	1.0242	1.0520	1.0883	1.1321	1.1823	1.238	1.299	1.364	1.4
3	1.0010	1.0071	1.0216	1.0458	1.0806	1.1259	1.181	1.246	1.319	1.4
4	1.0002	1.0023	1.0097	1.0257	1.0527	1.0922	1.145	1.211	1.290	1.3
5	1.0000	1.0008	1.0046	1.0151	1.0358	1.0699	1.119	1.186	1.286	1.3
10	1.0000	1.0000	1.0002	1.0015	1.0070	1.0226	1.056	1.115	1.206	1.3
20	1.0000	1.0000	1.0000	1.0000	1.0005	1.0038	1.018	1.059	1.150	1.2
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.001	1.015	1.083	1.2

B) *Chaining with coalescing lists.* Instead of providing a separate overflow area, we can adapt Algorithm C to external files. A doubly linked list of available space can be used which links together each bucket that is not yet full. Under this scheme, every bucket contains a count of how many record positions are empty, and the bucket is removed from the doubly linked list only when this count becomes zero. A 'roving pointer' can be used to distribute overflows (cf. exercise 2.5–6), so that different lists tend to use different overflow buckets. It may be a good idea to have separate available-space lists for the buckets on each cylinder of a disk file.

This method has not yet been analyzed, but it should prove to be quite useful.

C) *Open addressing.* We can also do without links, using an "open" method. Linear probing is probably better than random probing when we consider external searching, because the increment c can often be chosen so that it minimizes latency delays between consecutive accesses. The approximate theoretical model of linear probing which was worked out above can be generalized to account for the influence of buckets, and it shows that linear probing is indeed satisfactory unless the table has gotten very full. For example, see Table 4; when the load factor is 90 percent and the bucket size is 50, the average

Table 4

AVERAGE ACCESSES IN A SUCCESSFUL SEARCH BY LINEAR PROBING

Bucket size, b	Load factor, α 10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0556	1.1250	1.2143	1.3333	1.5000	1.7500	2.167	3.000	5.500	10.5
2	1.0062	1.0242	1.0553	1.1033	1.1767	1.2930	1.494	1.903	3.147	5.6
3	1.0009	1.0066	1.0201	1.0450	1.0872	1.1584	1.286	1.554	2.378	4.0
4	1.0001	1.0021	1.0085	1.0227	1.0497	1.0984	1.190	1.386	2.000	3.2
5	1.0000	1.0007	1.0039	1.0124	1.0307	1.0661	1.136	1.289	1.777	2.7
10	1.0000	1.0000	1.0001	1.0011	1.0047	1.0154	1.042	1.110	1.345	1.8
20	1.0000	1.0000	1.0000	1.0000	1.0003	1.0020	1.010	1.036	1.144	1.4
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.001	1.005	1.040	1.1

number of accesses in a successful search is only 1.04. This is actually *better* than the 1.08 accesses required by the chaining method (A) with the same bucket size!

The analysis of methods (A) and (C) involves some very interesting mathematics; we shall merely summarize the results here, since the details are worked out in exercises 49 and 55. The formulas involve two functions strongly related to the Q-functions of Theorem K, namely

$$R(\alpha, n) = \frac{n}{n+1} + \frac{n^2\alpha}{(n+1)(n+2)} + \frac{n^3\alpha^2}{(n+1)(n+2)(n+3)} + \cdots, \quad (55)$$

and

$$t_n(\alpha) = e^{-n\alpha}\left(\frac{(\alpha n)^n}{(n+1)!} + 2\,\frac{(\alpha n)^{n+1}}{(n+2)!} + 3\,\frac{(\alpha n)^{n+2}}{(n+3)!} + \cdots\right)$$

$$= \frac{e^{-n\alpha}n^n\alpha^n}{n!}\,(1 - (1-\alpha)R(\alpha, n)). \tag{56}$$

In terms of these functions, the average number of accesses made by the chaining method (A) in an unsuccessful search is

$$C'_N = 1 + \alpha b t_b(\alpha) + O\left(\frac{1}{M}\right) \tag{57}$$

as $M, N \to \infty$, and the corresponding number in a successful search is

$$C_N = 1 + (1 - \tfrac{1}{2}b(1-\alpha))t_b(\alpha) + \frac{e^{-b\alpha}b^b\alpha^b}{2b!}\,R(\alpha, b) + O\left(\frac{1}{M}\right). \tag{58}$$

These are the quantities shown in Tables 2 and 3.

Since chaining method (A) requires a separate overflow area, we need to estimate how many overflows will occur. The average number of overflows will be $M(C'_N - 1) = Nt_b(\alpha)$, since $C'_N - 1$ is the average number of overflows in any given list. Therefore Table 2 can be used to deduce the amount of overflow space required. For fixed α, the standard deviation of the total number of overflows will be roughly proportional to $\sqrt{M}$ as $M \to \infty$.

Asymptotic values for C'_N and C_N appear in exercise 53, but the approximations aren't very good when b is small or α is large; fortunately the series for $R(\alpha, n)$ converges rather rapidly even when α is large, so the formulas can be evaluated exactly without much difficulty. The maximum values occur for $\alpha = 1$, when

$$\max C'_N = 1 + \frac{e^{-b}b^{b+1}}{b!} = \sqrt{\frac{b}{2\pi}} + 1 + O(b^{-1/2}), \tag{59}$$

$$\max C_N = 1 + \frac{e^{-b}b^b}{2b!}\,(R(b)+1) = \frac{5}{4} + \sqrt{\frac{2}{9\pi b}} + O(b^{-1}), \tag{60}$$

as $b \to \infty$, by Stirling's approximation and the analysis of the function $R(n) = R(1, n) - 1$ in Section 1.2.11.3.

The average number of access in a successful external search with *linear* probing has the remarkably simple expression

$$C_N \approx 1 + t_b(\alpha) + t_{2b}(\alpha) + t_{3b}(\alpha) + \cdots, \tag{61}$$

which can be understood as follows: The average total number of accesses to look up all N keys is NC_N, and this is $N + T_1 + T_2 + \cdots$, where T_k is the average number of keys which require more than k accesses. Theorem P says that we can enter the keys in any order without affecting C_N, and it follows that T_k is the average number of overflow records that would occur in the

chaining method if we had M/k buckets of size kb, namely $Nt_{kb}(\alpha)$ by what we said above. Further justification of Eq. (61) appears in exercise 55.

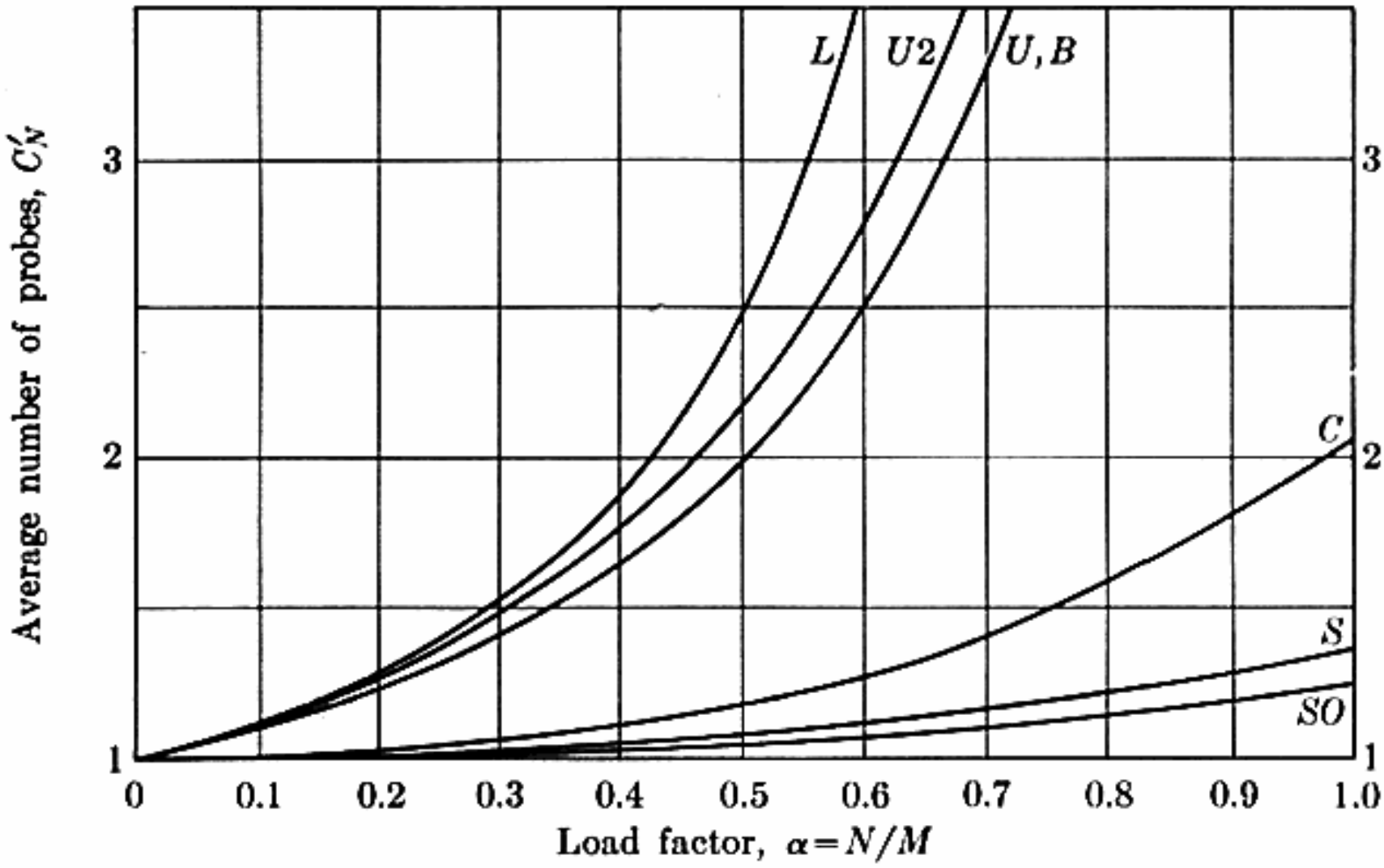
An excellent discussion of practical considerations involved in the design of external scatter tables has been given by Charles A. Olson, *Proc. ACM Nat'l Conf.* **24** (1969), 539–549. He includes several worked examples and points out that the number of overflow records will increase substantially if the file is subject to frequent insertion/deletion activity without relocating records; and he presents an analysis of this situation which was obtained jointly with J. A. de Peyster.

Comparison of the methods. We have now studied a large number of techniques for searching; how can we select the right one for a given application? It is difficult to summarize in a few words all the relevant details of the "trade-offs" involved in the choice of a search method, but the following things seem to be of primary importance with respect to the speed of searching and the requisite storage space.

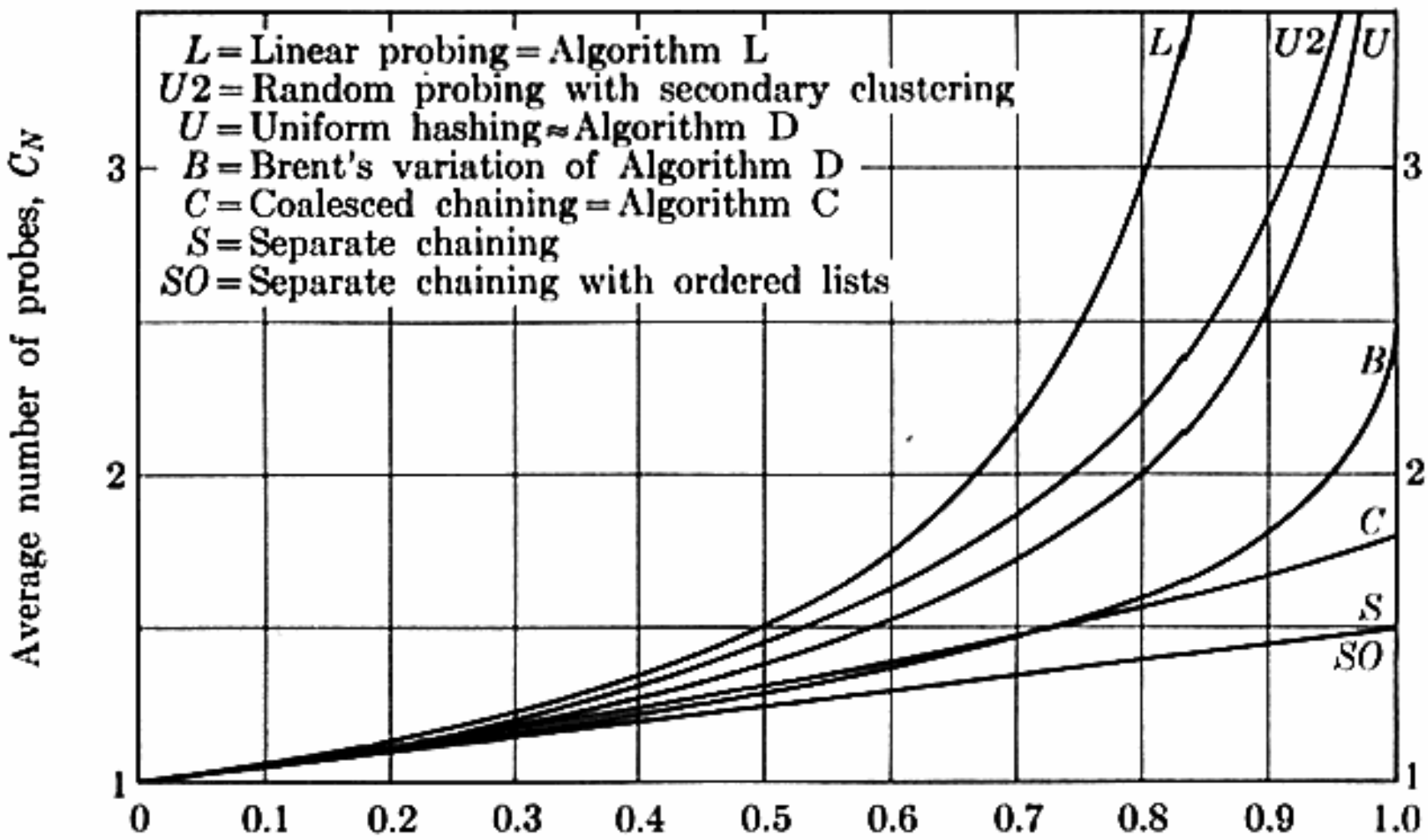
Figure 44 summarizes the analyses of this section, showing that the various methods for collision resolution lead to different numbers of probes. But this does not tell the whole story, since the time per probe varies in different methods, and the latter variation has a noticeable effect on the running time (as we have seen in Fig. 42). Linear probing accesses the table more frequently than the other methods shown in Fig. 44, but it has the advantage of simplicity. Furthermore, even linear probing isn't terribly bad: when the table is 90 percent full, Algorithm L requires an average of less than 5.5 probes to locate a random item in the table. (However, the average number of probes needed to insert a *new* item with Algorithm L is 50.5, with a 90-percent-full table.)

Figure 44 shows that the chaining methods are quite economical with respect to the number of probes, but the extra memory space needed for link fields sometimes makes open addressing more attractive for small records. For example, if we have to choose between a chained scatter table of capacity 500 and an open scatter table of capacity 1000, the latter is clearly preferable, since it allows efficient searching when 500 records are present and it is capable of absorbing twice as much data. On the other hand, sometimes the record size and format will allow space for link fields at virtually no extra cost. (Cf. exercise 65.)

How do hash methods compare with the other search strategies we have studied in this chapter? From the standpoint of speed we can argue that they are better, when the number of records is large, because the average search time for a hash method stays bounded as $N \longrightarrow \infty$ if we stipulate that the table never gets too full. For example, Program L will take only about 55 units of time for a successful search when the table is 90 percent full; this beats the fastest MIX binary search routine we have seen (exercise 6.2.1–24) when N is greater than 600 or so, at the cost of only 11 percent in storage space. Moreover the binary search is suitable only for fixed tables, while a scatter table allows efficient insertions.



(a) Unsuccessful search.



(b) Successful search

Fig. 44. Comparison of collision resolution methods: Limiting values of the average number of probes as $M \to \infty$.

We can also compare Program L to the tree-oriented search methods which allow dynamic insertions. Program L with a 90-percent-full table is faster than Program 6.2.2T when N is greater than about 90, and faster than Program 6.3D (exercise 6.3–9) when N is greater than about 75.

Only one search method in this chapter is efficient for successful searching with virtually no storage overhead, namely Brent's variation of Algorithm D. His method allows us to put N records into a table of size $M = N + 1$, and to

find any record in about $2\frac{1}{2}$ probes on the average. No extra space for link fields, etc., is needed; however, an unsuccessful search will be very slow, requiring about $\frac{1}{2}N$ probes.

Thus hashing has several advantages. On the other hand, there are three important respects in which scatter table searching is inferior to other methods we have discussed:

a) After an unsuccessful search in a scatter table, we know only that the desired key is not present. Search methods based on comparisons always yield more information, making it possible to find the largest key $\leq K$ and/or the smallest key $\geq K$; this is important in many applications (e.g., for interpolation of function values from a stored table). It is also possible to use comparison-based algorithms to locate all keys which lie *between* two given values K and K'. Furthermore the tree search algorithms of Section 6.2 make it easy to output the contents of a table in ascending order, without sorting it separately, and this is occasionally desirable.

b) The storage allocation for scatter tables is often somewhat difficult; we have to dedicate a certain area of the memory for use as the hash table, and it may not be obvious how much space should be allotted. If we provide too much memory, we may be wasting storage at the expense of other lists or other computer users; but if we don't provide enough room, the table will overflow. When a scatter table overflows, it is probably best to "rehash" it, i.e., to allocate a larger space and to change the hash function, reinserting every record into the larger table. F. R. A. Hopgood [*Comp. Bulletin* **11** (1968), 297–300] has suggested rehashing the table when it becomes α_0 percent full, replacing M by d_0M; suitable choices of these parameters α_0 and d_0 can be made by using the analyses above and characteristics of the data, so that the critical point at which it becomes cheaper to rehash can be determined. (Note that the method of chaining does not lead to any troublesome overflows, so it requires no rehashing; but the search time is proportional to N when M is fixed and N gets large.) By contrast, the tree search and insertion algorithms require no such painful rehashing; the trees grow no larger than necessary. In a virtual memory environment we probably ought to use tree search or digital tree search, instead of creating a large scatter table that requires bringing in a new page nearly every time we hash a key.

c) Finally, we need a great deal of faith in probability theory when we use hashing methods, since they are efficient only on the average, while their worst case is terrible! As in the case of random number generators, we are never completely sure that a hash function will perform properly when it is applied to a new set of data. Therefore scatter storage would be inappropriate for certain real-time applications such as air traffic control, where people's lives are at stake; the balanced tree algorithms of Sections 6.2.3 and 6.2.4 are much safer, since they provide guaranteed upper bounds on the search time.

History. The idea of hashing appears to have been originated by H. P. Luhn, who wrote an internal IBM memorandum suggesting the use of chaining, in

January 1953; this was also among the first applications of linked linear lists. He pointed out the desirability of using buckets containing more than one element, for external searching. Shortly afterwards, A. D. Lin carried Luhn's analysis a little further, and suggested a technique for handling overflows that used "degenerative addresses"; e.g., the overflows from primary bucket 2748 are put in secondary bucket 274; overflows from that bucket go to tertiary bucket 27, etc., assuming the presence of 10000 primary buckets, 1000 secondary buckets, 100 tertiary buckets, etc. The hash functions originally suggested by Luhn were digital in character, e.g., adding adjacent pairs of key digits mod 10, so that 31415926 would be compressed to 4548.

At about the same time the idea of hashing occurred independently to another group of IBMers: Gene M. Amdahl, Elaine M. Boehme, N. Rochester, and Arthur L. Samuel, who were building an assembly program for the IBM 701. In order to handle the collision problem, Amdahl originated the idea of open addressing with linear probing.

Hash coding was first described in the open literature by Arnold I. Dumey, *Computers and Automation* **5**, 12 (December 1956), 6–9. He was the first to mention the idea of dividing by a prime number and using the remainder as the hash address. Dumey's interesting article mentions chaining but not open addressing. A. P. Ershov of Russia independently discovered linear open addressing in 1957 [*Doklady Akad. Nauk SSSR* **118** (1958), 427–430]; he published empirical results about the number of probes, conjecturing correctly that the average number of probes per successful search is <2 when $N/M < 2/3$.

A classic article by W. W. Peterson, *IBM J. Research & Development* **1** (1957), 130–146, was the first major paper dealing with the problem of searching in large files. Peterson defined open addressing in general, analyzed the performance of uniform hashing, and gave numerous empirical statistics about the behavior of linear open addressing with various bucket sizes, noting the degradation in performance that occurred when items were deleted. Another comprehensive survey of the subject was published six years later by Werner Buchholz [*IBM Systems J.* **2** (1963), 86–111], who gave an especially good discussion of hash functions.

Up to this time linear probing was the only type of open addressing scheme that had appeared in the literature, but another scheme based on repeated random probing by independent hash functions had independently been developed by several people (see exercise 48). During the next few years hashing became very widely used, but hardly anything more was published about it. Then Robert Morris wrote a very influential survey of the subject [*CACM* **11** (1968), 38–44], in which he introduced the idea of random probing (with secondary clustering). Morris's paper touched off a flurry of activity which culminated in Algorithm D and its refinements.

It is interesting to note that the word "hashing" apparently never appeared in print, with its present meaning, until Morris's article was published in 1968, although it had already become common jargon in several parts of the world

by that time. The only previous occurrence of the word among approximately 60 relevant documents studied by the author as this section was being written was in an unpublished memorandum written by W. W. Peterson in 1961. Somehow the verb "to hash" magically became standard terminology for key transformation during the mid-1960's, yet nobody was rash enough to use such an undignified word publicly until 1968!

EXERCISES

1. [*20*] When the instruction 9H in Table 1 is reached, how small and how large can the contents of rI1 possibly be, assuming that bytes 1, 2, 3 of K contain alphabetic character codes less than 30?

2. [*20*] Find a reasonably common English word not in Table 1 that could be added to that table without changing the program.

3. [*23*] Explain why no program beginning with the five instructions

```
LD1  K(1:1)     or     LD1N  K(1:1)
LD2  K(2:2)     or     LD2N  K(2:2)
          INC1  a,2
          LD2   K(3:3)
          J2Z   9F
```

could be used in place of the more complicated program in Table 1, for any constant a, since unique addresses would not be produced for the given keys.

4. [*M30*] How many people should be invited to a party in order to make it likely that there are *three* with the same birthday?

5. [*15*] Mr. B. C. Dull was writing a FORTRAN compiler using a decimal MIX computer, and he needed a symbol table to keep track of the names of variables in the FORTRAN program being compiled. These names were restricted to be ten or less characters in length. He decided to use a scatter table with $M = 100$, and to use the fast hash function $h(K) =$ leftmost byte of K. Was this a good idea?

6. [*15*] Would it be wise to change the first two instructions of (3) to `LDA K; ENTX 0`?

7. [*HM30*] (*Polynomial hashing.*) The purpose of this exercise is to consider the construction of polynomials $P(x)$ such as (10), which convert n-bit keys into m-bit addresses, such that distinct keys differing in t or fewer bits will hash to different addresses. Given n and $t \le n$, and given an integer k such that n divides $2^k - 1$, we shall construct a polynomial whose degree m is a function of n, t, and k. (Usually n is increased, if necessary, so that k can be chosen to be reasonably small.)

Let S be the smallest set of integers such that $\{1, 2, \ldots, t\} \subseteq S$, and $(2j) \bmod n \in S$ for all $j \in S$. For example, when $n = 15$, $k = 4$, $t = 6$, we have $S = \{1, 2, 3, 4, 5, 6, 8, 10, 12, 9\}$. We now define the polynomial $P(x) = \prod_{j \in S}(x - \alpha^j)$, where α is an element of order n in the finite field $GF(2^k)$, and where the coefficients of $P(x)$ are computed in this field. The degree m of $P(x)$ is the number of elements of S. Since α^{2j} is a root of $P(x)$ whenever α^j is a root, it follows that the coefficients p_i of $P(x)$ satisfy $p_i^2 = p_i$, so they are all 0 or 1.

Prove that if $R(x) = r_{n-1}x^{n-1} + \cdots + r_1x + r_0$ is any nonzero polynomial modulo 2, with at most t nonzero coefficients, then $R(x)$ is not a multiple of $P(x)$, modulo 2. [It follows that the corresponding hash function behaves as advertised.]

8. [*M34*] (*The three-distance theorem.*) Let θ be an irrational number between 0 and 1, whose regular continued fraction representation in the notation of Section 4.5.3 is $\theta = /a_1, a_2, a_3, \ldots/$. Let $q_0 = 0$, $p_0 = 1$, $q_1 = 1$, $p_1 = 0$, and $q_{k+1} = a_kq_k + q_{k-1}$, $p_{k+1} = a_kp_k + p_{k-1}$, for $k \geq 1$. Let $\{x\}$ denote $x \bmod 1 = x - \lfloor x \rfloor$, and let $\{x\}^+$ denote $x - \lceil x \rceil + 1$. As the points $\{\theta\}$, $\{2\theta\}$, $\{3\theta\}$, ... are successively inserted into the interval [0, 1], let the line segments be numbered as they appear in such a way that the first segment of a given length is number 0, the next is number 1, etc. Prove that the following statements are all true: Interval number s of length $\{t\theta\}$, where $t = rq_k + q_{k-1}$ and $0 \leq r < a_k$ and k is even and $0 \leq s < q_k$, has left endpoint $\{s\theta\}$ and right endpoint $\{(s+t)\theta\}^+$. Interval number s of length $1 - \{t\theta\}$, where $t = rq_k + q_{k-1}$ and $0 \leq r < a_k$ and k is odd and $0 \leq s < q_k$, has left endpoint $\{(s+t)\theta\}$ and right endpoint $\{s\theta\}^+$. Every positive integer n can be uniquely represented as $n = rq_k + q_{k-1} + s$ for some $k \geq 1$, $1 \leq r \leq a_k$, $0 \leq s < q_k$. In terms of this representation, just before the point $\{n\theta\}$ is inserted the n intervals present are

the first s intervals (numbered $0, \ldots, s-1$) of length $\{(-1)^k(rq_k + q_{k-1})\theta\}$;
the first $n - q_k$ intervals (numbered $0, \ldots, n - q_k - 1$) of length $\{(-1)^kq_k\theta\}$;
the last $q_k - s$ intervals (numbered $s, \ldots, q_k - 1$) of length $\{(-1)^k((r-1)q_k + q_{k-1})\theta\}$.

The operation of inserting $\{n\theta\}$ removes interval number s of the latter type and converts it into interval number s of the first type, number $n - q_k$ of the second type.

9. [*M30*] When we successively insert the points $\{\theta\}$, $\{2\theta\}$, ... into the interval [0, 1], Theorem S asserts that each new point always breaks up one of the largest remaining intervals. If the interval $[a, c]$ is thereby broken into two parts $[a, b]$, $[b, c]$, we may call it a *bad break* if one of these parts is more than twice as long as the other, i.e. if $b - a > 2(c - b)$ or $c - b > 2(b - a)$.

Prove that bad breaks will occur for some $\{n\theta\}$ unless $\theta \bmod 1 = \phi^{-1}$ or ϕ^{-2}; and the latter values of θ *never* produce bad breaks.

10. [*M48*] (R. L. Graham.) Prove or disprove the following $3d$ *distance conjecture:* If $\theta, \alpha_1, \ldots, \alpha_d$ are real numbers with $\alpha_1 = 0$, and if $n_1, \ldots, n_d$ are positive integers, and if the points $\{n\theta + \alpha_i\}$ are inserted into the interval [0, 1] for $0 \leq n < n_i$, $1 \leq i \leq d$, the resulting $n_1 + \cdots + n_d$ (possibly empty) intervals have at most $3d$ different lengths.

11. [*16*] Successful searches are usually more frequent than unsuccessful ones. Would it therefore be a good idea to interchange lines 12–13 of Program C with lines 10–11?

▸ **12.** [*21*] Show that Program C can be rewritten so that there is only one conditional jump instruction in the inner loop. Compare the running time of the modified program with the original.

▸ **13.** [*24*] (*Abbreviated keys.*) Let $h(K)$ be a hash function, and let $q(K)$ be a function of K such that K can be determined once $h(K)$ and $q(K)$ are given. For example, in division hashing we may let $h(K) = K \bmod M$ and $q(K) = \lfloor K/M \rfloor$; in multiplicative hashing we may let $h(K)$ be the leading bits of $(AK/w) \bmod 1$, and $q(K)$ can be the other bits.

Show that when chaining is used without overlapping lists, we need only store $q(K)$ instead of K in each record. (This almost saves the space needed for the link fields.) Modify Algorithm C so that it allows such abbreviated keys by avoiding overlapping lists, yet uses no auxiliary storage locations for "overflow" records.

14. [*24*] (E. W. Elcock.) Show that it is possible to let a large scatter table *share memory* with any number of other linked lists. Let every word of the list area have a 2-bit TAG field, with the following interpretation:

TAG(P) = 0 indicates a word in the list of available space; LINK(P) points to the next available word.

TAG(P) = 1 indicates any word in use that is not part of the scatter table; the other fields of this word may have any desired format.

TAG(P) = 2 indicates a word in the scatter table; LINK(P) points to another word. Whenever we are processing a list that is not part of the scatter table and we access a word with TAG(P) = 2, we are supposed to set P ← LINK(P) until reaching a word with TAG(P) ≤ 1. (For efficiency we might also then change one of the prior links so that it will not be necessary to skip over the same scatter table entries again and again.)

Show how to define suitable algorithms for inserting and retrieving keys from such a combined table, assuming that words with TAG(P) = 2 also have another link field AUX(P).

15. [*16*] Why is it a good idea for Algorithm L and Algorithm D to signal overflow when $N = M - 1$ instead of when $N = M$?

16. [*10*] Program L says that K should not be zero. But doesn't it actually work even when K is zero?

17. [*15*] Why not simply define $h_2(K) = h_1(K)$ in (25), when $h_1(K) \neq 0$?

▸ **18.** [*21*] Is (31) better or worse than (30), as a substitute for lines 10–13 of Program D? Give your answer on the basis of the average values of A, $S1$, and C.

19. [*40*] Empirically test the effect of restricting the range of $h_2(K)$ in Algorithm D, so that (a) $1 \leq h_2(K) \leq r$ for $r = 1, 2, 3, \ldots, 10$; (b) $1 \leq h_2(K) \leq \rho M$ for $\rho = \frac{1}{10}, \frac{2}{10}, \ldots, \frac{9}{10}$.

20. [*M25*] (R. Krutar.) Change Algorithm D as follows, avoiding the hash function $h_2(K)$: In step D3, set $c \leftarrow 0$; and at the beginning of step D4, set $c \leftarrow c + 1$. Prove that if $M = 2^m$, the corresponding probe sequence $h_1(K)$, $(h_1(K) - 1) \bmod M, \ldots, (h_1(K) - \binom{M}{2}) \bmod M$ will be a permutation of $\{0, 1, \ldots, M - 1\}$. When this method is programmed for MIX, how does it compare with the three programs considered in Fig. 42, assuming that the behavior is like random probing with secondary clustering?

▸ **21.** [*20*] Suppose that we wish to delete a record from a table constructed by Algorithm D, marking it "deleted" as suggested in the text. Should we also decrease the variable N which is used to govern Algorithm D?

22. [*27*] Prove that Algorithm R leaves the table exactly as it would have been if KEY[i] had never been inserted in the first place.

▸ **23.** [*23*] Design an algorithm analogous to Algorithm R, for deleting entries from a chained scatter table that has been constructed by Algorithm C.

24. [*M20*] Suppose that the set of all possible keys that can occur has MP elements, where exactly P keys hash to a given address. (In practical cases, P is very large; e.g. if the keys are arbitrary 10-digit numbers and if $M = 10^3$, we have $P = 10^7$.) Assume that $M \geq 7$ and $N = 7$. If seven distinct keys are selected at random from the set of all possible keys, what is the exact probability that the hash sequence 1 2 6 2 1 6 1 will be obtained (i.e., that $h(K_1) = 1, h(K_2) = 2, \ldots, h(K_7) = 1$), as a function of M and P?

25. [*M19*] Explain why Eq. (39) is true.

26. [*M20*] How many hash sequences $a_1\, a_2 \ldots a_9$ yield the pattern of occupied cells (21), using linear probing?

27. [*M27*] Complete the proof of Theorem K. [*Hint:* Let

$$s(n, x, y) = \sum_{k \geq 0} \binom{n}{k} (x + k)^{k+1} (y - k)^{n-k-1} (y - n);$$

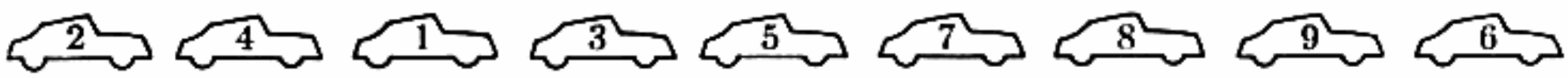
use Abel's binomial theorem, Eq. 1.2.6–16, to prove that $s(n, x, y) = x(x + y)^n + ns(n - 1, x + 1, y - 1)$.]

28. [*M30*] In the old days when computers were much slower than they are now, it was possible to watch the lights flashing and see how fast Algorithm L was running. When the table began to fill up, some entries would be processed very quickly, while others took a great deal of time.

This experience suggests that the standard deviation of C'_N is rather high, when linear probing is used. Find a formula which expresses the variance in terms of the Q_r functions defined in Theorem K, and estimate the variance when $N = \alpha M$ as $M \to \infty$.

29. [*M21*] (*The parking problem.*) A certain one-way street has m parking spaces in a row, numbered 1 through m. A man and his dozing wife drive by, and suddenly she wakes up and orders him to park immediately. He dutifully parks at the first available space; but if there are no places left that he can get to without backing up (i.e., if his wife awoke when the car approached space k, but spaces $k, k + 1, \ldots, m$ are all full), he expresses his regrets and drives on.

Suppose, in fact, that this happens for n different cars, where the jth wife wakes up just in time to park at space a_j. In how many of the sequences $a_1 \ldots a_n$ will all of the cars get safely parked, assuming that the street is initially empty and that nobody leaves after parking? For example, when $m = n = 9$ and $a_1 \ldots a_9 =$ 3 1 4 1 5 9 2 6 5, the cars get parked as follows:



[*Hint:* Use the analysis of linear probing.]

30. [*M28*] (John Riordan.) When $n = m$ in the parking problem of exercise 29, show that all cars get parked if and only if there exists a permutation $p_1\, p_2 \ldots p_n$ of $\{1, 2, \ldots, n\}$ such that $a_j \leq p_j$ for all j.

31. [*M40*] When $n = m$ in the parking problem of exercise 29, the number of solutions turns out to be $(n + 1)^{n-1}$; and from exercise 2.3.4.4–22 we know this is the same as the number of free trees on $n + 1$ labeled vertices! Find an interesting connection between parking sequences and trees.

32. [*M26*] Prove that the system of equations (44) has a unique solution $(c_0, c_1, \ldots, c_{M-1})$, whenever $b_0, b_1, \ldots, b_{M-1}$ are nonnegative integers whose sum is less than M. Design an algorithm which finds that solution.

▸**33.** [*M23*] Explain why (51) is only an approximation to the true average number of probes made by Algorithm L. [What was there about the derivation of (51) that wasn't rigorously exact?]

▸**34.** [*M22*] The purpose of this exercise is to investigate the average number of probes in a chained scatter table when the lists are kept separate as in Fig. 38. (a) What is P_{Nk}, the probability that a given list has length k, when the M^N hash sequences (35) are equally likely? (b) Find the generating function $P_N(z) = \sum_{k\geq 0} P_{Nk} z^k$. (c) Express the average number of probes for successful and unsuccessful search in terms of this generating function. (Assume that an unsuccessful search in a list of length k requires $k + \delta_{k0}$ probes.)

35. [*M21*] Continuing exercise 34, what is the average number of probes in an unsuccessful search when the individual lists are kept in order by their key values?

36. [*M22*] Find the variance of (18), the number of probes in separate chaining when the search is unsuccessful.

▸**37.** [*M29*] Find the variance of (19), the number of probes in separate chaining when the search is successful.

38. [*M32*] (*Tree hashing.*) A clever programmer might try to use binary search trees instead of linear lists in the chaining method, thereby combining Algorithm 6.2.2T with hashing. Analyze the average number of probes that would be required by this compound algorithm, for both successful and unsuccessful searches. [*Hint:* Cf. Eq. 5.2.1–11.]

39. [*M30*] The purpose of this exercise is to analyze the average number of probes in Algorithm C (chaining with coalescing lists). Let $c(k_1, k_2, k_3, \ldots)$ be the number of hash sequences (35) that cause Algorithm C to form exactly k_1 lists of length 1, k_2 of length 2, etc., when $k_1 + 2k_2 + 3k_3 + \cdots = N$. Find a recurrence relation which defines these numbers $c(k_1, k_2, k_3, \ldots)$, and use it to determine a simple formula for the sum

$$S_N = \sum_{\substack{j\geq 1 \\ k_1+2k_2+\cdots=N}} \binom{j}{2} k_j c(k_1, k_2, \ldots).$$

How is S_N related to the number of probes in an unsuccessful search by Algorithm C?

40. [*M33*] Find the variance of (15), the number of probes used by Algorithm C in an unsuccessful search.

41. [*M40*] Analyze T_N, the average number of times R is decreased by 1 when the $(N + 1)$st item is being inserted by Algorithm C.

▸**42.** [*M20*] Derive (17).

43. [*M42*] Analyze a modification of Algorithm C that uses a table of size $M' > M$. Only the first M locations are used for hasing, so the first $M' - M$ empty nodes found in step C5 will be in the extra locations of the table. For fixed M', what choice of M in the range $1 \leq M \leq M'$ leads to the best performance?

44. [*M43*] (*Random probing with secondary clustering.*) The object of this exercise is to determine the expected number of probes in the open addressing scheme with probe sequence

$$h(K),\quad (h(K)+p_1) \bmod M,\quad (h(K)+p_2) \bmod M,\quad \ldots,\quad (h(K)+p_{M-1}) \bmod M,$$

where $p_1\, p_2 \ldots p_{M-1}$ is a randomly chosen permutation of $\{1, 2, \ldots, M-1\}$ that depends on $h(K)$. In other words, all keys with the same value of $h(K)$ follow the same probe sequence, and the $(M-1)!^M$ possible choices of M probe sequences with this property are equally likely.

This situation can be accurately modeled by the following experimental procedure performed on an initially empty linear array of size m. Do the following operation n times:

> With probability p, occupy the leftmost empty position. Otherwise (i.e., with probability $q = 1 - p$), select any table position except the leftmost, with each of these $m - 1$ positions equally likely. If the selected position is empty, occupy it; otherwise select *any* empty position (including the leftmost) and occupy it, considering each of the empty positions equally likely.

For example when $m = 5$ and $n = 3$, the array configuration after the above experiment will be (occupied, occupied, empty, occupied, empty) with probability

$$\tfrac{7}{192}qqq + \tfrac{1}{6}pqq + \tfrac{1}{6}qpq + \tfrac{11}{64}qqp + \tfrac{1}{3}ppq + \tfrac{1}{4}pqp + \tfrac{1}{4}qpp.$$

(This procedure corresponds to random probing with secondary clustering, when $p = 1/m$, since we can renumber the table entries so that a particular probe sequence is 0, 1, 2, . . . and all the others are random.)

Find a formula for the average number of occupied positions at the left of the array (i.e., 2 in the above example). Also find the asymptotic value of this quantity when $p = 1/m$, $n = \alpha(m+1)$, and $m \to \infty$.

45. [*M48*] Solve the analog of exercise 44 with *tertiary clustering*, when the probe sequence begins $h_1(K)$, $((h_1(K) + h_2(K)) \bmod M$, and the succeeding probes are randomly chosen depending only on $h_1(K)$ and $h_2(K)$. (Thus the $(M-2)!^{M(M-1)}$ possible choices of $M(M-1)$ probe sequences with this property are considered to be equally likely.) Is this procedure asymptotically equivalent to uniform probing?

46. [*M42*] Determine C'_N and C_N for the open addressing method which uses the probe sequence

$$h(K), 0, 1, \ldots, h(K)-1, h(K)+1, \ldots, M-1.$$

47. [*M25*] Find the average number of probes needed by open addressing when the probe sequence is

$$h(K), h(K)-1, h(K)+1, h(K)-2, h(K)+2, \ldots.$$

This probe sequence was once suggested because all the distances between consecutive probes are distinct when M is even. [*Hint:* Find the trick and this problem is easy.]

▶**48.** [*M21*] Analyze the open addressing method that probes locations $h_1(K)$, $h_2(K)$, $h_3(K)$, . . . , given an infinite sequence of mutually independent random hash functions

$h_n(K)$. Note that it is possible to probe the same location twice, e.g. if $h_1(K) = h_2(K)$, but this is rather unlikely.

49. [*HM24*] Generalizing exercise 34 to the case of b records per bucket, determine the average number of probes (i.e., file accesses) C_N and C'_N, for chaining with separate lists, assuming that a list containing k elements requires $\min(1, k - b + 1)$ probes in an unsuccessful search. Instead of using the exact probability P_{Nk} as in exercise 34, use the *Poisson approximation*

$$\binom{N}{k}\left(\frac{1}{M}\right)^k\left(1-\frac{1}{M}\right)^{N-k} = \frac{N}{M}\,\frac{N-1}{M}\cdots\frac{N-k+1}{M}\left(1-\frac{1}{M}\right)^N\left(1-\frac{1}{M}\right)^{-k}\frac{1}{k!}$$
$$= e^{-\rho}\rho^k/k! + O(M^{-1}),$$

which is valid for $N = \rho M$ as $M \to \infty$.

50. [*M20*] Show that $Q_1(M, N) = M - (M - N - 1)Q_0(M, N)$, in the notation of (42). [*Hint:* Prove first that $Q_1(M, N) = (N + 1)Q_0(M, N) - NQ_0(M, N - 1)$.]

51. [*HM16*] Express the function $R(\alpha, n)$ defined in (55) in terms of the function Q_0 defined in (42).

52. [*HM20*] Prove that $Q_0(M, N) = \int_0^\infty e^{-t}(1 + t/M)^N\,dt$.

53. [*HM20*] Prove that the function $R(\alpha, n)$ can be expressed in terms of the incomplete gamma function, and use the result of exercise 1.2.11.3–9 to find the asymptotic value of $R(\alpha, n)$ to $O(n^{-2})$ as $n \to \infty$, for fixed $\alpha < 1$.

54. [*40*] Experiment with the behavior of Algorithm C when it has been adapted to external searching as described in the text.

55. [*HM43*] Generalize the Schay-Spruth model, discussed after Theorem P, to the case of M buckets of size b. Prove that $C(z)$ is equal to $Q(z)/(B(z) - z^b)$, where $Q(z)$ is polynomial of degree b and $Q(1) = 0$. Show that the average number of probes is

$$1 + \frac{M}{N}C'(1) = 1 + \frac{1}{b}\left(\frac{1}{1-q_1} + \cdots + \frac{1}{1-q_{b-1}} - \frac{1}{2}\,\frac{B''(1) - b(b-1)}{B'(1) - b}\right),$$

where $q_1, \ldots, q_{b-1}$ are the roots of $Q(z)/(z - 1)$. Replacing the binomial probability distribution $B(z)$ by the Poisson approximation $P(z) = e^{b\alpha(z-1)}$, where $\alpha = N/Mb$, and using Lagrange's inversion formula (cf. Eq. 2.3.4.4–9 and exercise 4.7–8), reduce your answer to Eq. (61).

56. [*M48*] Generalize Theorem K, obtaining an exact analysis of linear probing with buckets of size b.

57. [*M47*] Does the uniform assignment of probabilities to probe sequences give the minimum value of C_N, over all open addressing methods?

58. [*M21*] (S. C. Johnson.) Find ten permutations on {0, 1, 2, 3, 4} that are equivalent to uniform hashing in the sense of Theorem U.

59. [*M25*] Prove that if an assignment of probabilities to permutations is equivalent to uniform hashing, in the sense of Theorem U, the number of permutations with nonzero probabilities exceeds M^a for any fixed exponent a, when M is sufficiently large.

60. [*M48*] Let us say that an open addressing scheme involves *single hashing* if it uses exactly M probe sequences, one beginning with each possible value of $h(K)$, each of which occurs with probability $1/M$.

Are the best single-hashing schemes (in the sense of minimum C_N) asymptotically better than the random ones analyzed in exercise 44?

61. [*M46*] Is the method analyzed in exercise 46 the worst possible single-hashing scheme? (Cf. exercise 60.)

62. [*M49*] How good can a single-hashing scheme be when the increments $p_1\, p_2 \ldots p_{M-1}$ in the notation of exercise 44 are fixed for all K? (Examples of such methods are linear probing and the sequences considered in exercises 20 and 47.)

63. [*M25*] If repeated random insertions and deletions are made in a scatter table, how many independent insertions are needed on the average before all M locations have become occupied at one time or another?

64. [*M46*] Analyze the expected behavior of Algorithm R. How many times will step R4 be performed, on the average?

▸**65.** [*20*] (*Variable-length keys.*) Many applications of scatter tables deal with keys that can be any number of characters long. In such cases we can't simply store the key in the table as in the programs of this section. What would be a good way to deal with variable-length keys in a scatter table on the MIX computer?

6.5. RETRIEVAL ON SECONDARY KEYS

We have now completed our study of searching for "primary keys," i.e., for keys which uniquely specify a record in a file. But it is sometimes necessary to conduct a search based on the values of other fields in the records besides the primary key; these other fields are often called "secondary keys" or "attributes" of the record. For example, in an enrollment file which contains information about the students at a university, it may be desirable to search for all sophomores from Ohio who are not majoring in mathematics or statistics; or to search for all unmarried French-speaking graduate student women; etc.

In general, we assume that each record contains several attributes, and we want to search for all records that have certain values of certain attributes. The specification of which records are desired is called a *query*. Queries are usually restricted to at most the following three types:

a) A *simple query* which gives a specific value of a specific attribute; e.g., MAJOR = MATHEMATICS; or RESIDENCE.STATE = OHIO.

b) A *range query* which gives a specific range of values for a specific attribute; e.g., COST $<$ \$18.00; or $21 \leq$ AGE ≤ 23.

c) A *Boolean query* which consists of the previous types combined with the operations AND, OR, NOT; e.g.,

```
(CLASS = SOPHOMORE) AND (RESIDENCE.STATE = OHIO)
      AND NOT ((MAJOR = MATHEMATICS) OR (MAJOR = STATISTICS)).
```

The problem of discovering efficient search techniques for these three types of queries is already quite difficult, and therefore queries of more complicated types are usually not considered. For example, a railroad company might have a file giving the current status of all its freight cars; a query such as "find all empty refrigerator cars within 500 miles of Seattle" would not be explicitly allowed, unless "distance from Seattle" were an attribute stored within each record instead of a complicated function to be deduced from other attributes. And the use of logical quantifiers, in addition to AND, OR, and NOT, would introduce further complications, limited only by the imagination of the query-poser; given a file of baseball statistics, for example, we might ask for the longest consecutive hitting streak in night games. These examples are complicated, but they can still be handled by taking one pass through a suitably-arranged file; other queries are even more difficult, e.g. to find all pairs of records that have the same values on five or more attributes (without specifying which attributes must match). Such queries may be regarded as general programming tasks which are beyond the scope of this discussion, although they can often be broken down into subproblems of the kind considered here.

Before we begin to study the various techniques for secondary key retrieval, it is important to put the subject in a proper economic context. Although a vast number of applications fit into the general framework of the three types of queries outlined above, not many of these applications are really suited to the

sophisticated techniques we shall be studying, and some of them are better done by hand than by machine! Computers have increased the speed of scientific calculations by a factor of 10^7 or 10^8, but they have provided nowhere near this gain in efficiency with respect to problems of information handling. When large quantities of data are involved, today's computers are still constrained to working at mechanical (instead of electronic) speeds, so there is no dramatic increase in performance per unit cost when a computer system replaces a manual system. We shouldn't expect too much of a computer just because it performs certain other tasks so well.

People climb Mt. Everest "because it is there" and because tools have been developed which make the climb possible; similarly, when faced with a mountain of data, people are tempted to use a computer to find the answer to the most difficult queries they can dream up, in an "on-line, real-time" environment, without properly balancing the cost. The desired calculations are possible, but they're not right for everyone's application.

For example, consider the following simple approach to secondary key retrieval: After *batching* a number of queries, we can do a sequential search through the entire file, retrieving all the relevant records. ("Batching" means that we accumulate a number of queries before doing anything about them.) This method is quite satisfactory if the file isn't too large and if the queries don't have to be handled immediately. It can be used even with tape files, and it only ties up the computer at odd intervals, so it will tend to be very economical in terms of equipment costs. Moreover, it will even handle computational queries of the "distance to Seattle" type discussed above.

Another simple way to facilitate secondary key retrieval is to let *people* do part of the work, by providing them with suitable printed indexes to the information. This method is often the most reasonable and economical way to proceed (provided, of course, that the old paper is recycled whenever a new index is printed).

The applications which are not satisfactorily handled by the above simple schemes involve very large files for which quick responses to queries are important. Such a situation would occur, for example, if the file were continuously being queried by a number of simultaneous users, or if the queries were being generated by machine instead of by people. Our goal in this section will be to see how well we can do secondary key retrieval with conventional computers, under various assumptions about the file structure.

A lot of good ideas have been developed for dealing with the problem, but (as the reader will have guessed from all these precautionary remarks) the algorithms are by no means as good as those available for primary key retrieval. Because of the wide variety of files and applications, we will not be able to give a complete discussion of all the possibilities that have been considered, or to analyze the behavior of each algorithm in typical environments. The remainder of this section presents the basic approaches which have been proposed, and it is left to the reader's imagination to decide what combination of techniques is most appropriate in each particular case.

Inverted files. The first important class of techniques for secondary key retrieval is based on the idea of an "inverted file." This does not mean that the file is turned upside down, it means that the roles of records and attributes are reversed. Instead of listing the attributes of a given record, we list the records having a given attribute.

We encounter inverted files (under other names) quite often in our daily lives. For example, the inverted file corresponding to a Russian-English dictionary is an English-Russian dictionary. The inverted file corresponding to this book is the index which appears at the close of the book. Accountants traditionally use "double-entry bookkeeping," where all transactions are entered both in a cash account and in a customer account, so that the current cash position and customer liability are both readily accessible.

In general, an inverted file usually doesn't stand by itself, it is to be used together with the original uninverted file; it provides duplicate, redundant information in order to speed up secondary key retrieval. The components of an inverted file are called *inverted lists*, namely the lists of all records having a given value of some attribute.

Like all lists, the inverted lists can be represented in many ways within a computer, and different modes of representation are appropriate at different times. Some secondary key fields have only two values (e.g., a 'sex' attribute), and the corresponding inverted lists are quite long; but other fields typically have a great many values with few duplications (e.g., a 'telephone number' attribute).

Imagine, for example, that we want to store the information in a telephone directory so that all entries can be retrieved on the basis of either name, phone number, or residence address. One solution is simply to make three separate files, oriented to retrieval on each type of key. Another idea is to combine these files, for example by making three hash tables which serve as the list heads for the chaining method. In the latter scheme, each record of the file would be an element of three lists, and it would therefore contain three link fields; this is the so-called *Multilist* method which is discussed further below. A third possibility is to combine the three files into one super file, by analogy with library card catalogues in which author cards, title cards, and subject cards are all alphabetized together.

A consideration of the format used in the index to this book leads to further ideas on inverted list representation. For secondary key fields in which there are typically five or so entries per attribute value, we can simply make a short sequential list of the record locations (analogous to page locations in a book index), following the key value. If related records tend to be clustered consecutively, a "range specification" code (e.g., pages 200 through 214) could be useful. If the records in the file tend to be reallocated frequently, it may be better to use primary keys instead of record locations in the inverted files, so that no updating needs to be done when the locations change; for example, references to Bible passages are always given by chapter and verse, and the index to some books is based on paragraph numbers instead of page numbers.

None of these ideas is especially appropriate for the case of a two-valued attribute like 'SEX'. In such a case only one inverted list is needed, of course, since the non-males will be female and conversely. If each value relates to about half the items of the file, the inverted list will be horribly long, but we can solve the problem rather nicely on a binary computer by using a bit string representation, with each bit specifying the value of a particular record. Thus the bit string 01001011101 $\cdots$ might mean that the first record in the file refers to a male, the second female, the next two male, etc. (Cf. also the discussion of prime number representation at the end of Section 6.1.)

The above methods suffice to handle simple queries about specific attribute values. A slight extension makes it possible to treat range queries, except that some comparison-based search scheme (Section 6.2) must be used instead of hashing.

For Boolean queries like "(MAJOR = MATHEMATICS) AND (RESIDENCE.STATE = OHIO)", we need to intersect two inverted lists. This can be done in several ways; for example, if both lists are ordered, one pass through each one will pick out all common entries. Alternatively, we could select the *shortest* list and look up each of its records, checking the other attributes; but this method works only for AND's, not for OR's, and it is unattractive on external files because it requires many accesses to records that will not satisfy the query.

The same considerations show that a Multilist organization as described above is inefficient for Boolean queries, on an external file, since it implies many unnecessary accesses. For example, imagine what would happen if the index to this book were organized in a Multilist manner. This would mean that each entry of the index would refer only to the last page on which its particular subject was mentioned; then on every page there would be a further reference, for each subject on that page, to the previous occurrence of that subject. In order to find all pages relevant to "[Analysis of algorithms] and [(External sorting) or (External searching)]", it would be necessary to turn many pages. On the other hand, the same query can be resolved by looking at only two pages of the real index as it actually appears, doing simple operations on the inverted lists in order to find the small subset of pages which satisfy the query.

When an inverted list is represented as a bit string, Boolean combinations of simple queries are, of course, rather easily performed, because computers can manipulate bit strings at relatively high speed. For mixed queries in which some attributes are represented as sequential lists of record numbers while other attributes are represented as bit strings, it is not difficult to convert the sequential lists into bit strings, then to perform the Boolean operations on these bit strings.

A quantitative example of a hypothetical application may be helpful at this point. Assume that we have 1,000,000 records of 40 characters each, and that our file is stored on MIXTEC disks, as described in Section 5.4.9. The file itself therefore fills two disk packs, and the inverted lists will probably fill several more. Each track contains 5000 characters = 30,000 bits, so an inverted list for a particular attribute will take up at most 34 tracks. (This maximum

number of tracks occurs when the bitstring representation is the shortest possible one.) Suppose that we have a rather involved query that refers to a Boolean combination of 10 inverted lists; in the worst case we will have to read 340 tracks of information from the inverted file, for a total read time of 340×25 ms $= 8.5$ sec. The average latency delay will be about one half of this, but by careful programming it may be possible to eliminate the latency. By storing the first track of each bitstring list in one cylinder, and the second track of each list in the next, etc., most of the seek time will be eliminated, so we can estimate the seek time as at most about 34×26 ms ≈ 0.9 sec (or twice this if two different disk packs are involved). Finally, if k records satisfy the query, it will take about $k \times (60$ ms (seek) $+ 12.5$ ms (latency) $+ 0.2$ ms (read)) extra time to fetch each one for subsequent processing. Thus an optimistic estimate of the total expected time to process this rather complicated query is $< (10 + .073k)$ seconds. This may be contrasted with about 210 seconds to read through the entire file at top speed under the same assumptions without using any inverted lists.

This example shows that space optimization is closely related to time optimization in a disk memory; the time to process the inverted lists is roughly the time needed to seek and to read them.

The above discussion has more or less assumed that the file is not growing or shrinking as we query it; what should we do if frequent updates are necessary? In many applications it is sufficient to batch a number of requests for updates, and to take care of them in dull moments when no queries need to be answered. Alternatively, if updating the file has high priority, the method of B-trees (Section 6.2.4) is attractive. The entire collection of inverted lists could be made into one huge B-tree, with special conventions for the leaves so that the branch nodes contain key values while the leaves contain both keys and lists of pointers of records.

We have glossed over another point in the above discussion, namely the difficult subject of Boolean combinations of range queries. For example, suppose that the records of the file refer to North American cities, and that the query asks for all cities with

$$(21.49° \le \texttt{LATITUDE} \le 37.41°) \texttt{ AND } (70.34° \le \texttt{LONGITUDE} \le 75.72°).$$

No really nice data structures seem to be available for such "orthogonal range queries." (Reference to a map will show that many cities satisfy this LATITUDE range, and many satisfy the LONGITUDE range, but hardly any cities lie in both ranges.) Perhaps the best approach is to partition the set of all possible LATITUDE and LONGITUDE values rather coarsely, with only a few classes per attribute (e.g., by truncating to the next lower multiple of 5°), then to have one inverted list for each combined (LATITUDE, LONGITUDE) class. This is like having maps

with one page for each local region. Using 5° intervals, the above query would refer to eight pages, namely (20°, 70°), (25°, 70°), . . . , (35°, 75°). The range query needs to be processed for each of these pages, either by going to a finer partition within the page or by direct reference to the records themselves, depending on the number of records corresponding to that page. In a sense this is a tree structure with two-dimensional branching at each internal node.

Another tree-structured approach to orthogonal range queries has been proposed by Bruce McNutt. Suppose, for example, that we wish to handle queries like "What is the nearest city to point x?", given the value of x. Each node of McNutt's proposed binary search tree corresponds to a city y and a "test radius" r; the left subtree of this node corresponds to all cities z entered subsequently into this part of the tree such that the distance from y to z is $\leq r + \delta$, and the right subtree similarly is for distances $\geq r - \delta$. Here δ is a given tolerance; cities between $r - \delta$ and $r + \delta$ away from y must be entered in *both* subtrees. Searching in such a "post-office tree" makes it possible to locate all cities within distance δ of a given point. (See Fig. 45.)

Several experiments based on this idea were conducted in 1972 by McNutt and Edward Pring, using the 231 most populous cities in the continental United States, in random order, as an example data base. They let the test radii shrink in a regular manner, replacing r by $0.67r$ when going to the left, and by $0.57r$ when going to the right, except that r was left unchanged when taking the second of two consecutive right branches. The result was that 610 nodes were required in the tree for $\delta = 20$ miles, and 1600 nodes were required for $\delta = 35$ miles. The top levels of their smaller tree are shown in Fig. 45. (In the remaining levels of this tree, Orlando FL appeared below both Jacksonville and Miami. Some cities occurred quite often; e.g., 17 of the nodes were for Brockton MA!)

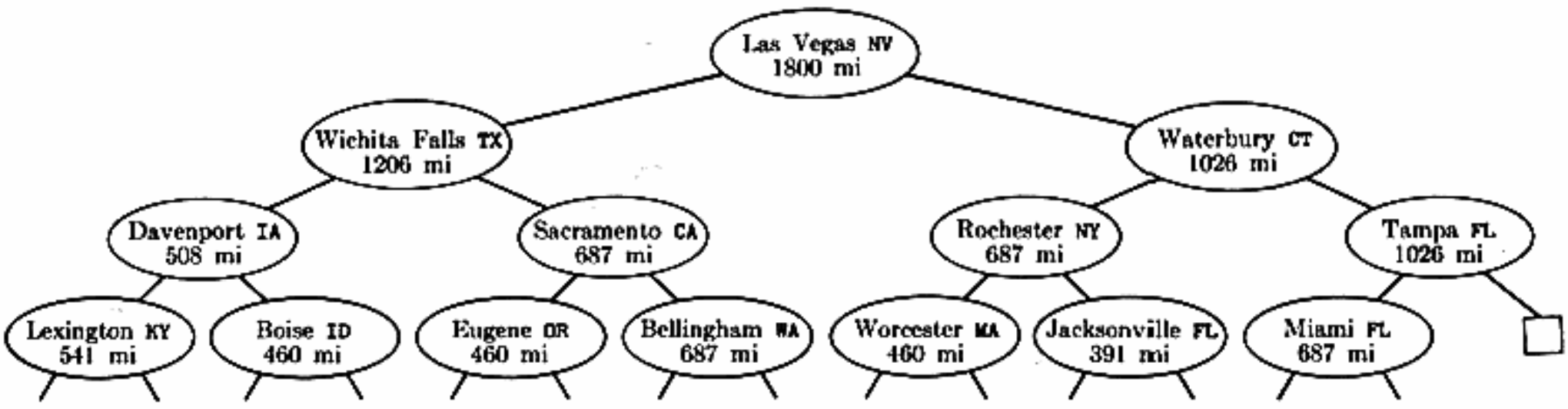


Fig. 45. The top levels of an example "post-office tree." To search for all cities near a given point x, start at the root: If x is within 1800 miles of Las Vegas, go left, otherwise go to the right; then repeat the process until encountering a terminal node. The method of tree construction ensures that all cities within 20 miles of x will be encountered during this search.

Table 1 A FILE WITH BINARY ATTRIBUTES

	Allspice	Anise seed	Baking powder	Baking soda	Butter	Cardamom	Chocolate	Cinnamon	Cloves	Coconut	Coffee	Cornstarch	Dates	Egg white	Egg yolk	Flour	Ginger	Lemon juice	Lemon peel	Milk	Molasses	Nutmeg	Nuts	Oatmeal	Raisins	Salt	Sugar, brown	Sugar, granulated	Sugar, powdered	Vanilla extract	Special ingredients
Almond Lace Wafers	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	1	0	0	0	0	1	0	0	—
Applesauce-Spice Squares	0	0	0	1	1	0	0	1	1	0	0	0	0	1	1	1	0	0	0	0	0	1	1	0	1	0	0	1	1	1	Applesauce
Banana-Oatmeal Cookies	0	0	0	1	1	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	1	1	1	0	1	0	1	0	1	Bananas
Chocolate Chip Cookies	0	0	0	1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	1	—
Coconut Macaroons	0	0	1	0	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	1	—
Cream-Cheese Cookies	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	1	Cream cheese
Delicious Prune Bars	0	0	0	0	1	0	0	0	0	1	0	1	0	1	1	1	0	0	0	0	0	0	1	0	0	1	1	0	0	0	Oranges, prunes
Double-Chocolate Drops	0	0	0	1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	0	—
Dream Bars	0	0	1	0	1	0	0	0	0	1	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	1	0	0	1	—
Filled Turnovers	0	0	1	0	1	0	0	0	0	0	0	0	1	1	1	1	0	1	1	0	0	0	1	0	0	1	0	1	0	1	—
Finska Kakor	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	0	Almond extract
Glazed Gingersnaps	0	0	1	1	1	0	0	1	1	0	0	0	0	1	1	1	1	0	0	1	1	0	0	0	0	0	1	0	1	0	Vinegar
Hermits	0	0	0	1	1	0	0	1	0	0	1	0	0	1	1	1	0	0	0	0	0	1	0	0	1	1	1	0	0	0	Apricots
Jewel Cookies	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	1	Currant jelly
Jumbles	0	0	0	0	1	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	1	Salad oil
Kris Kringles	0	0	0	0	1	1	0	0	0	0	0	0	0	1	1	1	0	0	1	0	0	0	1	0	0	0	0	1	0	0	—
Lebkuchen Rounds	1	0	0	1	0	0	0	1	1	0	0	0	0	1	1	1	0	1	1	1	0	1	1	0	0	1	1	0	1	0	Honey
Meringues	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	Candied cherries
Moravian Spice Cookies	1	0	0	1	1	0	0	1	1	0	0	0	0	1	0	1	1	0	0	0	1	1	0	0	0	1	1	0	1	0	—
Oatmeal-Date Bars	0	0	0	1	1	0	0	0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	1	1	0	1	1	1	0	0	—
Old-Fashioned Sugar Cookies	0	0	1	1	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1	0	0	0	1	0	1	0	1	Sour cream
Peanut-Butter Pinwheels	0	0	0	1	1	0	1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	1	1	0	0	1	Peanut butter
Petticoat Tails	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1	1	—
Pfeffernuesse	1	1	1	0	0	1	0	1	1	0	1	0	0	1	1	1	0	0	1	0	0	1	1	0	0	1	1	0	1	0	Citron, mace, pepper
Scotch Oatmeal Shortbread	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	1	0	1	0	1	—
Shortbread Stars	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	—
Springerle	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	1	0	1	1	0	—
Spritz Cookies	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	1	0	1	—
Swedish Kringler	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	1	0	0	—
Swiss-Cinnamon Crisps	0	0	0	0	1	0	0	1	0	0	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	1	1	1	0	0	—
Toffee Bars	0	0	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0	0	0	1	0	0	1	—
Vanilla-Nut Icebox Cookies	0	0	1	0	1	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	1	0	0	1	0	1	0	1	—

Reference: *McCall's Cook Book* (New York: Random House, 1963), Chapter 9.

Compound attributes. It is possible to combine two or more attributes into one super attribute. For example, a "(CLASS, MAJOR) attribute" could be created by combining the CLASS and MAJOR fields of a university enrollment file. In this way queries can often be satisfied by taking the union of disjoint, short lists instead of the intersection of longer lists.

The idea of attribute combination has been further developed by V. Y. Lum [*CACM* **13** (1970), 660–665], who suggests ordering the inverted lists of combined attributes lexicographically from left to right, and making multiple copies, with the individual attributes permuted in a clever way. For example, suppose that we have three attributes A, B, C; we can form three combined attributes

$$(A, B, C), \quad (B, C, A), \quad (C, A, B) \tag{1}$$

and construct ordered inverted lists for each of these. (Thus in the first list, the records occur in order of their A values, with all records of the same A value in order by B and then by C.) This organization makes it possible to satisfy queries based on any combination of the three attributes; e.g., all records having specified values for A and C will appear contiguously in the third list.

Similarly, from four attributes A, B, C, D, we can form the six combined attributes

$$(A, B, C, D), \quad (B, C, D, A), \quad (B, D, A, C), \quad (C, A, D, B), \quad (C, D, A, B), \quad (D, A, B, C), \tag{2}$$

which suffice to answer all combinations of simple queries relating to the simultaneous values of one, two, three, or four of the attributes. There is a general procedure for constructing $\binom{n}{k}$ combined attributes from n attributes, where $k \le \frac{1}{2}n$, such that all records having specified combinations of $\le k$ or $\ge n - k$ of the attribute values will appear consecutively in one of the combined attribute lists (see exercise 1). Alternatively, we can get by with fewer combinations when some attributes have a limited number of values. For example if D is simply a two-valued attribute, the three combinations

$$(D, A, B, C), \quad (D, B, C, A), \quad (D, C, A, B), \tag{3}$$

obtained by placing D in front of (1), will be almost as good as (2) with only half the redundancy, since queries that do not depend on D can be treated by looking in just two places in one of the lists.

Binary attributes. It is instructive to consider the special case in which all attributes are two-valued. In a sense this is the *opposite* of combining attributes, since we can represent any value as a binary number and regard the individual bits of that number as separate attributes. Table 1 shows a typical file involving "yes-no" attributes; in this case the records stand for selected cookie recipes, and the attributes specify which ingredients are used. For example, Almond Lace Wafers are made from butter, flour, milk, nuts, and granulated sugar.

If we think of Table 1 as a matrix of zeros and ones, the transpose of the matrix is the "inverted" file, in bitstring form.

The right-hand column of Table 1 is used to indicate special items that occur only rarely. These could be coded in a more efficient way than to devote an entire column to each one; the "Cornstarch" column could be similarly treated. Dually, we could find a more efficient way to encode the "Flour" column, since flour occurs in everything except Meringues. For the present, however, let us sidestep these considerations and simply ignore the "Special ingredients" column.

Let us define a *basic query* in a binary attribute file as a request for all records having 0's in certain columns, 1's in other columns, and arbitrary values in the remaining columns. Using "*" to stand for an arbitrary value, we can represent any basic query as a sequence of 0's, 1's, and *'s. For example, consider a man who is in the mood for some coconut cookies, but he is allergic to chocolate, hates anise, and has run out of vanilla extract; he can formulate the query

```
* 0 * * * * 0 * * 1 * * * * * * * * * * * * * * * * * 0.    (4)
```

Table 1 now says that Dream Bars are just the thing.

Before we consider the general problem of organizing a file for basic queries, it is important to look at the special case where no 0's are specified, only 1's and *'s. This may be called an *inclusive query*, because it asks for all records which include a certain set of attributes, if we assume that 1's denote attributes that are present and that 0's denote attributes that are absent. For example, the recipes in Table 1 which call for both baking powder and baking soda are Glazed Gingersnaps and Old-Fashioned Sugar Cookies.

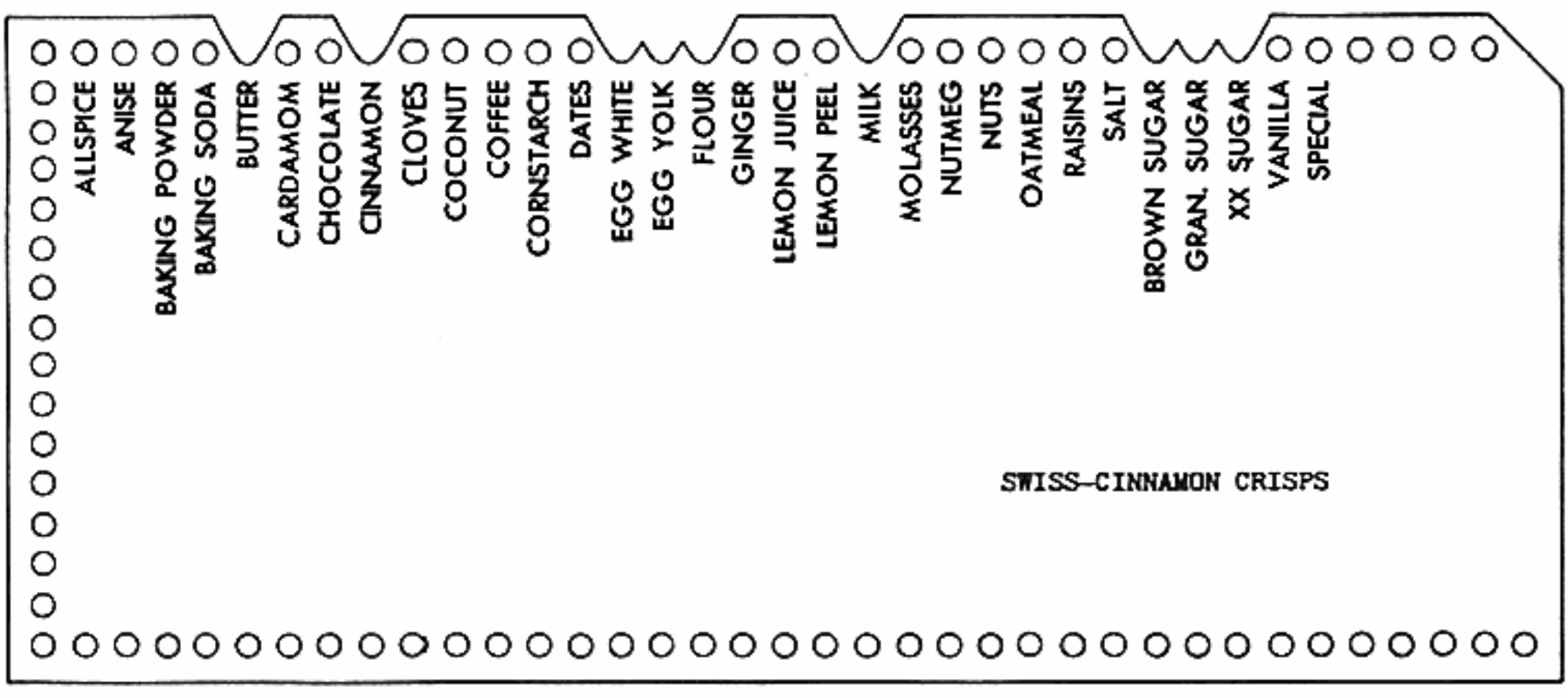


Fig. 46. An edge-notched card.

In some applications it is sufficient to provide for the special case of inclusive queries. This occurs, for example, in the case of many manual card-filing systems, such as "edge-notched cards" or "feature cards." An edge-notched card system corresponding to Table 1 would have one card for every recipe, with holes cut out for each ingredient (see Fig. 46). In order to process an inclusive query, the file of cards is arranged into a neat deck and needles are put in each column position corresponding to an attribute that is to be included. After raising the needles, all cards having the appropriate attributes will drop out.

A feature-card system works on the inverse file in a similar way. In this case there is one card for every attribute, and holes are punched in designated positions on the surface of the card for every record possessing that attribute. An ordinary 80-column card can therefore be used to tell which of $12 \times 80 = 960$ records have a given attribute. To process an inclusive query, the feature cards for the specified attributes are selected and put together; then light will shine through all positions corresponding to the desired records. This operation is analogous to the treatment of Boolean queries by intersecting inverted bit strings as explained above.

Superimposed coding. The reason these manual card systems are of special interest to us is that ingenious schemes have been devised to save space on edge-notched cards; the same principles can be applied in the representation of computer files. Superimposed coding is a technique similar to hashing, and it was actually invented several years before hashing itself was discovered. The idea is to map attributes into random k-bit codes in an n-bit field, and to superimpose the codes for each attribute that is present in a record. An inclusive query for some set of attributes can be converted into an inclusive query for the corresponding superimposed bit codes. A few extra records may satisfy this query, but the number of such "false drops" can be statistically controlled. [Cf. Calvin N. Mooers, *Amer. Chem. Soc. Meeting* **112** (September, 1947), 14E–15E; *American Documentation* **2** (1951), 20–32.]

As an example of superimposed coding, let's consider Table 1 again, but only the flavorings instead of the basic ingredients like baking powder, shortening, eggs, and flour. Table 2 shows what happens if we assign random 2-bit codes in a 10-bit field to each of the flavoring attributes and superimpose the coding. For example, the entry for Chocolate Chip Cookies is obtained by superimposing the codes for chocolate, nuts, and vanilla:

$$0010001000 \vee 0000100100 \vee 0000001001 = 0010101101.$$

The superimposition of these codes also yields some spurious attributes, in this case allspice, candied cherries, currant jelly, peanut butter, and pepper; these will cause false drops to occur on certain queries (and they also suggest the creation of a new recipe called False Drop Cookies!).

Table 2

AN EXAMPLE OF SUPERIMPOSED CODING

Codes for individual flavorings

Almond extract	0100000001	Dates	1000000100
Allspice	0000100001	Ginger	0000110000
Anise seed	0000011000	Honey	0000000011
Applesauce	0010010000	Lemon juice	1000100000
Apricots	1000010000	Lemon peel	0011000000
Bananas	0000100010	Mace	0000010100
Candied cherries	0000101000	Molasses	1001000000
Cardamom	1000000001	Nutmeg	0000010010
Chocolate	0010001000	Nuts	0000100100
Cinnamon	1000000010	Oranges	0100000100
Citron	0100000010	Peanut butter	0000000101
Cloves	0001100000	Pepper	0010000100
Coconut	0001010000	Prunes	0010000010
Coffee	0001000100	Raisins	0101000000
Currant jelly	0010000001	Vanilla extract	0000001001

Superimposed codes

Almond Lace Wafers	0000100100	Lebkuchen Rounds	1011110111
Applesauce-Spice Squares	1111111111	Meringues	1000101100
Banana-Oatmeal Cookies	1000111111	Moravian Spice Cookies	1001110011
Chocolate Chip Cookies	0010101101	Oatmeal-Date Bars	1000100100
Coconut Macaroons	0001111101	Old-Fashioned Sugar Cookies	0000011011
Cream-Cheese Cookies	0010001001	Peanut-Butter Pinwheels	0010001101
Delicious Prune Bars	0111110110	Petticoat Tails	0000001001
Double-Chocolate Drops	0010101100	Pfeffernuesse	1111111111
Dream Bars	0001111101	Scotch Oatmeal Shortbread	0000001001
Filled Turnovers	1011101101	Shortbread Stars	0000000000
Finska Kakor	0100100101	Springerle	0011011000
Glazed Gingersnaps	1001110010	Spritz Cookies	0000001001
Hermits	1101010110	Swedish Kringler	0000000000
Jewel Cookies	0010101101	Swiss-Cinnamon Crisps	1000000010
Jumbles	1000001011	Toffee Bars	0010111101
Kris Kringles	1011100101	Vanilla-Nut Icebox Cookies	0000101101

Superimposed coding actually doesn't work very well in Table 2, because that table is a small example with lots of attributes present. In fact, Applesauce-Spice Squares will drop out for *every* query, since it was obtained by superimposing seven codes that cover all ten positions; and Pfefferneusse is even worse, obtained by superimposing twelve codes! On the other hand Table 2 works surprisingly well in some respects; e.g., for the query "Vanilla extract", only the record for Pfefferneusse comes out as a false drop.

A more appropriate example of superimposed coding occurs if we have, say, a 32-bit field and a set of $\binom{32}{3} = 4960$ different attributes, where each record is allowed to possess up to six attributes and each attribute is encoded by specifying 3 of the 32 bits. In this situation, if we assume that each record has six randomly selected attributes, the probability of a false drop in an inclusive query

$$\begin{aligned} &\text{on one attribute is} && .0806933;\\ &\text{on two attributes is} && .0070878;\\ &\text{on three attributes is} && .0006709;\\ &\text{on four attributes is} && .0000679;\\ &\text{on five attributes is} && .0000073;\\ &\text{on six attributes is} && .0000008. \end{aligned} \tag{5}$$

Thus if there are M records which do not actually satisfy a two-attribute query, about $.007M$ will have a superimposed code that spuriously matches all code bits of the two specified attributes.. (These probabilities are computed in exercise 4.) The total number of bits needed in the inverted file is only 32 times the number of records, which is less than half the number of bits needed to specify the attributes themselves in the original file.

Malcolm C. Harrison [*CACM* **14** (1971), 777–779] has observed that superimposed coding can be used to speed up *text searching*. Assume that we want to locate all occurrences of a particular string of characters in a long body of text, without building an extensive table as in Algorithm 6.3P; and assume, for example, that the text is diyided into individual lines $c_1c_2 \ldots c_{50}$ of 50 characters each. Harrison suggests encoding each of the 49 pairs c_1c_2, c_2c_3, . . . , $c_{49}c_{50}$ by hashing each of them into a number between 0 and 127, say; then the "signature" of the line $c_1c_2 \ldots c_{50}$ is the string of 128 bits $b_0b_1 \ldots b_{127}$, where $b_i = 1$ if and only if $h(c_jc_{j+1}) = i$ for some j.

If now we want to search for all occurrences of the word NEEDLE in a large text file called HAYSTACK, we simply look for all lines whose signature contains 1-bits in positions h(NE), h(EE), h(ED), h(DL), and h(LE). Assuming that the hash function is random, the probability that a random line contains all these bits in its signature is only 0.00341 (cf. exercise 4); hence the intersection of five inverted-list bit strings will rapidly identify all the lines containing NEEDLE, together with a few false drops.

The assumption of randomness is not really justified in this application, since typical text has so much redundancy; the distribution of bigrams, i.e. of two-letter combinations in words, is highly biased. For example, it will probably be very helpful to discard all pairs c_jc_{j+1} containing a blank character, since blanks are usually much more common than any other symbol.

Another interesting application of superimposed coding to search problems has been suggested by Burton H. Bloom [*CACM* **13** (1970), 422–426]; his method actually applies to *primary* key retrieval, although it is most appropriate for us to discuss it in this section. Imagine a search application with a large data base

in which no calculation needs to be done if the search was unsuccessful. For example, we might want to check somebody's credit rating or passport number, etc., and if no record for him appears in the file we don't have to investigate further. Similarly in an application to computerized typesetting, we might have a simple algorithm that hyphenates most words correctly, but it fails on some 50,000 exceptional words; if we don't find the word in the exception file we are free to use the simple algorithm.

In such situations it is possible to maintain a bit table in internal memory so that most keys not in the file can be recognized as absent without making *any* references to the external memory. Here's how: Let the internal bit table be $b_0 b_1 \ldots b_{m-1}$, where M is rather large. For each key K_j in the file, compute k independent hash functions $h_1(K_j), \ldots, h_k(K_j)$, and set the corresponding k b's equal to 1. (These k values need not be distinct.) Thus $b_i = 1$ if and only if $h_l(K_j) = i$ for some j and l. Now to determine if a search argument K is in the external file, first test whether or not $b_{h_l(K)} = 1$ for $1 \le l \le k$; if not, there is no need to access the external memory, but if so, a conventional search will probably find K if k and M have been chosen properly. The chance of a false drop when there are N records in the file is approximately $(1 - e^{-kN/M})^k$. In a sense, Bloom's method treats the entire file as one record, with the primary keys as the attributes which are present, and with superimposed coding in a huge M-bit field.

Still another variation of superimposed coding has been developed by Richard A. Gustafson [Ph.D. thesis (Univ. South Carolina, 1969)]. Suppose that we have N records and that each record possesses six attributes chosen from a set of 10,000 possible attributes. The records may, for example, stand for technical articles and the attributes may be keywords describing the article. Let h be a hash function which maps each attribute into a number between 0 and 15. If a record has attributes $a_1, a_2, \ldots, a_6$, Gustafson suggests mapping the record into the 16-bit number $b_0 b_1 \ldots b_{15}$, where $b_i = 1$ if and only if $h(a_j) = i$ for some j; and furthermore if this method results in only k of the b's equal to 1, for $k < 6$, another $6 - k$ 1's are supplied by some random method (not necessarily depending on the record itself). There are $\binom{16}{9} = 8008$ sixteen-bit codes in which exactly six 1 bits are present, and with luck about $N/8008$ records will be mapped into each value. We can keep 8008 lists of records, directly calculating the address corresponding to $b_0 b_1 \ldots b_{15}$ using a suitable formula. In fact, if the 1's occur in positions $0 \le p_1 < p_2 < \cdots < p_6$, the function

$$\binom{p_1}{1} + \binom{p_2}{2} + \cdots + \binom{p_6}{6}$$

will convert each string $b_0 b_1 \ldots b_1$ into a unique number between 0 and 8007, as we have seen in exercises 1.2.6–56, 2.2.6–7.

Now if we want to find all records having three particular attributes A_1, A_2, A_3, we compute $h(A_1)$, $h(A_2)$, $h(A_3)$; assuming that these three values are distinct, we need only look at the records stored in the $\binom{13}{3} = 286$ lists whose

bit code $b_0 b_1 \ldots b_{15}$ contains 1's in those three positions. In other words, only $286/8008 \approx 3.5$ percent of the records need to be examined in the search.

Combinatorial hashing. The idea underlying Gustafson's method just described is to find some way to map the records into memory locations so that comparatively few locations are relevant to a particular query. But his method applies only to inclusive queries when the individual records possess few attributes. Another type of mapping, designed to handle arbitrary basic queries like (4) consisting of 0's, 1's, and *'s, was discovered by Ronald L. Rivest in 1971.

Suppose first that we have 1,000,000 records each containing 10 secondary keys, where each secondary key has a fairly large number of possible values. We can map the records whose secondary keys are $(K_1, K_2, \ldots, K_{10})$ into the 20-bit number

$$h(K_1)h(K_2)\ldots h(K_{10}), \tag{6}$$

where h is a hash function taking each secondary key into a 2-bit value, and (6) stands for the juxtaposition of these ten pairs of bits. This scheme maps 1,000,000 records into $2^{20} = 1{,}048{,}576$ possible values, and we can consider the total mapping as a hash function with $M = 2^{20}$; chaining can be used to resolve collisions. If we want to retrieve all records having specified values of any five secondary keys, we need to look at only 2^{10} lists [corresponding to the five unspecified bit pairs in (6)], so only about $1000 = \sqrt{N}$ records need to be examined on the average. (A similar approach has been suggested by M. Arisawa, *J. Inf. Proc. Soc. Japan* **12** (1971), 163–167.)

Rivest has developed this idea further so that in many cases we have the following situation. Assume that there are $N \approx 2^n$ records, each having m secondary keys. Each record is mapped into an n-bit hash address, in such a way that a query which leaves the values of k keys unspecified corresponds to approximately $N^{k/m}$ hash addresses. All the other methods we have discussed in this section (except Gustafson's) require order N steps for retrieval, although the constant of proportionality is small; for large enough N, Rivest's method will be faster, and it requires no inverted files.

But we have to define an appropriate mapping, before we can apply this technique. Here is an example with small parameters, when $m = 4$ and $n = 3$ and when all secondary keys are binary-valued; we can map 4-bit records into eight addresses as follows:

$$\begin{array}{llll} * \ 0 \ 0 \ 0 \to 1 & & 0 \ 1 \ * \ 0 \to 5 \\ * \ 1 \ 1 \ 1 \to 2 & & 1 \ 0 \ * \ 1 \to 6 \\ 0 \ * \ 0 \ 1 \to 3 & & 0 \ 0 \ 1 \ * \to 7 \\ 1 \ * \ 1 \ 0 \to 4 & & 1 \ 1 \ 0 \ * \to 8 \end{array} \tag{7}$$

An examination of this table reveals that all records corresponding to the query 0 * * * are mapped into locations 1, 2, 3, 5, and 7; and similarly *any* basic query with three *'s corresponds to exactly five locations. The basic queries with two *'s each correspond to three locations; and the basic queries with one * cor-

respond to either one or two locations, $(8 \times 1 + 24 \times 2)/32 = 1.75$ on the average. Thus we have

Number of unspecified bits in the query	Number of locations to search
4	$8 = 8^{4/4}$
3	$5 \approx 8^{3/4}$
2	$3 \approx 8^{2/4}$
1	$1.75 \approx 8^{1/4}$
0	$1 = 8^{0/4}$

(8)

Of course this is such a small example, we could handle it more easily by brute force. But it leads to nontrivial applications, since we can use it also when $m = 4r$ and $n = 3r$, mapping $4r$-bit records into $2^{3r} \approx N$ locations by dividing the secondary keys into r groups of 4 bits each and applying (7) in each group. The resulting mapping has the desired property: *A query that leaves k of the m bits unspecified will correspond to approximately $N^{k/m}$ locations.* (See exercise 6.)

Some further mappings designed by Rivest appear in exercise 7; they can be used in combination with (7) to produce combinatorial hash functions for cases with $1 \leq m/n \leq 2$. In practice, buckets of size b would be used, and we would take $N \approx 2^n b$; the case $b = 1$ has been used in the above discussion for simplicity in exposition.

***Balanced filing schemes.** Another combinatorial approach to information retrieval, based on "balanced incomplete block designs," has been the subject of considerable investigation. Although the subject is very interesting from a mathematical point of view, it has unfortunately not yet proved to be very useful by comparison with the other methods described above. A brief introduction to the theory will be presented here in order to indicate the flavor of the results, in hopes that some reader might think of a good way to put the ideas to practical use.

A *Steiner triple system* is an arrangement of v objects into unordered triples in such a way that every pair of objects occurs in exactly one triple. For example, when $v = 7$ there is essentially only one Steiner triple system, namely

Triple	Pairs included
{1, 2, 4}	{1, 2}, {1, 4}, {2, 4}
{2, 3, 5}	{2, 3}, {2, 5}, {3, 5}
{3, 4, 6}	{3, 4}, {3, 6}, {4, 6}
{4, 5, 0}	{0, 4}, {0, 5}, {4, 5}
{5, 6, 1}	{1, 5}, {1, 6}, {5, 6}
{6, 0, 2}	{0, 2}, {0, 6}, {2, 6}
{0, 1, 3}	{0, 1}, {0, 3}, {1, 3}

(9)

Since there are $\frac{1}{2}v(v-1)$ pairs of objects and three pairs per triple, there must be $\frac{1}{6}v(v-1)$ triples in all; and since each object must be paired with $v-1$

others, each object must appear in exactly $\frac{1}{2}(v-1)$ triples. These conditions imply that a Steiner triple system can't exist unless $\frac{1}{6}v(v-1)$ and $\frac{1}{2}(v-1)$ are integers, and this is equivalent to saying that v is odd and not congruent to 2 modulo 3, i.e.,

$$v \bmod 6 = 1 \text{ or } 3. \tag{10}$$

Conversely, T. P. Kirkman proved in 1847 that Steiner triple systems do exist for all $v \geq 1$ such that (10) holds. His interesting construction is given in exercise 10.

Steiner triple systems can be used to reduce the redundancy of combined inverted file indexes. For example, consider again the cookie recipe file of Table 1, and convert the rightmost column into a 31st attribute which is 1 if any special ingredients are necessary, 0 otherwise. Assume that we want to answer all inclusive queries on pairs of attributes, e.g., "What recipes use both coconut and raisins?" We could make up an inverted list for each of these $\binom{31}{2} = 465$ possible queries. But it would turn out that this takes a lot of space since Pfeffernuesse (for example) would appear in $\binom{17}{2} = 136$ of the lists, and a record with all 31 attributes would appear in every list! A Steiner triple system can be used to make a slight improvement in this situation. There is a Steiner triple system on 31 objects, with 155 triples and each pair of objects occurring in exactly one of the triples. We can associate four lists with each triple $\{a, b, c\}$, one list for all records having attributes a, b, $\bar{c}$ (i.e., not c); another for a, $\bar{b}$, c; another for $\bar{a}$, b, c; and another for records having all three attributes a, b, c. This guarantees that no record will be included in more than 155 of the inverted lists, and it saves space whenever a record has three attributes corresponding to a triple of the system.

Triple systems are special cases of block designs which have blocks of three or more objects. For example, there is a way to arrange 31 objects into sextuples so that every pair of objects appears in exactly one sextuple:

{1, 5, 17, 22, 23, 25}	{ 7, 11, 23, 28, 29, 0}	{13, 17, 29, 3, 4, 6}	{20, 24, 5, 10, 11, 13}	{26, 30, 11, 16, 17, 19}
{2, 6, 18, 23, 24, 26}	{ 8, 12, 24, 29, 30, 1}	{14, 18, 30, 4, 5, 7}	{21, 25, 6, 11, 12, 14}	{27, 0, 12, 17, 18, 20}
{3, 7, 19, 24, 25, 27}	{ 9, 13, 25, 30, 0, 2}	{15, 19, 0, 5, 6, 8}	{22, 26, 7, 12, 13, 15}	{28, 1, 13, 18, 19, 21}
{4, 8, 20, 25, 26, 28}	{10, 14, 26, 0, 1, 3}	{16, 20, 1, 6, 7, 9}	{23, 27, 8, 13, 14, 16}	{29, 2, 14, 19, 20, 22}
{5, 9, 21, 26, 27, 29}	{11, 15, 27, 1, 2, 4}	{17, 21, 2, 7, 8, 10}	{24, 28, 9, 14, 15, 17}	{30, 3, 15, 20, 21, 23}
{6, 10, 22, 27, 28, 30}	{12, 16, 28, 2, 3, 5}	{18, 22, 3, 8, 9, 11}	{25, 29, 10, 15, 16, 18}	{ 0, 4, 16, 21, 22, 24}
		{19, 23, 4, 9, 10, 12}		

(11)

(This design is formed from the first block by addition mod 31. To verify that it has the stated property, note that the 30 values $(a_i - a_j) \bmod 31$, for $i \neq j$, are distinct, where $(a_1, a_2, \ldots, a_6) = (1, 5, 17, 22, 23, 25)$. To find the sextuple containing a given pair (x, y), choose i and j such that $a_i - a_j \equiv x - y$ (modulo 31); now if $k = (x - a_i) \bmod 31$, we have $(a_i + k) \bmod 31 = x$ and $(a_j + k) \bmod 31 = y$.)

We can use the above design to store the inverted lists in such a way that no record can appear more than 31 times. Each sextuple $\{a, b, c, d, e, f\}$ is associated with 57 lists, for the various possibilities of records having two or

more of the attributes a, b, c, d, e, f, namely $(a, b, \bar{c}, \bar{d}, \bar{e}, \bar{f})$, $(a, \bar{b}, c, \bar{d}, \bar{e}, \bar{f})$, . . . , (a, b, c, d, e, f); and the answer to each inclusive 2-attribute query is the disjoint union of 16 appropriate lists in the appropriate sextuple. For this design, Pfefferneuse would be stored in 29 of the 31 blocks, since that record has two of the six attributes in all but blocks {19, 23, 4, 9, 10, 12} and {13, 17, 29, 3, 4, 6} if we number the columns from 0 to 30.

A similar idea can be used to cut down redundancy of compound inverted lists when we wish to process basic queries instead of inclusive queries. For example, suppose that we have records with five secondary keys K_1, K_2, K_3, K_4, K_5, each of which has four possible values {0, 1, 2, 3}. In order to answer queries about records that have $K_i = a$ and $K_j = b$, given a and b and $i \neq j$, we could form inverted lists for all 160 such queries; each record would then appear in ten of these inverted lists. An alternative is to make use of the following configuration of ordered quintuples based on a combinatorial pattern known as "mutually orthogonal latin squares":

$$\begin{array}{llll} (0,0,0,0,0) & (1,0,1,2,3) & (2,0,2,3,1) & (3,0,3,1,2) \\ (0,1,1,1,1) & (1,1,0,3,2) & (2,1,3,2,0) & (3,1,2,0,3) \\ (0,2,2,2,2) & (1,2,3,0,1) & (2,2,0,1,3) & (3,2,1,3,0) \\ (0,3,3,3,3) & (1,3,2,1,0) & (2,3,1,0,2) & (3,3,0,2,1) \end{array} \tag{12}$$

Here if we look at any two of the five components, we will see all 16 possible ordered pairs of values occurring exactly once in those components. We can associate with block (a, b, c, d, e) of this configuration the records which satisfy at least two of the conditions $K_1 = a$, $K_2 = b$, . . . , $K_5 = e$. In this way each of the 16 blocks will be associated with 376 of the 1024 possible records, so the average redundancy is lowered from 10 to $16 \times 376/1024 = 5\frac{7}{8}$.

The theory of block designs, latin squares, etc. is developed in detail in Marshall Hall, Jr.'s book *Combinatorial Theory* (Waltham, Mass.: Blaisdell, 1967). Although these combinatorial configurations are very beautiful, their only real application to information retrieval so far has been to decrease the redundancy incurred when compound inverted lists are being used; and David K. Chow [*Information and Control* **15** (1969), 377–396] has observed that this type of decrease can be obtained even without using combinatorial designs.

A short history and bibliography. The first published article dealing with a technique for secondary key retrieval was by L. R. Johnson in *CACM* **4** (1961), 218–222. The Multilist system was independently developed by Noah S. Prywes, H. J. Gray, W. I. Landauer, D. Lefkowitz and S. Litwin at about the same time; cf. *IEEE Trans. on Communication and Electronics* **68** (1963), 488–492. Another rather early publication which influenced later work was by D. R. Davis and A. D. Lin, *CACM* **8** (1965), 243–246.

Since then a large literature on the subject has grown up, but most of it deals with the user interface and with programming language considerations, which are not within the scope of this book. In addition to the papers already

cited, the following published articles were found to be most helpful to the author as this section was being written: Jack Minker and Jerome Sable, *Ann. Rev. of Information Science and Technology* **2** (1967), 123–160; Robert E. Bleier, *Proc. ACM Nat. Conf.* **22** (1967), 41–49; Jerome A. Feldman and Paul D. Rovner, *CACM* **12** (1969), 439–449; Burton H. Bloom, *Proc. ACM Nat. Conf.* **24** (1969), 83–95; H. S. Heaps and L. H. Thiel, *Information Storage and Retrieval* **6** (1970), 137–153; Vincent Y. Lum and Huei Ling, *Proc. ACM Nat. Conf.* **26** (1971), 349–356. A good survey of manual card-filing systems appears in *Methods of Information Handling* by C. P. Bourne (New York: Wiley, 1963), Chapter 5. "Balanced filing schemes" were originally developed by C. T. Abraham, S. P. Ghosh and D. K. Ray-Chaudhuri in 1965; perhaps the best summary of this work and its subsequent extensions appears in an article by R. C. Bose and Gary G. Koch, *SIAM J. Appl. Math.* **17** (1969), 1203–1214.

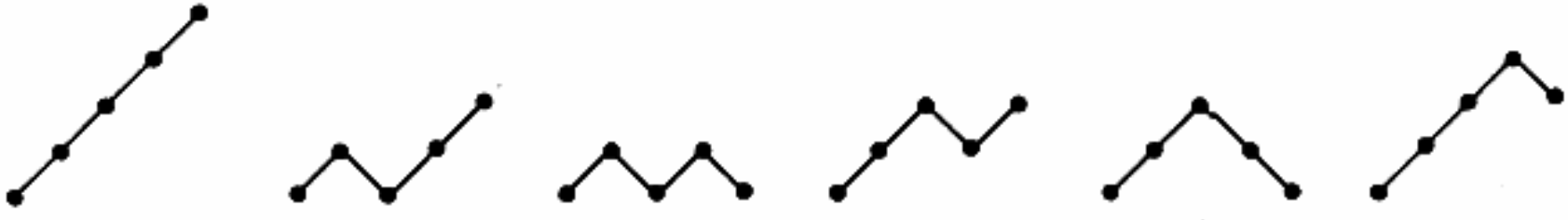
In this section we have discussed a fairly large number of interesting ideas that are helpful in quite different situations. Since many of these techniques were introduced shortly before this section was written, further advances will probably be forthcoming soon.

It is interesting to note that the human brain is much better at secondary key retrieval than computers are; in fact, it is rather easy for us to recognize faces, or melodies, etc., from only fragmentary information, while computers have barely been able to do this at all. Therefore it is not unlikely that a completely new approach to machine design will someday be discovered, which solves the problem of secondary key retrieval once and for all, making this entire section obsolete.

EXERCISES

▶ **1.** [*M27*] Let $0 \le k \le \frac{1}{2}n$. Prove that the following construction produces $\binom{n}{k}$ permutations of $\{1, 2, \ldots, n\}$, such that every t-element subset of $\{1, 2, \ldots, n\}$ appears as the first t elements of at least one of the permutations, for $t \le k$ or $t \ge n - k$: Consider a path in the plane from $(0, 0)$ to (n, r) where $r \ge n - 2k$, in which the ith step is from $(i - 1, j)$ to $(i, j + 1)$ or to $(i, j - 1)$; the latter possibility is allowed only if $j \ge 1$, so that the path never goes below the x-axis. There are exactly $\binom{n}{k}$ such paths. For each path of this kind, a permutation is constructed as follows, using three lists that are initially empty: For $i = 1, 2, \ldots, n$, if the ith step of the path goes up, put the number i into list B; if the step goes down, put i into list A and move the currently largest element of list B into list C. The resulting permutation is equal to the final contents of list A, then list B, then list C, each list in increasing order.

For example, when $n = 4$ and $k = 2$, the six paths and permutations defined by this procedure are



(Vertical lines show the division between lists A, B, and C. These six permutations correspond to the compound attributes in (2).)

Hint: Represent each t-element subset S by a path that goes from $(0, 0)$ to $(n, n - 2t)$, whose ith step runs from $(i - 1, j)$ to $(i, j + 1)$ if $i \notin S$ and to $(i, j - 1)$ if $i \in S$. Convert every such path into an appropriate path having the special form stated above.

2. [*M25*] (Sakti P. Ghosh.) Find the minimum possible length l of a list $r_1 r_2 \ldots r_l$ of references to records, such that the set of all responses to any of the inclusive queries `**1`, `*1*`, `1**`, `*11`, `1*1`, `11*`, `111` on three binary-valued secondary keys will appear in consecutive locations $r_i \ldots r_j$.

3. [*19*] In Table 2, what inclusive queries will cause (a) Old-Fashioned Sugar Cookies, (b) Oatmeal-Date Bars, to be obtained among the false drops?

4. [*M30*] Find exact formulas for the probabilities in (5), assuming that each record has r distinct attributes chosen randomly from among the $\binom{n}{k}$ k-bit codes in an n-bit field and that the query involves q distinct but otherwise random attributes. (Don't be alarmed if the formulas do not simplify.)

5. [*40*] Experiment with various ways to avoid the redundancy of text when using Harrison's technique for substring searching.

▶ **6.** [*M20*] The total number of m-bit basic queries with k bits specified is $s = \binom{m}{k}2^k$. If a combinatorial hashing function like that in (7) converts these queries into $l_1, l_2, \ldots, l_s$ locations, respectively, $L(k) = (l_1 + l_2 + \cdots + l_s)/s$ is the average number of locations per query. [For example, in (7) we have $L(3) = 1.75$.]

Consider now a composite hash function on an $(m_1 + m_2)$-bit field, formed by mapping the first m_1 bits with one hash function and the remaining m_2 with another, where $L_1(k)$ and $L_2(k)$ are the corresponding average numbers of locations per query. Find a formula that expresses $L(k)$, for the composite function, in terms of L_1 and L_2.

7. [*M24*] (R. L. Rivest.) Find the functions $L(k)$, as defined in the previous exercise, for the following combinatorial hash functions:

(a) $m = 3, n = 2$

$$\begin{aligned} 0\ 0\ * &\to 1\\ 1\ *\ 0 &\to 2\\ *\ 1\ 1 &\to 3\\ 1\ 0\ 1 &\to 4\\ 0\ 1\ 0 &\to 4 \end{aligned}$$

(b) $m = 4, n = 2$

$$\begin{aligned} 0\ 0\ *\ * &\to 1\\ *\ 1\ *\ 0 &\to 2\\ *\ 1\ 1\ 1 &\to 3\\ 1\ 0\ 1\ * &\to 3\\ *\ 1\ 0\ 1 &\to 4\\ 1\ 0\ 0\ * &\to 4 \end{aligned}$$

8. [*M47*] Develop further useful classes of combinatorial hash functions like (7).

9. [*M20*] Prove that when $v = 3^n$, the set of all triples of the form

$$\{(a_1 \ldots a_{k-1}\,0\,b_1 \ldots b_{n-k})_3,\ (a_1 \ldots a_{k-1}\,1\,c_1 \ldots c_{n-k})_3,\ (a_1 \ldots a_{k-1}\,2\,d_1 \ldots d_{n-k})_3\},\qquad 1 \le k \le n,$$

forms a Steiner triple system, where the a's, b's, c's, and d's range over all combinations of 0's, 1's, and 2's such that $b_i + c_i + d_i \equiv 0$ (modulo 3).

10. [*M32*] (Thomas P. Kirkman, *Cambridge and Dublin Math. Journal* **2** (1847), 191–204.) Let us say that a Kirkman triple system of order v is an arrangement of

$v+1$ objects $\{x_0, x_1, \ldots, x_v\}$ into triples such that every pair $\{x_i, x_j\}$ for $i \neq j$ occurs in exactly one triple, except that the v pairs $\{x_i, x_{(i+1) \bmod v}\}$ do not ever occur in the same triple, for $0 \leq i < v$. For example,

$$\{x_0, x_2, x_4\},\ \{x_1, x_3, x_4\}$$

is a Kirkman triple system of order 4.

a) Prove that a Kirkman triple system can exist only when $v \bmod 6 = 0$ or 4.

b) Given a Steiner triple system S on v objects $\{x_1, \ldots, x_v\}$, prove that the following construction yields another one S' on $2v+1$ objects and a Kirkman triple system K' of order $2v-2$: The triples of S' are those of S plus

i) $\{x_i, y_j, y_k\}$ where $j+k \equiv i$ (modulo v) and $j < k$, $1 \leq i, j, k \leq v$;

ii) $\{x_i, y_j, z\}$ where $2j \equiv i$ (modulo v), $1 \leq i, j \leq v$.

The triples of K' are those of S' minus all those containing y_1 and/or y_v.

c) Given a Kirkman triple system K on $\{x_0, x_1, \ldots, x_v\}$, where $v = 2u$, prove that the following construction yields a Steiner triple system S' on $2v+1$ objects and a Kirkman triple system K' of order $2v-2$: The triples of S' are those of K plus

i) $\{x_i, x_{(i+1) \bmod v}, y_{i+1}\}$, $0 \leq i < v$;

ii) $\{x_i, y_j, y_k\}$, $j+k \equiv 2i+1$ (modulo $v-1$), $1 \leq j < k-1 \leq v-2$, $1 \leq i \leq v-2$;

iii) $\{x_i, y_j, y_v\}$, $2j \equiv 2i+1$ (modulo $v-1$), $1 \leq j \leq v-1$, $1 \leq i \leq v-2$;

iv) $\{x_0, y_{2j}, y_{2j+1}\}$, $\{x_{v-1}, y_{2j-1}, y_{2j}\}$, $\{x_v, y_j, y_{v-j}\}$, for $1 \leq j < u$;

v) $\{x_v, y_u, y_v\}$.

The triples of K' are those of S' minus all those containing y_1 and/or y_{v-1}.

d) Use the preceding results to prove that Kirkman triple systems of order v exist for all $v \geq 0$ of the form $6k$ or $6k+4$, and Steiner triple systems or v objects exist for all $v \geq 1$ of the form $6k+1$ or $6k+3$.

11. [*M25*] The text describes the use of Steiner triple systems in connection with inclusive queries; in order to extend this to all basic queries it is natural to define the following concept. A *complemented triple system* of order v is an arrangement of $2v$ objects $\{x_1, \ldots, x_v, \bar{x}_1, \ldots, \bar{x}_v\}$ into triples such that every pair of objects occurs together is exactly one triple, except that complementary pairs $\{x_i, \bar{x}_i\}$ never occur together. For example,

$$\{x_1, x_2, x_3\},\ \{x_1, \bar{x}_2, \bar{x}_3\},\ \{\bar{x}_1, x_2, \bar{x}_3\},\ \{\bar{x}_1, \bar{x}_2, x_3\}$$

is a complemented triple system of order three.

Prove that complemented triple systems of order v exist for all $v \geq 0$ not of the form $3k+2$.

12. [*M23*] Continuing exercise 11, construct a complemented *quadruple* system of order 7.

13. [*M25*] Construct quadruple systems with $v = 4^n$ elements, analogous to the triple systems of exercise 9.

14. [*25*] Discuss the problem of deleting nodes from post-office trees like Fig. 45.

ANSWERS TO EXERCISES

"I have answered three questions, and that is enough,"
Said his father. "Don't give yourself airs!
Do you think I can listen all day to such stuff?
Be off, or I'll kick you down-stairs!"

— LEWIS CARROLL (Alice's Adventures in Wonderland, Chapter 5)

NOTES ON THE EXERCISES

1. An average problem for a mathematically inclined reader.

3. See W. J. LeVeque, *Topics in Number Theory* **2** (Reading, Mass.: Addison-Wesley, 1956), Chapter 3. (*Note:* One of the men who read a preliminary draft of the manuscript for this book reported that he had discovered a truly remarkable proof, which the margin of his copy was too small to contain.)

CHAPTER 5

1. Let $p(1) \ldots p(n)$ and $q(1) \ldots q(n)$ be different permutations satisfying the conditions, and let i be minimal with $p(i) \neq q(i)$. Then $p(i) = q(j)$ and $q(i) = p(k)$ for some j, k, i. Since $K_{p(i)} \leq K_{p(k)} = K_{q(i)} \leq K_{q(j)} = K_{p(i)}$ we have $K_{p(i)} = K_{q(i)}$; hence by stability $p(i) < p(k) = q(i) < q(j) = p(i)$, a contradiction.

2. Yes, if the sorting operations were all stable. (If they were not stable we cannot say.) Mr. A and Mr. C certainly have the same result; and so does Mr. B, since the stability shows that equal major keys in his result are accompanied by minor keys in nondecreasing order.

Formally, assume that Mr. B obtains the result $R_{p(1)}, \ldots, R_{p(N)} = R'_1, \ldots, R'_N$ after sorting the minor keys, then $R'_{q(1)}, \ldots, R'_{q(N)} = R_{p(q(1))}, \ldots, R_{p(q(N))}$ after sorting the major keys; we want to show that

$$(K_{p(q(i))}, k_{p(q(i))}) \leq (K_{p(q(i+1))}, k_{p(q(i+1))})$$

for $1 \leq i < N$. If $K_{p(q(i))} \neq K_{p(q(i+1))}$, we have $K_{p(q(i))} < K_{p(q(i+1))}$; and if $K_{p(q(i))} = K_{p(q(i+1))}$, then $K'_{q(i)} = K'_{q(i+1)}$, hence $q(i) < q(i+1)$, hence $k'_{q(i)} \leq k'_{q(i+1)}$, i.e., $k_{p(q(i))} \leq k_{p(q(i+1))}$.

3. We can always bring all records with equal keys together, preserving their relative order, treating these groups of records as a unit in further operations; hence we may assume that all keys are distinct. Let $a < b < c < a$; then we can arrange things so that the first three keys are *abc*, *bca*, or *cab*. Now if $N - 1$ distinct keys can be sorted, so can N; for if $K_1 < \cdots < K_{N-1} > K_N$ we always have either $K_{i-1} < K_N < K_i$ for some i, or $K_N < K_1$.

4. Overflow is possible, and it can lead to a false equality indication. He should have written, e.g., `LDA A; CMPA B` and tested the comparison indicator. (The inability to make full-word comparisons by subtraction is a problem on many computers; it is the chief reason for including `CMPA, . . . , CMPX` in `MIX`'s repertoire.)

5.

```
COMPARE  STJ   9F
1H       LDX   A,1
         CMPX  B,1
         JNE   9F
         DEC1  1
         J1P   1B
9H       JMP   *
```

6. Solution 1, based on the identity $\min(a, b) = \frac{1}{2}(a + b - |a - b|)$:

```
LDA   A
SRAX  5
DIV   =2=
STA   A1     a = 2a1 + a2
STX   A2     |a2| ≤ 1
LDA   B
SRAX  5
DIV   =2=
STA   B1     b = 2b1 + b2
STX   B2     |b2| ≤ 1
LDA   A1
SUB   B1     no overflow possible
STA   AB1    a1 - b1
LDA   A2
SUB   B2
STA   AB2    a2 - b2
SRAX  1
ADD   AB1
ENTX  1
SLAX  5
MUL   AB2
STX   AB3          (a2 - b2) sign (a - b)
LDA   A2
ADD   B2
SUB   AB3
SRAX  5
DIV   =2=
ADD   A1
ADD   B1           no overflow possible
SUB   AB1(1:5)
STA   C
```

Solution 2, based on the fact that indexing can cause interchanges in a tricky way:

```
LDA   A
STA   C
STA   TA
LDA   B
STA   TB
```

Now duplicate the following code k times, where $2^k > 10^{10}$:

```
LDA   TA          LDA   TB          INC1  0,2
SRAX  5           SRAX  5           INC1  0,2
DIV   =2=         DIV   =2=         INC1  0,2
STX   TEMP        STX   TEMP        LD3   TMIN,1
LD1   TEMP        LD2   TEMP        LDA   0,3
STA   TA          STA   TB          STA   C
```

(This scans the binary representations of a and b from right to left.) Finally, the table

```
        HLT
        CON  C      -1  -1
        CON  B       0  -1
        CON  B      +1  -1
        CON  A      -1   0
TMIN    CON  C       0   0
        CON  B       1   0
        CON  A      -1   1
        CON  A       0   1
        CON  C       1   1
```

7. $\sum_j \binom{r+j-1}{j}(-1)^j\binom{N}{r+j}x^{r+j}$, by the method of inclusion and exclusion (exercise 1.3.3–26). This is $r\binom{N}{r}\int_0^x t^{r-1}(1-t)^{N-r}\,dt$, a "beta distribution."

8. Sort it, then count. (Some sorting methods make it convenient to drop records whose keys are duplicated elsewhere in the file.)

9. Assign each person an identification number, which must appear on all forms concerning him. Sort the information forms and the tax forms separately, with this identification number as the key. Denote the sorted tax forms by $R_1, \ldots, R_N$, with keys $K_1 < \cdots < K_N$. (There should be no two tax forms with equal keys.) Add a new $(N+1)$st record whose key is ∞, and set $i \leftarrow 1$. Then, for each record in the information file, check if it has been reported, as follows: Let K denote the key on the information form being processed.

a) If $K > K_i$, increase i by 1 and repeat this step.
b) If $K < K_i$, or if $K = K_i$ and the information is not reflected on tax form R_i, signal an error.

Try to do all this processing without wasting the taxpayers' money.

10. One way is to attach the key (j, i) to the entry $a_{i,j}$ and to sort using lexicographic order, then omit the keys. (A similar idea can be used to obtain any desired reordering of information, when a simple formula for the reordering can be given.)

In the special case considered in this problem, the method of "balanced two-way merge sorting" treats the keys in such a simple manner that it is unnecessary to actually write them explicitly on the tapes. Given an $n \times n$ matrix, we may proceed as follows: First put odd-numbered rows on tape 1, even-numbered rows on tape 2, etc., obtaining

Tape 1: $a_{11}\ a_{12}\ \ldots\ a_{1n}\ a_{31}\ a_{32}\ \ldots\ a_{3n}\ a_{51}\ a_{52}\ \ldots\ a_{5n}\ \ldots$
Tape 2: $a_{21}\ a_{22}\ \ldots\ a_{2n}\ a_{41}\ a_{42}\ \ldots\ a_{4n}\ a_{61}\ a_{62}\ \ldots\ a_{6n}\ \ldots$

Then rewind these tapes, and process them synchronously, to obtain

Tape 3: $a_{11}\ a_{21}\ a_{12}\ a_{22}\ \ldots\ a_{1n}\ a_{2n}\ a_{51}\ a_{61}\ a_{52}\ a_{62}\ \ldots\ a_{5n}\ a_{6n}\ \ldots$
Tape 4: $a_{31}\ a_{41}\ a_{32}\ a_{42}\ \ldots\ a_{3n}\ a_{4n}\ a_{71}\ a_{81}\ a_{72}\ a_{82}\ \ldots\ a_{7n}\ a_{8n}\ \ldots$

Rewind these tapes, and process them synchronously, to obtain

Tape 1: $a_{11}\ a_{21}\ a_{31}\ a_{41}\ a_{12}\ \ldots\ a_{42}\ \ldots\ a_{4n}\ a_{9,1}\ \ldots$
Tape 2: $a_{51}\ a_{61}\ a_{71}\ a_{81}\ a_{52}\ \ldots\ a_{82}\ \ldots\ a_{8n}\ a_{13,1}\ \ldots$

And so on, until the desired transpose is obtained after $\lceil \log_2 n \rceil$ passes.

11. One way is to attach random distinct key values, sort on these keys, then discard the keys. (Cf. exercise 10; a similar method for obtaining a random *sample* was discussed in Section 3.4.2.) Another technique, involving about the same amount of work but apparently not straining the accuracy of the random number generator as much, is to attach a random integer in the range $0 \leq K_i \leq N - i$ to R_i, then rearrange using the technique of exercise 5.1.1–5.

12. For example, prepare 84-character records $a_1\ a_2 \ldots a_{84}$ as follows. For each committee card, $c_1 \ldots c_{80}$, set $a_1a_2a_3 \leftarrow c_{78}c_{79}c_{80}$, $a_4 \leftarrow$ blank, $a_5 \ldots a_{84} \leftarrow c_1 \ldots c_{80}$. For each faculty card $f_1 \ldots f_{80}$, and for each nonblank field $f_{21+3k}f_{22+3k}f_{23+3k}$ on this card for $0 \leq k < 20$, prepare a record with $a_1a_2a_3 \leftarrow f_{21+3k}f_{22+3k}f_{23+3k}$, $a_4 \ldots a_{22} \leftarrow f_2 \ldots f_{20}$, $a_{23} \ldots a_{21+j} \leftarrow f_1 \ldots f_j$, $a_{22+j} \leftarrow$ ",", $a_{24+j} \leftarrow f_{19}$, $a_{25+j} \leftarrow$ ".", $a_{27+j} \leftarrow f_{20}$, and (if f_{20} is nonblank) $a_{28+j} \leftarrow$ "."; other positions blank. Here j is the largest integer ≤ 18 with f_j nonblank. (If blanks are not lowest in the collating sequence, a_4 on the committee records and all blank characters among $a_4 \ldots a_{22}$ on the faculty records should be changed to the character lowest in the collating sequence.)

Now sort these 84-character records alphabetically. After sorting, process each 84-character record in turn as follows: If a_4 is blank, start a new page and print $a_4 \ldots a_{84}$. If a_4 is nonblank, print $a_{23} \ldots a_{84}$ preceded by 19 blanks. (There are obvious refinements to check for committee numbers without names, and faculty members not on enough committees. This exercise illustrates a fairly general method for creating listings, by sorting on a key which is not printed as part of the listing itself. It would be possible to keep the committee records separate from the faculty records, sorting the two files separately and listing them by processing them in a synchronous manner; but the separation probably doesn't save enough time to be worth the extra effort.)

13. With a character-conversion table, you can design a lexicographic comparison routine which simulates the order used on the other machine. Alternatively, you could create artificial keys, different from the actual characters but giving the desired ordering. The latter method has the advantage that it needs to be done only once; but it takes more space and requires conversion of the entire key. The former method can often determine the result of a comparison by converting only one or two letters of the keys; during later stages of sorting, the comparison will be between nearly equal keys, and it will perhaps be advantageous in the former method to check for equality of letters before converting them.

14. For this problem, just run through the file once keeping 50 or so individual counts. But if "city" were substituted for "state," and if the total number of cities were quite large, it would be a good idea to sort on the city name.

15. As in exercise 14, it depends on the size of the problem. If the total number of cross-reference entries fits into high-speed memory, the best approach is probably to use a symbol table algorithm (Chapter 6) with each identifier associated with the head of a list of references. For larger problems, create a file of records, one record for each cross-reference citation to be put in the index, and sort it.

16. Carry along with each card a key which, sorted lexicographically in the usual simple way, will define the desired ordering. This key is to be supplied by library personnel and attached to the card data when it first enters the system. A possible

key, using MIX character code to define the collating sequence, uses the following two-letter codes to separate words from each other:

␣␣	end of key
␣.	end of cross-reference
␣,	end of surname
␣(	hyphen of multiple surname
␣)	end of author name
␣+	end of place name
␣–	end of subject heading
␣*	end of book title
␣/	space between words.

The given example would then come out as follows (showing only the first 25 characters):

```
ACCADEMIA␣/NAZIONALE␣/DEI
ACHTZEHNHUNDERT␣/ZWOLF␣/E
BIBLIOTHEQUE␣/D␣/HISTOIRE
BIBLIOTHEQUE␣/DES␣/CURIOS
BROWN␣,J␣/CROSBY␣)␣␣
BROWN␣,JOHN␣)␣␣
BROWN␣,JOHN␣)MATHEMATICIA
BROWN␣,JOHN␣)OF␣/BOSTON␣␣
BROWN␣,JOHN␣)1715␣␣
BROWN␣,JOHN␣)1715␣–␣␣
BROWN␣,JOHN␣)1761␣␣
BROWN␣,JOHN␣)1810␣␣
BROWN␣(WILLIAMS␣,REGINALD
BROWN␣/AMERICA␣*␣␣
BROWN␣/AND␣/DALLISONS␣/NE
BROWNJOHN␣,ALAN␣)␣␣
DEN␣,VLADIMIR␣/EDUARDOVIC
DEN␣*␣␣
DEN␣/LIEBEN␣/SUSSEN␣/MADE
DIX␣,MORGAN␣)1827␣␣
DIX␣/HUIT␣/CENT␣/DOUZE␣/O
DIX␣/NEUVIEME␣/SIECLE␣/FR
EIGHTEEN␣/FORTY␣/SEVEN␣/I
EIGHTEEN␣/TWELVE␣/OVERTUR
I␣/AM␣/A␣/MATHEMATICIAN␣*
I␣/B␣/M␣/JOURNAL␣/OF␣/RES
I␣/HA␣/EHAD␣*␣␣
IA␣/A␣/LOVE␣/STORY␣*␣␣
INTERNATIONAL␣/BUSINESS␣/
KHUWARIZMI␣,MUHAMMAD␣/IBN
LABOR␣*␣␣
LABOR␣/RESEARCH␣/ASSOCIAT
LABOUR␣.␣␣
MACCALLS␣/COOKBOOK␣*␣
MACCARTHY␣,JOHN␣)1927␣␣
MACHINE␣/INDEPENDENT␣/COM
MACMAHON␣,PERCY␣/ALEXANDE
MISTRESS␣/DALLOWAY␣*␣␣
MISTRESS␣/OF␣/MISTRESSES␣
ROYAL␣/SOCIETY␣/OF␣/LONDO
SAINT␣/PETERSBURGER␣/ZEIT
SAINT␣/SAENS␣,CAMILLE␣)18
SAINTE␣/ANNE␣/DES␣/MONTS␣
SEMINUMERICAL␣/ALGORITHMS
UNCLE␣/TOMS␣/CABIN␣*␣␣
UNITED␣/STATES␣/BUREAU␣/O
VANDERMONDE␣,ALEXANDER␣/T
VANVALKENBURG␣,MAC␣/ELWYN
VONNEUMANN␣,JOHN␣)1903␣␣
WHOLE␣/ART␣/OF␣/LEGERDEMA
WHOS␣/AFRAID␣/OF␣/VIRGINI
WIJNGAARDEN␣,ADRIAAN␣/VAN
```

This auxiliary key should be followed by the card data, so that unequal cards having the same auxiliary key (e.g., Sir John = John) are distinguished properly.

17. Prepare two files, one containing $a^{m^n} \bmod p$ and the other containing $(ba^{-n}) \bmod p$ for $0 \leq n < m$. Sort these files and look for a common entry.

Note: This reduces the work from $O(p)$ to $O(\sqrt{p} \log p)$. Is a further reduction possible? It is not difficult to determine if n is even or odd, in $\log p$ steps, by testing

whether $b^{(p-1)/2} \bmod p = 1$ or $(p-1)$. So we may reduce the problem to the case n even; and $a^{2n} \equiv b$ is equivalent to $a^n \equiv \pm\sqrt{b}$. Unfortunately, there is no apparent way to select the appropriate $\sqrt{b}$ which has $n \leq (p-1)/2$.

18. For example, we can make two files containing values of $(u^6 + v^6 + w^6) \bmod W$ and $(z^6 - x^6 - y^6) \bmod W$ for $u \leq v \leq w$, $x \leq y \leq z$, where W is the word size of our computer. Sort these and look for duplicates, then subject the duplicates to further tests. (Some congruences modulo small primes might also be used to place further restrictions on u, v, w, x, y, z.)

19. In general, to find all pairs of numbers (x_i, x_j) with $x_i + x_j = c$, where c is given: Sort the file so that $x_1 < x_2 < \cdots < x_N$. Set $i \leftarrow 1$, $j \leftarrow N$, and then repeat the following operation until $j \leq i$:

If $x_i + x_j = c$, output (x_i, x_j) and (x_j, x_i), set $i \leftarrow i + 1$, $j \leftarrow j - 1$;

If $x_i + x_j < c$, set $i \leftarrow i + 1$;

If $x_i + x_j > c$, set $j \leftarrow j - 1$.

Finally if $j = i$ and $2x_i = c$, output (x_i, x_i). This process is like those of exercises 16 and 17: We are essentially making two sorted files, one containing $x_1, \ldots, x_N$ and the other containing $c - x_N, \ldots, c - x_1$, and checking for duplicates. But the second file doesn't need to be explicitly formed in this case. Another approach when c is odd is to sort on a key such as (x odd $\Rightarrow x$, x even $\Rightarrow c - x$).

20. Some of the alternatives are: (a) For each of the 499,500 pairs i, j with $1 \leq i < j \leq 1000$, set $y_1 \leftarrow x_i \oplus x_j$, $y_2 \leftarrow y_1 \wedge (y_1 - 1)$, $y_3 \leftarrow y_2 \wedge (y_2 - 1)$; then print (x_i, x_j) if and only if $y_3 = 0$. Here $\oplus$ denotes "exclusive or" and $\wedge$ denotes "and". (b) Create a file with 31000 entries, forming 31 entries from each original word x_i by including x_i and the 30 words that differ from x_i in one position. Sort this file and look for duplicates. (c) Do a test analogous to (a) on

i) all pairs of words which agree in their first 10 bits;

ii) all pairs of words which agree in their middle 10 bits (but not the first 10);

iii) all pairs of words which agree in their last 10 bits (but neither the first nor middle 10).

This involves three sorts of the data, using a specified 10-bit key each time. The expected number of pairs in each of the three cases is less than 500, if the original words are randomly distributed.

21. First prepare a file containing all five-letter English words. (Be sure to consider adding suffixes such as -ED, -ER, -ERS, -S to shorter words.) Now take each five-letter word α and sort its letters into ascending order, obtaining the sorted five-letter sequence α'. Finally sort all pairs (α', α) to bring all anagrams together.

Experiments by Kim D. Gibson in 1967 indicate that the second longest set of anagrams is LEAST, SLATE, STALE, STEAL, TAELS, TALES, TEALS. [The use of a larger dictionary would have enlarged this set by adding ASTEL, a splinter; LEATS, water junctions; TESLA, a unit of magnetic induction. And we might also count Madame de Staël! Cf. H. E. Dudeney, *300 Best Word Puzzles*, ed. by Martin Gardner (N. Y.: Chas. Scribner's Sons, 1968), puzzle 194.]

The first and last sets found were ALBAS, BALAS, BALSA, BASAL and STRUT, STURT, TRUST, respectively. The unexpected sets ADDER, DARED, DREAD and ALGOR, ARGOL, GORAL, LARGO are of possible interest to computer scientists.

A faster way to proceed is to compute $f(\alpha) = (x + a_1)(x + a_2)(x + a_3)(x + a_4)(x + a_5) \bmod m$, where $a_1, \ldots, a_5$ are numerical codes for the individual letters in α, and where m is the computer word size. Here x is any fixed value which may be chosen "at random" before starting the computation. Sorting the file $(f(\alpha), \alpha)$ will bring anagrams together; afterwards when $f(\alpha) = f(\beta)$ we must make sure that we have a true anagram with $\alpha' = \beta'$. The value $f(\alpha)$ can be calculated more rapidly than α', and this method avoids the determination of α' for most of the words α in the file.

Note: A similar technique can be used when we want to bring together all sets of records which have equal n-word keys $(a_1, \ldots, a_n)$. Suppose that we don't care about the order of the file, except that records with equal keys are to be brought together; it is sometimes faster to sort on the one-word key $(a_1 x^{n-1} + a_2 x^{n-2} + \cdots + x_n) \bmod m$ instead of the original n-word key.

22. Find isomorphic invariants of the graphs (i.e., functions which take equal values on isomorphic directed graphs) and sort on these, to separate "obviously nonisomorphic" graphs from each other. Examples of isomorphic invariants: (a) Represent vertex v_i by (a_i, b_i), where a_i is its in-degree and b_i is its out-degree; then sort the pairs (a_i, b_i) into lexicographic order. The resulting file is an isomorphic invariant. (b) Represent an arc from v_i to v_j by (a_i, b_i, a_j, b_j), and sort these quadruples into lexicographic order. (c) Separate the directed graph into connected components (cf. Algorithm 2.3.3E), determine invariants of each component, and sort the components into order of their invariants in some way. See also the discussion in exercise 21.

After sorting the directed graphs on their invariants, it will still in general be necessary to make secondary tests to see whether directed graphs with identical invariants are in fact isomorphic. The invariants are helpful for these tests too. In the case of trees it is possible to find "characteristic" or "canonical" invariants which completely characterize the tree, so that secondary testing is unnecessary [see H. I. Scoins, *Machine Intelligence* **3** (1969), 43–60].

23. One way is to form a file containing all three-person cliques, then transform it into a file containing all four-person cliques, etc.; if there are no large cliques this method will be quite satisfactory. (On the other hand, if there is a clique of size n, there are at least $\binom{n}{k}$ cliques of size k, so this method can blow up even when n is only 20 or so.)

Given a file which lists all $(k - 1)$-person cliques, in the form $(a_1, \ldots, a_{k-1})$ where $a_1 < \cdots < a_{k-1}$, we can find the k-person cliques by (i) creating a new file containing the entries $(b, c, a_1, \ldots, a_{k-2})$ for each pair of $(k - 1)$-person cliques of the respective forms $(a_1, \ldots, a_{k-2}, b)$, $(a_1, \ldots, a_{k-2}, c)$ with $b < c$; (ii) sorting this file on its first two components; (iii) for each entry $(b, c, a_1, \ldots, a_{k-2})$ in this new file which matches a pair (b, c) of acquaintances in the originally given file, output the k-person clique $(a_1, \ldots, a_{k-2}, b, c)$.

24. Make another copy of the input file; sort one copy on the first components and the other on the second. Passing over these files in sequence now allows us to create a new file containing all pairs (x_i, x_{i+2}) for $1 \le i \le N - 2$, and to identify (x_{N-1}, x_N). The pairs $(N - 1, x_{N-1})$ and (N, x_N) should be written on still another file.

The process continues inductively. Assume that file F contains all pairs (x_i, x_{i+t}) for $1 \le i \le N - t$, in random order, and that file G contains all pairs (i, x_i) for $N - t < i \le N$ in order of the second components. Let H be a copy of file F, and sort H by first components, F by second. Now go through F, G, and H, creating two

new files F' and G', as follows. If the current records of files F, G, H are, respectively, (x, x'), (y, y'), (z, z'), then:

i) If $x' = z$, output (x, z') to F' and advance files F and H.

ii) If $x' = y'$, output $(y - t, x)$ to G' and advance files F and G.

iii) If $x' > y'$, advance file G.

iv) If $x' > z$, advance file H.

When file F is exhausted, sort G' by second components and merge G with it; then replace t by $2t$, F by F', G by G'.

Thus t takes the values 2, 4, 8, . . . ; for fixed t we do $O(\log n)$ passes over the data to sort it; hence the total number of passes is $O((\log n)^2)$. Eventually $t \geq N$, so F is empty; then we simply sort G on its *first* components.

This ingenious solution is due to Norman Hardy. One phase could be saved at the expense of some complication, by recognizing x_{N+1-i} on the same pass as x_i is presently identified.

SECTION 5.1.1

1. 2 0 5 2 2 3 0 0 0; 2 7 3 5 4 1 8 6.

2. $b_1 = (m - 1) \bmod n$; $b_{j+1} = (b_j + m - 1) \bmod (n - j)$.

3. $\bar{a}_j = a_{n+1-j}$ (the "reflected" permutation). This idea was used by O. Terquem [*Journ. de Math.* (1) **3** (1838), 559–560] to prove that the average number of inversions in a random permutation is $\frac{1}{2}\binom{n}{2}$.

4. **C1.** Set $x_0 \leftarrow 0$. (It is possible to let x_j share memory with b_j in what follows, for $1 \le j \le n$.)

C2. For $k = n, n - 1, \ldots, 1$ (in this order) do the following: Set $j \leftarrow 0$; then set $j \leftarrow x_j$ exactly b_k times; then set $x_k \leftarrow x_j$ and $x_j \leftarrow k$.

C3. Set $j \leftarrow 0$.

C4. For $k = 1, 2, \ldots, n$ (in this order), do the following: Set $a_k \leftarrow x_j$; then set $j \leftarrow x_j$. ∎

To save memory space, see exercise 5.2–12.

5. Let α be a string $[m_1, n_1] \cdots [m_k, n_k]$ of ordered pairs of nonnegative integers; we write $|\alpha| = k$, the length of α. Let ϵ denote the empty (length 0) string. Consider the binary operation defined recursively on pairs of such strings as follows:

$$\epsilon \circ \alpha = \alpha \circ \epsilon = \alpha;$$

$$([m, n]\alpha) \circ ([m', n']\beta) = \begin{cases} [m, n](\alpha \circ ([m' - m, n']\beta)), & \text{if } m \le m', \\ [m', n'](([m - m' - 1, n]\alpha) \circ \beta), & \text{if } m > m'. \end{cases}$$

It follows that the computation time required to evaluate $\alpha \circ \beta$ is proportional to $|\alpha \circ \beta| = |\alpha| + |\beta|$. Furthermore, we can prove that $\circ$ is associative and that $[b_1, 1] \circ [b_2, 2] \circ \cdots \circ [b_n, n] = [0, a_1][0, a_2] \ldots [0, a_n]$. The expression on the left can be evaluated in $\lceil \log_2 n \rceil$ passes, each pass combining pairs of strings, for a total of $O(n \log n)$ steps.

Example: Starting from (2), we want to evaluate $[2, 1] \circ [3, 2] \circ [6, 3] \circ [4, 4] \circ [0, 5] \circ [2, 6] \circ [2, 7] \circ [1, 8] \circ [0, 9]$. The first pass reduces this to $[2, 1][1, 2] \circ [4, 4][1, 3] \circ [0, 5][2, 6] \circ [1, 8][0, 7] \circ [0, 9]$. The second pass reduces it to $[2, 1][1, 2][1, 4][1, 3] \circ [0, 5][1, 8][0, 6][0, 7] \circ [0, 9]$. The third pass yields

$$[0, 5][1, 1][0, 8][0, 2][0, 6][0, 4][0, 7][0, 3] \circ [0, 9].$$

The fourth pass yields (1).

Motivation: A string such as $[4, 4][1, 3]$ stands for "␣␣␣␣4␣3", where "␣" denotes a large number to be filled in later. Note that, together with exercise 2, we obtain an algorithm for the Josephus problem which is $O(n \log n)$ instead of $O(mn)$, partially answering a question raised in exercise 1.3.2–22.

Another $O(n \log n)$ solution to this problem, using a random-access memory, follows from the use of balanced trees in a straightforward manner.

6. Start with $b_1 = b_2 = \cdots = b_n = 0$. For $k = \lfloor \log_2 n \rfloor, \lfloor \log_2 n \rfloor - 1, \ldots, 0$ do the following: Set $x_s \leftarrow 0$ for $0 \le s \le n/2^{k+1}$; then for $j = 1, 2, \ldots, n$ do the following: Set $r \leftarrow \lfloor a_j/2^k \rfloor \bmod 2$, $s \leftarrow \lfloor a_j/2^{k+1} \rfloor$ (these are essentially bit extractions); if $r = 0$, set $b_{a_j} \leftarrow b_{a_j} + x_s$, and if $r = 1$ set $x_s \leftarrow x_s + 1$.

Another solution appears in exercise 5.2.4–21.

7. $B_j < j$ and $C_j \le n - j$ since a_j has less than j elements to its left and $n - j$ elements to its right. To reconstruct $a_1\, a_2 \ldots a_n$ from $B_1\, B_2 \ldots B_n$, start with the element 1; then for $k = 2, \ldots, n$ add one to each element $> B_k$ and append $(B_k + 1)$ at the right. (Cf. Method 2 in Section 1.2.5.) A similar procedure works for the C's. Alternatively, we could use the result of the following exercise.

8. $b' = C$, $c' = B$, $B' = c$, $C' = b$, since each inversion (a_i, a_j) of $a_1 \ldots a_n$ corresponds to the inversion (j, i) of $a'_j \ldots a'_n$. Some further relations: (a) $c_j = j - 1$ iff $(b_i > b_j$ for all $i < j)$; (b) $b_j = n - j$ iff $(c_i > c_j$ for all $i > j)$; (c) $b_j = 0$ iff $(c_i - i < c_j - j$ for all $i > j)$; (d) $c_j = 0$ iff $(b_i + i < b_j + j$ for all $i < j)$; (e) $b_i \le b_{i+1}$ iff $c_i \ge c_{i+1}$; (f) $a_j = j + C_j - B_j$; $a'_j = j + b_j - c_j$.

9. $b = C = b'$ is equivalent to $a = a'$.

10. $\sqrt{10}$. (One way to coordinatize the truncated octahedron lets the respective vectors $(1, 0, 0)$, $(0, 1, 0)$, $\frac{1}{2}(1, 1, \sqrt{2})$, $\frac{1}{2}(1, -1, \sqrt{2})$, $\frac{1}{2}(-1, 1, \sqrt{2})$, $\frac{1}{2}(-1, -1, \sqrt{2})$ stand for adjacent interchanges of the respective pairs 21, 43, 41, 31, 42, 32. The sum of these vectors gives $(1, 1, 2\sqrt{2})$ as the difference between vertices 4321 and 1234.)

A more symmetric solution is to represent vertex π in *four* dimensions by

$$\sum \{\mathbf{e}_u - \mathbf{e}_v \mid (u, v) \text{ is an inversion of } \pi\},$$

where $\mathbf{e}_1 = (1, 0, 0, 0)$, $\mathbf{e}_2 = (0, 1, 0, 0)$, $\mathbf{e}_3 = (0, 0, 1, 0)$, $\mathbf{e}_4 = (0, 0, 0, 1)$. Thus, $1\ 2\ 3\ 4 \leftrightarrow (0, 0, 0, 0)$; $1\ 2\ 4\ 3 \leftrightarrow (0, 0, -1, 1)$; $\ldots$; $4\ 3\ 2\ 1 \leftrightarrow (-3, -1, 1, 3)$. All points lie on the three-dimensional subspace $\{(w, x, y, z) \mid w + x + y + z = 0\}$; the distance between adjacent vertices is $\sqrt{2}$. Equivalently (cf. exercise 8(f)) we may represent $\pi = a_1 a_2 a_3 a_4$ by the vector (a'_1, a'_2, a'_3, a'_4), where $a'_1 a'_2 a'_3 a'_4$ is the inverse permutation. (This 4-D representation of the truncated octahedron with permutations as coordinates, and the n-dimensional generalization, was discussed by C. Howard Hinton in *The Fourth Dimension* (London, 1904), Chapter 10.)

Replicas of the truncated octahedron will fill three-dimensional space in the "simplest" possible way [see H. Steinhaus, *Mathematical Snapshots* (Oxford, 1960),

200–203; C. S. Smith, *Scientific American* 190 (January, 1954), 58–64]. For illustrations of the 14 "Archimedean" solids (i.e., nonprism polyhedra whose faces are regular polygons), see W. W. Rouse Ball, *Mathematical Recreations and Essays*, rev. by H. S. M. Coxeter (Macmillan, 1939), 129–140; H. Martyn Cundy and A. P. Rollett, *Mathematical Models* (Oxford, 1952), 94–109.

11. (a) Obvious. (b) Construct a directed graph with vertices $(1, 2, \ldots, n)$, arcs from $x \longrightarrow y$ if $x > y$ and $(x, y) \in E$ or $x < y$ and $(y, x) \in \overline{E}$. If there are no oriented cycles, this directed graph can be topologically sorted, and the resulting linear order is the desired permutation. If there is an oriented cycle, the shortest has length three, since there are none of length 1 or 2 and since a longer cycle $a_1 \longrightarrow a_2 \longrightarrow a_3 \longrightarrow a_4 \longrightarrow \cdots \longrightarrow a_1$ can be shortened (either $a_1 \longrightarrow a_3$ or $a_3 \longrightarrow a_1$). But an oriented cycle of length 3 contains two arcs of either E or $\overline{E}$, and proves that E or $\overline{E}$ is not transitive after all.

12. [C. Berge, *Principes de Combinatoire* (Paris, 1968), 114–117.] Suppose that $(a, b) \in \overline{E}$, $(b, c) \in \overline{E}$, $(a, c) \notin \overline{E}$. Then for some $k \geq 1$ we have $a = x_0 > x_1 > \cdots > x_k = c$, where $(x_i, x_{i+1}) \in E(\pi_1) \cup E(\pi_2)$ for $0 \leq i < k$. Consider a counterexample of this type where k is minimal. Since $(a, b) \notin E(\pi_1)$ and $(b, c) \notin E(\pi_1)$, we have $(a, c) \notin E(\pi_1)$, and similarly $(a, c) \notin E(\pi_2)$; hence $k > 1$. But if $x_1 > b$, then $(x_1, b) \in \overline{E}$ contradicts the minimality of k, while $(x_1, b) \in E$ implies that $(a, b) \in E$. Similarly if $x_1 < b$ we find that both $(b, x_1) \in \overline{E}$ and $(b, x_1) \in E$ are impossible.

13. For $0 \leq t < m$, the quantity $\sum_{k \bmod m = t} I_n(k)$ is the coefficient of z^{mN+t} in $G_n(z)/(1 - z^m)$ for all large N. The latter function is $1/(1 - z)$ times the polynomial

$$(1 + z) \ldots (1 + z + \cdots + z^{n-1})/(1 + z + \cdots + z^{m-1}),$$

and the desired quantity is the sum of the coefficients of the latter polynomial.

14. The hinted construction takes pairs of distinct-part partitions into each other, except in the two cases $j = k = p_k$ and $j = k = p_k - 1$. In the exceptional cases, n is $(2j - 1) + \cdots + j = (3j^2 - j)/2$ and $(2j) + \cdots + (j + 1) = (3j^2 + j)/2$, respectively, and there is a unique unpaired partition with j parts. [Euler's original proof, in *Novi Comment. acad. sc. Pet.* **5** (1754), 75–83, was also very interesting. He showed by simple manipulations that the infinite product equals s_1, where $s_n = 1 - z^{2n-1} - z^{3n-1}s_{n+1}$ for $n \geq 1$.]

15. Transpose the dot diagram, to go from the p's to the P's. The generating function for the P's is easily obtained, since we first choose any number of 1's (generating function $1/(1 - z)$), then independently choose any number of 2's (generating function $(1 - z^2)$), . . . , finally any number of n's.

16. The coefficient of $z^n q^m$ in the first identity is the number of partitions of m into at most n parts. In the second identity it is the number of partitions of m into n distinct nonnegative parts, i.e., $m = p_1 + p_2 + \cdots + p_n$, where $p_1 > p_2 > \cdots > p_n \geq 0$. This is the same as $m - \binom{n}{2} = q_1 + q_2 + \cdots + q_n$, where $q_1 \geq q_2 \geq \cdots \geq q_n \geq 0$, under the correspondence $q_i = p_i - n + i$. [*Commentarii academiæ scientiarum Petropolitanæ* **13** (1741), 64–93.]

17.

0 0 0 0	0 0 1 0	0 1 0 0	0 0 0 1
1 0 1 1	1 0 2 1	1 2 0 1	2 1 0 1
0 1 0 1	0 1 1 0	0 2 1 0	2 0 1 0
1 1 0 1	1 1 1 0	1 2 1 0	2 1 1 0
1 0 0 1	1 0 1 0	1 1 0 0	2 1 0 0
2 1 0 2	2 1 2 0	2 2 1 0	3 2 1 0

18. Let $q = 1 - p$. The sum $\sum \Pr(\alpha)$ over all instances α of inversions may be evaluated by summing on k, where $0 \le k < n$ is the exact number of leftmost bit positions in which there is equality between i and j as well as between X_i and X_j, in an inversion $X_i \oplus i > X_j \oplus j$ for $i < j$. In this way we obtain the formula $\sum_{0 \le k < n} 2^k (p^2 + q^2)^k (p^2 2^{n-k-1} 2^{n-k-1} + 2pq 2^{n-k-1}(2^{n-k-1} - 1))$; summing and simplifying yields $2^{n-1}(p(2-p)(2^n - (p^2+q^2)^n)/(2 - p^2 - q^2) + (p^2+q^2)^n - 1)$.

19. [*Proc. Amer. Math. Soc.* **19** (1968), 236–240.] Assume that $n > 1$, and that a permutation $a_1 \ldots a_n$ is given. By induction we may assume that $a_1 \ldots a_{n-1}$ corresponds to $b_1 \ldots b_{n-1}$, where index $(a_1 \ldots a_{n-1})$ = inversions $(b_1 \ldots b_{n-1})$ and $a_{n-1} = b_{n-1}$. *Case 1*, $a_{n-1} < a_n$. Let $b_1 \ldots b_{n-1} = \alpha_1 x_1 \ldots \alpha_r x_r$, where $\alpha_1, \ldots, \alpha_r$ are (possibly empty) strings of elements $> a_n$, and where $x_1, \ldots, x_r$ are the elements $< a_n$. Then we take $a_1 \ldots a_n$ into $x_1 \alpha_1 \ldots x_r \alpha_r a_n$. *Case 2*, $a_{n-1} > a_n$. Same as Case 1, with "$<$" and "$>$" interchanged. This transformation is clearly reversible, since we can distinguish Case 1 from Case 2 by comparing the first and last elements. *Example:* 2 7 1 8 3 6 4 is taken into 7 2 8 1 6 3 4; hence 2 7 1 8 3 6 4 5 is left unchanged.

20. See E. M. Wright, *J. London Math. Soc.* **40** (1965), 55–57; and J. Zolnowsky (to appear).

SECTION 5.1.2

1. False (because of a reasonably important technicality). If you said "true," you probably didn't know the definition of $M_1 \cup M_2$ given in Section 4.6.3, which has the property that $M_1 \cup M_2$ is a set whenever M_1 and M_2 are sets. Actually, $\alpha \intercal \beta$ is a permutation of $M_1 \uplus M_2$.

2. $b\ c\ a\ d\ d\ a\ d\ a\ d\ b$.

3. Certainly not, since we may have $\alpha = \beta$. (The unique factorization theorem shows that there aren't too many possibilities, however.)

4. $(d) \intercal (b\ c\ d) \intercal (b\ b\ c\ a\ d) \intercal (b\ a\ b\ c\ d) \intercal (d)$.

5. The number of occurrences of the pair $\ldots xx \ldots$ is always either equal to or one less than the number of $\frac{x}{x}$ columns. When x is the smallest element, the numbers of occurrences are equal iff x is not first in the permutation.

6. Counting the associated number of two-line arrays is easy: $\binom{m}{k}\binom{n}{k}$.

7. Using part (a) of Theorem B, a derivation like that of (20) gives

$$\binom{A-1}{A-k-m-1}\binom{B}{m}\binom{C}{k}\binom{B+k}{B-l}\binom{C-k}{l};$$

$$\binom{A-1}{A-k-m}\binom{B}{m}\binom{C}{k}\binom{B+k-1}{B-l-1}\binom{C-k}{l};$$

$$\binom{A-1}{A-k-m}\binom{B}{m}\binom{C}{k}\binom{B+k-1}{B-l}\binom{C-k}{l}.$$

8. The complete factorization into primes is $(d) \intercal (b\ c\ d) \intercal (b) \intercal (a\ d\ b\ c) \intercal (a\ b) \intercal (b\ c\ d) \intercal (d)$, which is unique since no adjacent pairs commute. So there are eight solutions, with $\alpha = \epsilon$, (d), $(d) \intercal (b\ c\ d)$, $\ldots$.

10. False, but true in interesting cases. Given any linear ordering of the primes, there is at least one factorization of the stated form, since whenever the condition is

violated we can make an interchange which reduces the number of "inversions" in the factorization. So the condition fails only because some permutations have more than one such factorization.

Let $\rho \sim \sigma$ mean that ρ commutes with σ. The following condition is necessary and sufficient for the uniqueness of the factorization as stated:

$$\rho \sim \sigma \sim \tau \qquad \text{and} \qquad \rho \prec \sigma \prec \tau \qquad \text{implies} \qquad \rho \sim \tau.$$

Proof. If $\rho \sim \sigma \sim \tau$ and $\rho \prec \sigma \prec \tau$ and $\rho \not\sim \tau$, we would have two factorizations $\sigma \mathbin{⟙} \tau \mathbin{⟙} \rho = \tau \mathbin{⟙} \rho \mathbin{⟙} \sigma$; hence the condition is necessary. Conversely, to show that it is sufficient for uniqueness, let $\rho_1 \mathbin{⟙} \cdots \mathbin{⟙} \rho_n = \sigma_1 \mathbin{⟙} \cdots \mathbin{⟙} \sigma_n$ be two distinct factorizations satisfying the condition. We may assume that $\sigma_1 \prec \rho_1$, and hence $\sigma_1 = \rho_k$ for some $k > 1$; furthermore $\sigma_1 \sim \rho_j$ for $1 \leq j < k$. Since $\rho_{k-1} \sim \sigma_1 = \rho_k$, we have $\rho_{k-1} \prec \sigma_1$; hence $k > 2$. Let j be such that $\sigma_1 \prec \rho_j$ and $\rho_i \prec \sigma_1$ for $j < i < k$. Then $\rho_{j+1} \sim \sigma_1 \sim \rho_j$ and $\rho_{j+1} \prec \sigma_1 \prec \rho_j$ implies that $\rho_{j+1} \sim \rho_j$; hence $\rho_j \prec \rho_{j+1}$, a contradiction.

Therefore if we are given an ordering relation on a set S of primes, satisfying the above condition, and if we know that all prime factors of a permutation π belong to S, we can conclude that π has a unique factorization of the stated type. Such a condition holds, for example, when S is the set of cycles in (29).

But the set of *all* primes cannot be so ordered; for if we have, say, $(a\ b) \prec (d\ e)$, then we are forced to define

$$(a\ b) \prec (d\ e) \succ (b\ c) \prec (e\ a) \succ (c\ d) \prec (a\ b) \succ (d\ e),$$

a contradiction. (See also the following exercise.)

11. We wish to show that, if $p(1) \ldots p(t)$ is a permutation of $(1, \ldots, t)$, the permutation $x_{p(1)} \ldots x_{p(t)}$ is topologically sorted iff $\sigma_{p(1)} \mathbin{⟙} \cdots \mathbin{⟙} \sigma_{p(t)} = \sigma_1 \mathbin{⟙} \cdots \mathbin{⟙} \sigma_t$; and that if $x_{p(1)} \ldots x_{p(t)}$ and $x_{q(1)} \ldots x_{q(t)}$ are distinct topological sortings, we have $\sigma_{p(j)} \neq \sigma_{q(j)}$ for some j. The first property follows by observing that $x_{p(1)}$ can be first in a topological sort iff $\sigma_{p(1)}$ commutes with (yet is distinct from) $\sigma_{p(1)-1}, \ldots, \sigma_1$; and this condition implies that $\sigma_{p(2)} \mathbin{⟙} \cdots \mathbin{⟙} \sigma_{p(t)} = \sigma_1 \mathbin{⟙} \cdots \mathbin{⟙} \sigma_{p(1)-1} \mathbin{⟙} \sigma_{p(1)+1} \mathbin{⟙} \cdots \mathbin{⟙} \sigma_t$, so induction can be used. The second property follows because if j is minimal with $p(j) \neq q(j)$, we have, say, $p(j) < q(j)$ and $x_{p(j)} \not\prec x_{q(j)}$ by definition of topological sorting; hence $\sigma_{p(j)}$ has no letters in common with $\sigma_{q(j)}$.

To get an arbitrary partial ordering, let the cycle σ_k consist of all ordered pairs (i, j) such that $x_i \prec x_j$ and either $i = k$ or $j = k$; these ordered pairs are to appear in some arbitrary order as individual elements of the cycle. Thus the cycles for the partial ordering $x_1 \prec x_3$, $x_2 \prec x_4$, $x_1 \prec x_4$ would be $\sigma_1 = \big((1, 3)(1, 4)\big)$, $\sigma_2 = \big((2, 4)\big)$, $\sigma_3 = \big((1, 3)\big)$, $\sigma_4 = \big((2, 4)(1, 4)\big)$.

12. No other cycles can be formed, since, for example, the original permutation contains no $\binom{a}{c}$ columns. If $(a\ b\ c\ d)$ occurs s times, then $(a\ b)$ must occur $A - r - s$ times, since there are $A - r$ columns $\binom{a}{b}$, and only two kinds of cycles contribute to such columns.

13. In the two-line notation, first place $A - t$ columns of the form $\binom{d}{a}$, then put the other t a's in the second line, then place the b's, and finally the remaining letters.

14. Since the elements below any given letter in the two-line notation for π^{-1} are

in nondecreasing order, we do not always have $(\pi^{-1})^{-1} = \pi$; but it is true that $((\pi^{-1})^{-1})^{-1} = \pi^{-1}$. In fact, the identity

$$(\alpha \intercal \beta)^{-1} = ((\alpha^{-1} \intercal \beta^{-1})^{-1})^{-1}$$

holds for all α, β.

Given a multiset whose distinct letters are $x_1 < \cdots < x_m$, we can characterize its self-inverse permutations by observing that they each have a unique prime factorization of the form $\beta_1 \intercal \cdots \intercal \beta_m$, where β_j has zero or more prime factors $(x_j) \intercal \cdots \intercal (x_j) \intercal (x_j x_{k_1}) \intercal \cdots \intercal (x_j x_{k_t})$, $j < k_1 \le \cdots \le k_t$. For example, $(a) \intercal (a\ b) \intercal (a\ b) \intercal (b\ c) \intercal (c)$ is a self-inverse permutation. The number of self-inverse permutations of $(m \cdot a, n \cdot b)$ is therefore $\min(m, n) + 1$; of $(\ell \cdot a, m \cdot b, n \cdot c)$ it is the number of solutions of the inequalities $x + y \le \ell$, $y + z \le m$, $z + x \le n$ in nonnegative integers x, y, z. The number of self-inverse permutations of a *set* is considered in Section 5.1.4.

The number of permutations of $(n_1 \cdot x_1, \ldots, n_m \cdot x_m)$ having n_{ij} occurrences of x_i *over* x_j in their two-line notation is $\prod_i n_i! / \prod_{i,j} n_{ij}!$, the same as the number having n_{ij} occurrences of x_i *under* x_j in the two-line notation. Hence there ought to be another way to define the inverse of a multiset permutation, by giving an appropriate one-to-one correspondence.

15. See Theorem 2.3.4.2D. Removing one arc of the directed graph must leave an oriented tree.

16. If the multiset is $(n_1 \cdot x_1, n_2 \cdot x_2, \ldots)$, with $x_1 < x_2 < \cdots$, the inversion table entries for the x_j's must have the form $b_{j1} \le \cdots \le b_{jn_j}$ where b_{jn_j} (the number of inversions of the rightmost x_j) is at most $n_{j+1} + n_{j+2} + \cdots$. So the generating function for the jth part of the inversion table is the generating function for partitions into at most n_j parts, no part exceeding $n_{j+1} + n_{j+2} + \cdots$. The generating function for partitions into at most m parts, no part exceeding n, is the z-binomial coefficient $\binom{m+n}{m}_z$; this is readily proved by induction, and it can also be proved by means of an ingenious construction due to Sylvester [*Amer. J. Math.* **5** (1882), 268–269], or another ingenious construction due to Pólya [*Elemente der Mathematik* **26** (1971), 102–109]. Multiplying the generating functions for $j = 1, 2, \ldots$ gives the desired result.

17. Let $h_n(z) = (n!_z)/n!(1 - z)^n$; the desired generating function is $g(z) = h_n(z)/h_{n_1}(z)h_{n_2}(z)\cdots$. The mean of $h_n(z)$ is $\frac{1}{2}\binom{n}{2}$, by Eq. 5.1.1–12, so the mean of g is

$$\frac{1}{2}\left(\binom{n}{2} - \binom{n_1}{2} - \binom{n_2}{2} - \cdots\right) = \frac{1}{4}\left(n^2 - n_1^2 - n_2^2 - \cdots\right) = \frac{1}{2}\sum_{i<j} n_i n_j.$$

The variance is, similarly,

$$\tfrac{1}{72}(n(n-1)(2n+5) - n_1(n_1-1)(2n_1+5) - \cdots) \\ = \tfrac{1}{36}(n^3 - n_1^3 - n_2^3 - \cdots) + \tfrac{1}{24}(n^2 - n_1^2 - n_2^2 - \cdots).$$

18. Yes. The construction of exercise 5.1.1–17 can be extended in a straightforward way. Alternatively we can generalize the proof given in Section 5.2.1, constructing a one-to-one correspondence between m-tuples $(q_1, \ldots, q_m)$ where q_j is a multiset containing n_j nonnegative integers, on the one hand, and ordered pairs of n-tuples $((a_1, \ldots, a_n), (p_1, \ldots, p_n))$ on the other hand, where $a_1 \ldots a_n$ is a permutation of

$(n_1 \cdot 1, \ldots, n_m \cdot m)$, and $p_1 \geq \cdots \geq p_n \geq 0$. This correspondence is defined as before, inserting the elements of q_j in nonincreasing order; it satisfies the condition

$$\left(\sum q_1\right) + \cdots + \left(\sum q_m\right) = J(a_1 \ldots a_n) + (p_1 + \cdots + p_n)$$

where $\sum q_j$ is the sum of the elements of q_j. [For a further generalization of the technique used in this proof (and in the derivation of Eq. (8) in Section 5.1.3) see D. E. Knuth, *Math. Comp.* **24** (1970), 955–961. See also the comprehensive treatment by Richard P. Stanley, *Memoirs Amer. Math. Soc.* **119** (1972), 104 pp.]

19. (a) Let $S = \{\sigma | \sigma \text{ is prime, } \sigma \text{ is a left factor of } \pi\}$. If S has k elements, the left factors λ of π such that $\mu(\lambda) \neq 0$ are precisely the 2^k intercalations of the subsets of S (see the proof of Theorem C); hence $\sum \mu(\lambda) = \prod_{\sigma \in S} (1 + \mu(\sigma)) = 0$, since $\mu(\sigma) = -1$ and S is nonempty. (b) Clearly $\epsilon(i_1 \ldots i_n) = \mu(\pi) = 0$ if $i_j = i_k$ for some $j \neq k$. Otherwise $\epsilon(i_1 \ldots i_n) = (-1)^r$ where $i_1 \ldots i_n$ has r inversions; this is $(-1)^s$, where $i_1 \cdots i_n$ has s even cycles; and this is $(-1)^{n+t}$ where $i_1 \ldots i_n$ has t cycles.

20. (a) Obvious, by definition of intercalation. (b) By definition,

$$\det (b_{ij}) = \sum_{1 \leq i_1, \ldots, i_m \leq m} \epsilon(i_1 \ldots i_m) b_{1 i_1} \ldots b_{m i_m}.$$

Setting $b_{ij} = \delta_{ij} - a_{ij} x_j$, this becomes

$$\sum_{n \geq 0} \sum_{1 \leq i_1, \ldots, i_n < m} (-1)^n x_{i_1} \ldots x_{i_n} \mu(x_{i_1} \ldots x_{i_n}) \nu(x_{i_1} \ldots x_{i_n}),$$

since $\mu(\pi)$ is usually zero.

(c) Use exercise 19(a) to show that $D \intercal G = 1$ when we regard the products of x's as permutations of noncommutative variables, using the natural algebraic convention $(\alpha + \beta) \intercal \pi = \alpha \intercal \pi + \beta \intercal \pi$.

SECTION 5.1.3

1. We must only show that this value makes (11) valid for $x = k$, when $k \geq 1$. The formula becomes

$$k^n = \sum_{0\leq j\leq r\leq k} (-1)^j (r-j)^n \binom{n+1}{j}\binom{n+k-r}{n}$$

$$= \sum_{0\leq s\leq k} s^n \sum_{0\leq j\leq k-s} (-1)^j \binom{n+1}{j}\binom{n+k-s-j}{n}.$$

For $s < k$, the sum on j can be extended to the range $0 \leq j \leq n+1$, and it is zero (the $(n+1)$st difference of an nth degree polynomial in j).

2. (a) The number of sequences $a_1\, a_2 \ldots a_n$ containing each of the elements $(1, 2, \ldots, q)$ at least once is $\left\{{n \atop q}\right\} q!$ (cf. exercise 1.2.6–64); the number of such sequences satisfying the analog of (10), for $m = q$, is $\binom{n-k}{n-q}$, since we must choose $n - q$ of the possible $=$ signs. (b) Add the results of (a) for $m = n - q$ and $n - q - 1$.

3. By (20),

$$\sum \frac{x^n}{n!} \sum \left\langle {n \atop k} \right\rangle (-1)^k = \sum g_n(-1)x^n = -2/(e^{-2x}+1)$$
$$= -\frac{1}{x}\left(\frac{(-2x)}{e^{-2x}-1} - \frac{(-4x)}{e^{-4x}-1}\right)$$
$$= -\frac{1}{x}\sum_{n\geq 0} \frac{B_n x^n}{n!}\left((-2)^n - (-4)^n\right);$$

hence the result is $(-1)^n B_{n+1} 2^{n+1}(2^{n+1}-1)/(n+1)$. Alternatively, since $-2/(e^{-2x}+1) = -(1+\tanh x)$, we can express the answer as $(-1)^{(n+1)/2} T_n$, when n is odd, where T_n denotes the tangent number defined by the formula

$$\tan z = T_1 z + T_3 z^3/3! + T_5 z^5/5! + \cdots.$$

When $n > 0$ is even, the sum obviously vanishes, by (7).

Note that, by (18), we get the curious identity

$$\sum_k \left\{ {n \atop k} \right\} k!(-\tfrac{1}{2})^k = 2B_{n+1}(1-2^{n+1})/(n+1).$$

4. $(-1)^{n+m}\left\langle {n \atop m+1} \right\rangle$. (Consider the coefficient of z^{m+1} in (18).)

5. First we find that $\left\langle {p-1 \atop k} \right\rangle \bmod p = 1$, for $1 \leq k < p$, by formula (13). Hence by the defining recurrence (2), $\left\langle {p \atop k} \right\rangle \bmod p = 1$ for $1 \leq k \leq p$.

6. Summing first on k is not allowed, because the terms are nonzero for arbitrarily large j and k, and the sum of the absolute values is infinite.

For a simpler example of the fallacy, let $a_{jk} = 0$ when $|j-k| \neq 1$, and let $a_{j(j+1)} = +1$, $a_{(j+1)j} = -1$. Then $\sum_{j\geq 0}(\sum_{k\geq 0} a_{jk}) = \sum_{j\geq 0}(\delta_{j0}) = +1$, while $\sum_{k\geq 0}(\sum_{j\geq 0} a_{jk}) = \sum_{k\geq 0}(-\delta_{k0}) = -1$.

7. Yes. [David and Barton, *Combinatorial Chance* (1962), 150–154.]

8. [*Combinatory Analysis* **1** (1915), 190.] By inclusion and exclusion. For example, $1/(l_1+l_2)!l_3!(l_4+l_5+l_6)!$ is the probability that $x_1 < \cdots < x_{l_1+l_2}$, $x_{l_1+l_2+1} < \cdots < x_{l_1+l_2+l_3}$, and $x_{l_1+l_2+l_3+1} < \cdots < x_{\Sigma l}$.

9. $p_{km} = q_{km} - q_{k(m+1)}$ in (23). Since $\sum_{k,m} q_{km} z^m x^k = g(x,z)/(1-x)$, we have $h(z,x) = \sum h_k(z)x^k = g(x,z)/(1-x) - \big(g(x,z) - g(x,0)\big)/z(1-x) = (z-1)x/(ze^{(x-1)z} - x) + x/z(1-x)$. Thus $h_1(z) = e^z - (e^z-1)/z$; $h_2(z) = (e^{2z} - ze^z) + e^z - (e^{2z}-1)/z$.

10. Let $M_n = L_1 + \cdots + L_n$ be the mean; then $\sum M_n x^n = h'(1,x)$, where the derivative is taken with respect to z, and this is $x/(e^{x-1} - x) - x/(1-x) = M(x)$, say. By the residue theorem,

$$\frac{1}{2\pi i}\oint M(z) z^{-n-1}\,dz = M_n - 2(n+\tfrac{1}{3}) + 1 + \frac{z_1^{-n}}{z_1 - 1} + \frac{\bar{z}_1^{-n}}{\bar{z}_1 - 1},$$

if we integrate around a circle of radius r where $|z_1| < r < |z_2|$. (Note the double pole at $z = 1$.) Furthermore, the absolute value of this integral is $< \oint |M(z)| r^{-n-1}\,dz = O(r^{-n})$. Integrating over larger and larger circles gives the convergent series $M_n = 2n - \frac{1}{3} + \sum_{k\geq 1} 2\,\Re\,(1/z_k^n(1-z_k))$.

To determine the variance, we have $h''(1, x) = -2h'(1, x) + 2x(x-1)e^{x-1}/(e^{x-1} - x)^2$. An argument similar to that used for the mean, this time with a triple pole, shows that the coefficients of $h''(1, x)$ are asymptotically $4n^2 + \frac{4}{3}n - 2M_n$ plus smaller terms; this leads to the asymptotic formula $\frac{2}{3}n + \frac{2}{9}$ (plus exponentially smaller terms) for the variance.

11. $P_{kn} = \sum_{t_1 \geq 1, \ldots, t_k \geq 1} D(t_1, \ldots, t_{k-1}, n, 1)$, where $D(l_1, l_2, \ldots, l_k)$ is MacMahon's determinant of exercise 8. Evaluating this determinant by its first row, we find $P_{kn} = c_0 P_{(k-1)n} + c_1 P_{(k-2)n} + \cdots + c_{k-2} P_{1n} - E_k(n)$, where c_j and E_k are defined as follows:

$$c_j = (-1)^j \sum_{t_1, \ldots, t_{j+1} \geq 1} \frac{1}{(t_1 + \cdots + t_{j+1})!} = (-1)^j \sum_{m \geq 0} \binom{m}{j} \frac{1}{(m+1)!}$$

$$= (-1)^j \sum_{r, m \geq 0} \binom{-1}{j-r} \binom{m+1}{r} \frac{1}{(m+1)!} = -1 + e\left(\frac{1}{0!} - \frac{1}{1!} + \cdots + (-1)^k \frac{1}{k!}\right).$$

$$E_1(n) = -1/n! + 1/(n+1)!; \qquad E_2(n) = 1/(n+1)!;$$

$$E_k(n) = (-1)^k \sum_{m \geq 0} \binom{m}{k-3} \frac{1}{(n+2+m)!}, \qquad k \geq 3.$$

Let $P_{0n} = 0$, $C(z) = \sum c_j z^j = (e^{1-z} - 1)/(1-z)$, and let

$$E(z, x) = \sum_{n,k} E_{k+1}(n) z^n x^k$$

$$= 1 - e^z + \frac{(e^{1-x} - 1)x^2}{(1-x)^2} - \frac{x^2(e^{1-x} - e^z)}{(1-x)(1-x-z)} + \frac{e^z - 1 - z}{z(1-x)}.$$

The recurrence relation we have derived is equivalent to the formula $C(x)H(z, x) = H(z, x)/x + E(z, x)$; hence $H(z, x) = E(z, x)x(1-x)/(xe^{1-x} - 1)$. Expanding this power series gives $H_1(z) = h_1(z)$ (see exercise 9); $H_2(z) = eh_1(z) + 1 - e^z$.

[*Note:* The generating functions for the first three runs were derived by Knuth, *CACM* **6** (1963), 685–688. Barton and Mallows, *Ann. Math. Statistics* **36** (1965), 249, stated the formula $1 - H_{n+1}(z) = (1 - H_n(z))/(1-z) - L_n h_1(z)$, together with (25). Another way to attack this problem is illustrated in exercise 23. Because adjacent runs are not independent, there is no simple relation between the problem solved here and the simpler (probably more useful) result of exercise 9.]

12. [*Combinatory Analysis* **1** (1915), 209–211.] The number of ways to put the multiset into t distinguishable boxes is

$$N_t = \binom{t + n_1 - 1}{n_1} \binom{t + n_2 - 1}{n_2} \cdots \binom{t + n_m - 1}{n_m},$$

since there are $\binom{t+n_1-1}{n_1}$ ways to place the 1's, etc. If we require that no box be empty, the method of inclusion and exclusion tells us that the number of ways is

$$M_t = N_t - \binom{t}{1} N_{t-1} + \binom{t}{2} N_{t-2} - \cdots.$$

Let P_k be the number of permutations having k runs; if we put $k-1$ vertical lines between the runs, and $t-k$ additional vertical lines in any of the $n-k$ remaining places, we get one of the M_t ways to divide the multiset into t nonempty distinguishable parts. Hence

$$M_t = P_t + \binom{n-t+1}{1} P_{t-1} + \binom{n-t+2}{2} P_{t-2} + \cdots .$$

Equating the two values of M_t allows us to determine $P_1, P_2, \ldots$ successively in terms of $N_1, N_2, \ldots$. (A more direct proof would be desirable.)

13. $1 + \frac{1}{2}13 \times 3 = 20.5$.

14. By Foata's correspondence the given permutation corresponds to

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 \\ 3 & 1 & 1 & 2 & 3 & 4 & 3 & 2 & 1 & 1 & 3 & 4 & 2 & 2 & 4 & 4 \end{pmatrix};$$

by (33) this corresponds to

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 \\ 2 & 4 & 4 & 3 & 3 & 3 & 1 & 1 & 4 & 4 & 2 & 1 & 2 & 1 & 2 & 3 \end{pmatrix};$$

and this corresponds to 2 3 4 2 3 4 1 4 2 1 4 3 2 1 3 1 with 9 runs.

15. The number of alternating runs is the number of j such that $1 < j < n$ and either $a_{j-1} < a_j > a_{j+1}$ or $a_{j-1} > a_j < a_{j+1}$. For fixed j, the probability is $\frac{2}{3}$; hence the average, for $n \geq 2$, is $1 + \frac{2}{3}(n-2)$.

16. Each permutation on $\{1, 2, \ldots, n-1\}$, having k alternating runs, yields k permutations with k such runs, 2 with $k+1$, and $n-k-2$ with $k+2$, when the new element n is inserted in all possible places. Hence

$$\left\langle\!\left\langle {n \atop k} \right\rangle\!\right\rangle = k \left\langle\!\left\langle {n-1 \atop k} \right\rangle\!\right\rangle + 2 \left\langle\!\left\langle {n-1 \atop k-1} \right\rangle\!\right\rangle + (n-k) \left\langle\!\left\langle {n-1 \atop k-2} \right\rangle\!\right\rangle .$$

It is convenient to let $\langle\langle {1 \atop k} \rangle\rangle = \delta_{k0}$, $G_1(z) = 1$. Then

$$G_n(z) = \frac{z}{n}\left((1-z^2)G'_{n-1}(z) + (2+(n-2)z)G_{n-1}(z)\right).$$

Differentiation leads to the recurrence

$$x_n = \frac{1}{n}\left(x_{n-1}(n-2) + 2n - 2\right)$$

for $x_n = G'_n(1)$, and this has the solution $x_n = \frac{2}{3}n - \frac{1}{3}$ for $n \geq 2$. Another differentiation leads to the recurrence

$$y_n = \frac{1}{n}\left(y_{n-1}(n-4) + \tfrac{8}{3}n^2 - \tfrac{26}{3}n + 6\right)$$

for $y_n = G''_n(1)$. Set $y_n = \alpha n^2 + \beta n + \gamma$ and solve for α, β, γ to get $y_n = \frac{4}{9}n^2 - \frac{14}{15}n + \frac{11}{90}$ for $n \geq 4$. Hence var $(g_n) = \frac{1}{90}(16n - 29)$, $n \geq 4$.

These formulas for the mean and variance are due to J. Bienaymé, who stated them without proof [*Bull. Soc. Math. de France* **2** (1874), 153–154; *Comptes Rendus* **81** (Acad. Sciences, Paris, 1875), 417–423, see also Bertrand's remarks on p. 458]. The recurrence relation for $\langle\langle {n \atop k} \rangle\rangle$ is due to D. André [*Comptes Rendus* **97** (Acad. Sciences, Paris, 1883), 1356–1358; *Annales scientifiques de l'Ecole normale supérieure* (3) **1** (Paris, 1884), 121–134]. André noted that $g_n(-1) = 0$ for $n \geq 4$; i.e., the number of permutations with an even number of alternating runs is $n!/2$. He also proved the formula for the mean, and determined the number of permutations which have the maximum number of alternating runs (see exercise 5.1.4–22). It can be shown that

$$G_n(z) = \left(\frac{1+z}{2}\right)^{n-1} (1+w)^{n+1} g_n\left(\frac{1-w}{1+w}\right), \qquad w = \sqrt{\frac{1-z}{1+z}}, \qquad n \geq 2,$$

where $g_n(z)$ is the generating function (18) for ascending runs. [See David and Barton, *Combinatorial Chance* (London: Griffin, 1962), 157–162.]

17. $({n+1 \atop 2k-1})$; $({n \atop 2k-2})$ end with 0, $({n \atop 2k-1})$ end with 1.

18. Let the given sequence be $C_1\ C_2\ \ldots\ C_n$, an inversion table such as we have considered in exercise 5.1.1–7. If there are k runs in this sequence, there are k in the corresponding permutation; hence the answer is $\langle {n \atop k} \rangle$.

19. (a) $\langle {n \atop k+1} \rangle$, by the correspondence of Theorem 5.1.2B. (b) There are $(n-k)!$ ways to put $n-k$ further nonattacking rooks on the entire board; hence the answer is $1/(n-k)!$ times $\sum_{j \geq 0} a_{nj} {j \atop k}$, where $a_{nj} = \langle {n \atop j+1} \rangle$ is the solution for part (a) with $j = k$. This comes to $\{ {n \atop n-k} \}$, by exercise 2.

A direct proof of this result, due to E. A. Bender, associates each partition of $\{1, 2, \ldots, n\}$ into k nonempty disjoint subsets with an arrangement of $(n-k)$ rooks: Let the partition be

$$\{1, 2, \ldots, n\} = \{a_{11}, a_{12}, \ldots, a_{1n_1}\} \cup \cdots \cup \{a_{k1}, \ldots, a_{kn_k}\},$$

where $a_{ij} < a_{i(j+1)}$ for $1 \leq j < n_i$, $1 \leq i \leq k$. The corresponding arrangement puts rooks in column a_{ij} of row $a_{i(j+1)}$, for $1 \leq j < n_i$, $1 \leq i \leq k$. For example, the configuration illustrated in Fig. 4 corresponds to the partition $\{1, 3, 8\} \cup \{2\} \cup \{4, 6\} \cup \{5\} \cup \{7\}$.

20. The number of readings is the number of runs in the inverse permutation. The first run corresponds to the first reading, etc.

21. It has $n+1-k$ runs and requires $n+1-j$ readings.

22. [*J. Combinatorial Theory* **1** (1966), 350–374.] If $rs < n$, some reading will pick up $t > r$ elements, $a_{i_1} = j+1, \ldots, a_{i_t} = j+t$, where $i_1 < \cdots < i_t$. We cannot have $a_m > a_{m+1}$ for all m in the range $i_k \leq m < i_{k+1}$, so the permutation contains at least $t-1$ places with $a_m < a_{m+1}$; it therefore has at most $n-t+1$ runs.

But if $rs \geq n$, consider the permutation

$$(tr) \ldots (2r)\ (r) \ldots (3)\ ((t-1)r+2) \ldots (r+2)\ (2)\ ((t-1)r+1) \ldots (r+1)\ (1)$$

where $\lceil n/r \rceil = t \leq s$, and where elements $> n$ are to be omitted. It has $n+1-r$ runs and it requires $n+1-r$ readings. By rearranging the groups $\{kr+r, \ldots, kr+1\}$ independently among themselves, it is possible to adjust the number of readings to any desired lesser value $s \geq t$.

23. [*SIAM Review* **3** (1967), 121–122.] Assume that the infinite permutation consists of independent samples from the uniform distribution. Let $f_k(x)\,dx$ be the probability that the kth long run begins with x; and let $g(u, x)\,dx$ be the probability that a long run begins with x, when the preceding long run begins with u. Then $f_1(x) = 1$, $f_{k+1}(x) = \int_0^1 f_k(u)g(u, x)\,du$. We have $g(u, x) = \sum_{m\geq 1} g_m(u, x)$, where $g_m(u, x) = \Pr(u < X_1 < \cdots < X_m > x \text{ or } u > X_1 > \cdots > X_m < x) = \Pr(u < X_1 < \cdots < X_m) + \Pr(u > X_1 > \cdots > X_m) - \Pr(u < X_1 < \cdots < X_m < x) - \Pr(u > X_1 > \cdots > X_m > x) = (u^m + (1-u)^m + |u-x|^m)/m!$; hence $g(u, x) = e^u + e^{1-u} - 1 - e^{|u-x|}$. Consequently, $f_2(x) = 2e - 1 - e^x - e^{1-x}$. It can be shown that $f_k(x)$ approaches the limiting value $(2\cos(x-\frac{1}{2}) - \sin\frac{1}{2} - \cos\frac{1}{2})/(3\sin\frac{1}{2} - \cos\frac{1}{2})$. The average length of a run starting with x is $e^x + e^{1-x} - 1$ hence the length LL_k of the kth long run is $\int_0^1 f_k(x)(e^x + e^{1-x} - 1)\,dx$; $LL_1 = 2e - 3 \approx 2.4365$; $LL_2 = 3e^2 - 8e + 2 \approx 2.4209$. See Section 5.4.1 for similar results.

24. Arguing as before, the result is

$$1 + \sum_{0\leq k<n} 2^k(p^2+q^2)^k\big(p^2 + 2pq(2^{n-k-1} - 1 + q^2((2pq)^{n-k-1} - 1)/(2pq-1))\big);$$

carrying out the sum and simplifying yields

$$2^n(p^2+q^2)^n(p(p-q)/(p^2+q^2-pq) - \tfrac{1}{2}) + (2pq)^n pq^3/(p^2+q^2)(p^2+q^2-pq) + q^2/(p^2+q^2) + 2^{n-1}.$$

SECTION 5.1.4

1.

1	2	3	8
4	5	7	
6	9		

1	3	5	8
2	4	9	
6	7		

; $\begin{pmatrix} 1 & 3 & 4 & 5 & 7 & 8 & 9 \\ 5 & 9 & 2 & 4 & 8 & 1 & 7 \end{pmatrix}$.

2. When p_i is inserted into column t, let the element in column $t-1$ be p_j. Then (q_j, p_j) is in class $t-1$, $q_j < q_i$, and $p_j < p_i$; so, by induction, indices $i_1, \ldots, i_t$ exist with the property. Conversely, if $q_j < q_i$ and $p_j < p_i$ and if (q_j, p_j) is in class $(t-1)$, then column $t-1$ contains an element $< p_i$ when p_i is inserted, so (q_i, p_i) is in class t.

3. The columns are the "bumping sequences" (9) when p_i is inserted. Lines 1 and 2 reflect the operations on row 1, cf. (14). If we remove columns in which line 2 has ∞ entries, lines 0 and 2 constitute the "bumped" array, as in (15). The stated method for going from line k to line $k+1$ is just the class-determination algorithm of the text.

4. Let there be t classes in all; exactly k of these have an odd number of elements, since the elements of a class have the form

$$(p_{i_k}, p_{i_1}), \qquad (p_{i_{k-1}}, p_{i_2}), \qquad \ldots, \qquad (p_{i_1}, p_{i_k}).$$

(See (18) and (22).) The "bumped" two-line array has exactly $t-k$ fixed points, because of the way it is constructed; hence by induction the tableau minus its first

row has $t - k$ columns of odd length. So the t elements in the first row lead to k odd-length columns in the whole tableau.

5. (a) Use a case analysis, by induction on the size of the tableau, considering first the effect on row 1 and then the effect on the sequence of elements bumped from row 1. (b) Admissible interchanges can simulate the operations of Algorithm I, with the tableau represented as a canonical permutation before and after the algorithm. For example, we can transform

$$17\ 11\ 4\ 13\ 14\ 2\ 6\ 10\ 15\ 1\ 3\ 5\ 9\ 12\ 16\ 8$$

into

$$17\ 11\ 13\ 4\ 10\ 14\ 2\ 6\ 9\ 15\ 1\ 3\ 5\ 8\ 12\ 16$$

by a sequence of admissible interchanges (cf. (4) and (5)).

6. Admissible interchanges are symmetrical between left and right, and the canonical permutation for P obviously goes into P^T when the insertion order is reversed.

7. The number of columns, namely the length of row 1, is the number of classes (exercise 2). The number of rows is the number of columns of P^T, so exercise 6 (or Theorem D) completes the proof.

8. With more than n^2 elements, the corresponding P tableau must either have more than n rows or more than n columns. But there are $n \times n$ tableaux. [This result was originally proved in *Compositio Math.* **2** (1935), 463–470.]

9. Such permutations are in 1–1 correspondence with pairs of tableaux of shape $(n, n, \ldots, n)$; so by (34) the answer is

$$\left(\frac{n^2!\,\Delta(2n-1, 2n-2, \ldots, n)}{(2n-1)!(2n-2)!\ldots n!}\right)^2 = \left(\frac{n^2!}{(2n-1)(2n-2)^2\ldots n^n(n-1)^{n-1}\ldots 1^1}\right)^2$$

The existence of such a simple formula for this problem is truly amazing. We can also count the number of permutations of $\{1, 2, \ldots, mn\}$ with no increasing subsequences longer than m, no decreasing subsequences longer than n.

10. We prove inductively that, at step S3, $P_{(r-1)s}$ and $P_{r(s-1)}$ are both less than $P_{(r+1)s}$ and $P_{r(s+1)}$.

11. We also need to know, of course, the element which was originally P_{11}. Then it is possible to restore things using an algorithm remarkably similar to Algorithm S.

12. $\binom{n_1+1}{2} + \binom{n_2+2}{2} + \cdots + \binom{n_m+m}{2} - \binom{m+1}{3}$.

The minimum is the sum of the first n terms of the sequence 1, 2, 2, 3, 3, 3, 4, 4, 4, 4, ... of exercise 1.2.4–41; this sum is approximately $\sqrt{8/9}\, n^{3/2}$. (Since the majority of tableaux on n elements come reasonably near this lower bound, according to the tables cited in exercise 28, the average number of times is almost surely $O(n^{3/2})$, but this has not been proved.)

13. Assume that the elements permuted are $\{1, 2, \ldots, n\}$ so that $a_i = 1$; and assume that $a_j = 2$. *Case 1*, $j < i$. Then 1 bumps 2, so row 1 of the tableau corresponding to $a_1 \ldots a_{i-1}\ a_{i+1} \ldots a_n$ is row 1 of P^S; and the "bumped" permutation is the former bumped permutation except for its smallest element, 2, so we may use

induction on n. *Case 2*, $j > i$. Apply Case 1 to P^T, in view of exercise 6 and the fact $(P^T)^S = (P^S)^T$.

15. As in (37), the example permutation corresponds to the tableau

1	2	5	9	11
3	6	7		
4	8	10		

;

hence the number is $f(l, m, n) = (l+m+n)!(l-m+1)(l-n+2)(m-n+1)/(l+2)!(m+1)!(n)!$, provided, of course, that $l \geq m \geq n$.

16. By Theorem H, 80080.

17. Since g is antisymmetric in the x's, it is zero when $x_i = x_j$, so it is divisible by $x_i - x_j$ for all $i < j$. Hence $g(x_1, \ldots, x_n; y) = h(x_1, \ldots, x_n; y)\Delta(x_1, \ldots, x_n)$. Here h must be homogeneous in $x_1, \ldots, x_n, y$, of total degree 1, and symmetric in $x_1, \ldots, x_n$; so $h(x_1, \ldots, x_n; y) = a(x_1 + \cdots + x_n) + by$ for some a, b depending only on n. We can evaluate a by setting $y = 0$; we can evaluate b by taking the partial derivative with respect to y and then setting $y = 0$. We have

$$\frac{\partial}{\partial y}\Delta(x_1, \ldots, x_i + y, \ldots, x_n)|_{y=0} = \frac{\partial}{\partial x_i}\Delta(x_1, \ldots, x_n)$$

$$= \Delta(x_1, \ldots, x_n)\sum_{j\neq i}\frac{1}{x_i - x_j}.$$

Finally,

$$\sum_i\sum_{j\neq i}(x_i/(x_i - x_j)) = \sum_i\sum_{j<i}(x_i/(x_i - x_j) + x_j/(x_j - x_i)) = \binom{n}{2}.$$

18. It must be $\Delta(x_1, \ldots, x_n)\cdot(b_0 + b_1y + \cdots + b_my^m)$, where each b_k is a homogeneous symmetric polynomial of degree $m - k$ in the x's. We have

$$\frac{\partial^k}{k!\partial y^k}\Delta(x_1, \ldots, x_i + y, \ldots, x_n)|_{y=0} = \Delta(x_1, \ldots, x_n)\sum 1/\prod_l(x_i - x_{j_l})$$

summed over all $\binom{n-1}{k}$ choices of distinct indices $j_1, \ldots, j_k \neq i$. Now in $b_k = \sum x_i^m/\prod_l(x_i - x_j)$ we may combine those groups of $k+1$ terms having a given set of indices $\{i, j_1, \ldots, j_k\}$; for example, when $k = 2$, we group sets of three terms of the form $a^m/(a-b)(a-c) + b^m/(b-a)(b-c) + c^m/(c-a)(c-b)$. The sum of each such group is evaluated as in exercise 1.2.3–33; it is the coefficient of z^{m-k} in $1/(1 - x_iz)(1 - x_{j_1}z)\cdots(1 - x_{j_k}z)$. We find therefore that

$$b_k = \sum_j\binom{n-j}{k+1-j}\sum s(p_1, \ldots, p_j),$$

where $s(p_1, \ldots, p_j)$ is the monomial symmetric function consisting of all distinct terms having the form $x_{i_1}^{p_1}\ldots x_{i_j}^{p_j}$, for distinct indices $i_1, \ldots, i_j \in \{1, \ldots, n\}$; and the second sum is over all partitions of $m - k$ into exactly j parts, namely $p_1 \geq \cdots \geq p_j \geq 1, p_1 + \cdots + p_j = m - k$. (This result was obtained jointly with E. A. Bender.)

When $m = 2$ the answer is $(s(2) + (n-1)s(1)y + \binom{n}{3}y^2)\,\Delta(x_1, \ldots, x_n)$; for $m = 3$ we get $(s(3) + ((n-1)s(2) + s(1,1))y + \binom{n-1}{2}s(1)y^2 + \binom{n}{4}y^3)\,\Delta(x_1, \ldots, x_n)$; etc.

Another expression gives b_k as the coefficient of z^m in

$$\left(\binom{n}{k+1}z^k - \binom{n-1}{k+1}a_1 z^{k+1} + \binom{n-2}{k+1}a_2 z^{k+2} - \cdots\right) \Big/ (1 - a_1 z + a_2 z^2 - \cdots),$$

where $a_l = \sum_{1 \le i_1 < \cdots < i_l \le n} x_{i_1} \ldots x_{i_l}$ is an "elementary symmetric function." Multiplying by y^k and summing on k gives the answer as the coefficient of z^m in

$$\frac{1}{yz}\left(\frac{(1 + z(y - x_1)) \cdots (1 + z(y - x_n))}{(1 - zx_1) \cdots (1 - zx_n)} - 1\right)\Delta(x_1, \ldots, x_n).$$

19. Let the shape of the transposed tableau be $(n_1', n_2', \ldots, n_r')$; the answer is $\frac{1}{2}f(n_1, n_2, \ldots, n_m)$ times $(\sum(n_i^2) - \sum(n_j'^2))/n(n-1) + 1$, where $n = \sum n_i = \sum n_j'$. (This formula was obtained after a rather elaborate computation, using the sum

$$\sum_{i<j} x_i x_j \Delta(x_1, \ldots, x_i + y, \ldots, x_j + y, \ldots, x_n)$$
$$= \left(s(1,1) + s(1)\binom{n-1}{2}y + \left\{\begin{matrix} n \\ n-2 \end{matrix}\right\}y^2\right)\Delta(x_1, \ldots, x_n)$$

at one point; cf. exercise 18, and note that

$$\left\{\begin{matrix} n \\ n-2 \end{matrix}\right\} = \binom{n}{3} + 3\binom{n}{4}.$$

The answer can be expressed in a less symmetrical form using the relation $\sum i n_i = n + \frac{1}{2}\sum n_j'^2$.)

20. The fallacious argument in the discussion following Theorem H is actually valid for this case (the corresponding probabilities *are* independent).

21. [*Michigan Math. J.* 1 (1952), 81–88.] Let $g(n_1, \ldots, n_m) = (n_1 + \cdots + n_m)!\,\Delta(n_1, \ldots, n_m)/n_1! \ldots n_m!\sigma(n_1, \ldots, n_m)$, where $\sigma(x_1, \ldots, x_m) = \prod_{1 \le i < j \le m}(x_i + x_j)$. To prove that $g(n_1, \ldots, n_m)$ is the number of ways to fill the shifting tableau, we must prove that $g(n_1, \ldots, n_m) = g(n_1 - 1, \ldots, n_m) + \cdots + g(n_1, \ldots, n_m - 1)$. The identity corresponding to exercise 17 is $x_1\Delta(x_1 + y, \ldots, x_n)/\sigma(x_1 + y, \ldots, x_n) + \cdots + x_n\Delta(x_1, \ldots, x_n + y)/\sigma(x_1, \ldots, x_n + y) = (x_1 + \cdots + x_n)\Delta(x_1, \ldots, x_n)/\sigma(x_1, \ldots, x_n)$, independent of y; for if we calculate the derivative as in exercise 17, we find that $2x_i x_j/(x_j^2 - x_i^2) + 2x_j x_i/(x_i^2 - x_j^2) = 0$.

22. [*Journ. de Math.* (3) 7 (1881), 167–184.] (This is a special case of exercise 5.1.3–8, with all runs, except perhaps the last, of length 2.) When $n > 0$, element n must appear in one of the rightmost positions of a row; once it has been placed in the rightmost box on row k, we have $\binom{n-1}{2k-1}A_{2k-1}A_{n-2k}$ ways to complete the job. Let

$$h(z) = \sum_{n \ge 1} A_{2n-1}z^{2n-1}/(2n-1)! = \tfrac{1}{2}(g(z) - g(-z));$$

then

$$h(z)g(z) = \sum_{k,n \ge 1}\binom{n}{2k-1}A_{2k-1}A_{n-2k+1}z^n/n! = \left(\sum_{n \ge 1} A_{n+1}z^n/n!\right) - 1 = g'(z) - 1.$$

Replace z by $-z$ and add, obtaining $h(z)^2 = h'(z) - 1$; hence $h(z) = \tan z$. Setting $k(z) = g(z) - h(z)$, we have $h(z)k(z) = k'(z)$; hence $k(z) = \sec(z)$ and $g(z) = \sec z + \tan z = \tan(\frac{1}{2}z + \frac{1}{4}\pi)$. The coefficients A_{2n} are therefore the Euler numbers E_{2n}; the coefficients A_{2n+1} are the tangent numbers T_{2n+1}. Tables of these numbers appear in *Math. Comp.* **21** (1967), 663–688.

23. Assume that $m = N$, by adding 0's to the shape if necessary; if $m > N$ and $n_m > 0$, the number of ways is clearly zero. When $m = N$ the answer is

$$\det \begin{pmatrix} \binom{n_1+m-1}{m-1} & \binom{n_2+m-2}{m-1} & \cdots & \binom{n_m}{m-1} \\ \vdots & & & \vdots \\ \binom{n_1+m-1}{0} & \binom{n_2+m-1}{0} & \cdots & \binom{n_m}{0} \end{pmatrix}.$$

Proof: We may assume that $n_m = 0$, for if $n_m > 0$, the first n_m columns of the array must be filled with i in row i, and we may consider the remaining shape $(n_1 - n_m, \ldots, n_m - n_m)$. By induction on m, the number of ways is

$$\sum_{\substack{n_2 \le k_1 \le n_1 \\ \vdots \\ n_m \le k_{m-1} \le n_{m-1}}} \det \begin{pmatrix} \binom{k_1+m-2}{m-2} & \binom{k_2+m-2}{m-2} & \cdots & \binom{k_m}{m-2} \\ \vdots & & & \vdots \\ \binom{k_1+m-2}{0} & \binom{k_2+m-2}{0} & \cdots & \binom{k_m}{0} \end{pmatrix},$$

where $n_j - k_j$ represents the number of m's in row j. The sum on each k_j may be carried out independently, giving

$$\det \begin{pmatrix} \binom{n_1+m-1}{m-1} - \binom{n_2+m-2}{m-1} & \binom{n_2+m-2}{m-1} - \binom{n_3+m-3}{m-1} & \cdots & \binom{n_{m-1}+1}{m-1} - \binom{n_m}{m-1} \\ \vdots & & & \vdots \\ \binom{n_1+m-1}{1} - \binom{n_2+m-2}{1} & \binom{n_2+m-2}{1} - \binom{n_3+m-3}{1} & \cdots & \binom{n_{m-1}+1}{1} - \binom{n_m}{1} \end{pmatrix}.$$

which is the desired answer since $n_m = 0$. The answer can be converted into a Vandermonde determinant by row operations, giving the formula $\Delta(n_1 + m - 1, n_2 + m - 2, \ldots, n_m)/(m-1)!(m-2)!\ldots 0!$. [The answer to this exercise, in connection with an equivalent problem in group theory, appears in D. E. Littlewood's *Theory of Group Characters* (Oxford, 1940), 189.]

25. In general, if u_{nk} is the number of permutations on $\{1, 2, \ldots, n\}$, having no cycles of length $> k$, $\sum u_{nk} z^n/n! = \exp(z + z^2/2 + \cdots + z^k/k)$; this is proved by multiplying $\exp(z) \times \cdots \times \exp(z^k/k)$, obtaining

$$\sum_n z^n/n! \left(\sum_{j_1+2j_2+\cdots+kj_k=n} 1/1^{j_1} j_1! 2^{j_2} j_2! \ldots \right);$$

cf. exercise 1.3.3–21. Similarly, $\exp(\sum_{s \in S} z^s/s)$ is the corresponding generating function for permutations whose cycle lengths are all members of a given set S.

26. The integral from 0 to ∞ is $n^{(t+1)/4}\Gamma((t+1)/2)/2^{(t+3)/2}$, by the gamma function integral (exercise 1.2.5–20, $t = 2x^2/\sqrt{n}$). So, from $-\infty$ to ∞, we get 0 when t is odd, otherwise $n^{(t+1)/4}\sqrt{\pi}\, t!/2^{(3t+1)/2}(t/2)!$.

27. (a) If $r_i < r_{i+1}$ and $c_i < c_{i+1}$, $i < Q_{r_i c_{i+1}} < i+1$ is impossible. If $r_i \geq r_{i+1}$ and $c_i < c_{i+1}$, clearly $r_i \neq r_{i+1}$ so a similar contradiction is obtained. (b) Prove, by induction on the number of rows in the tableau for $a_1 \ldots a_i$, that $a_i < a_{i+1}$ implies $c_i < c_{i+1}$, and $a_i > a_{i+1}$ implies $c_i \geq c_{i+1}$. (Consider row 1 and the "bumped" sequences.) (c) This follows from Theorem D(c).

30. [*Discrete Math.* **2** (1972), 73–94.]

31. $x_n = a_{\lfloor n/2 \rfloor}$ where $a_0 = 1$, $a_1 = 2$, $a_n = 2a_{n-1} + (2n-2)a_{n-2}$; $\sum a_n z^n/n! = \exp(2z + z^2) = (\sum t_n z^n/n!)^2$; $x_n \sim \exp(\frac{1}{4}n \ln n - \frac{1}{4}n + \sqrt{n} - \frac{1}{2} - \frac{1}{2}\ln 2)$.

SECTION 5.2

1. Yes, i and j may run through the set of values $1 \le j < i \le N$ in any order. This shows that counting can take place simultaneously as records are being read in.

2. The sorting is "stable" as defined at the beginning of the chapter; for the algorithm is essentially sorting by lexicographic order on the *distinct* key-pairs $(K_1, 1)$, $(K_2, 2)$, . . . , (K_N, N). (If we think of each key as extended on the right by its location in the file, no equal keys are present, and the sorting is stable.)

3. It would sort, but not in a "stable" manner; if $K_j = K_i$ and $j < i$, R_j will come *after* R_i in the final ordering. This change would also make Program C run more slowly.

4.
```
ENT1    N               1
LD2     COUNT,1         N
LDA     INPUT,1         N
STA     OUTPUT+1,2      N
DEC1    1               N
J1P     *-4             N
```
∎

5. The running time is changed by $N - 1 - A + B$ units, and this is almost always an improvement.

6. $u = 0, v = 9$.

After D1,	COUNT =	0	0	0	0	0	0	0	0	0	0
After D2,	COUNT =	2	2	1	0	1	3	3	2	1	1
After D4,	COUNT =	2	4	5	5	6	9	12	14	15	16
During D5,	COUNT =	2	3	5	5	5	8	9	12	15	16 $j = 8$
	OUTPUT =	—	—	—	1G	—	4A	—	—	5L	6A 6T 6I 7O 7N — —
After D5,	OUTPUT =	0C	0O	1N	1G	2R	4A	5T	5U	5L	6A 6T 6I 7O 7N 8S 9.

7. Yes (note that COUNT[K_j] is decreased in step D6, and j decreases).

8. It would sort, but not in a "stable" manner (cf. exercise 7).

9. Let $M = v - u$; LOC(R_j) ≡ INPUT + j; LOC(COUNT[j]) ≡ COUNT + j; LOC(S_j) ≡ OUTPUT + j; rI1 ≡ i; rI2 ≡ j; rI3 ≡ $i - v$, K_j. Assume that $|u|$, $|v|$ fit in two bytes.

```
 M      EQU     V-U
KEY     EQU     0:2             (Satellite information in bytes 3:5.)
1H      ENN3    M               1           D1. Clear COUNTs.
        STZ     COUNT+V,3       M+1         COUNT[v - k] ← 0.
        INC3    1               M+1
        J3NP    *-2             M+1         u ≤ i ≤ v.
2H      ENT2    N               1           D2. Loop on j.
```

```
3H  LD3   INPUT,2(KEY)   N    D3. Increase COUNT[K_j].
    LDA   COUNT,3        N
    INCA  1              N
    STA   COUNT,3        N
    DEC2  1              N
    J2P   3B             N    N ≥ j > 0.
    ENN3  M-1            1    D4. Accumulate.
    LDA   COUNT+U        1    rA ← COUNT[i - 1].
4H  ADD   COUNT+V,3      M    COUNT[i - 1] + COUNT[i]
    STA   COUNT+V,3      M      → COUNT[i].
    INC3  1              M
    J3NP  4B             M    u ≤ i ≤ v.
5H  ENT2  N              1    D5. Loop on j.
6H  LD3   INPUT,2(KEY)   N    D6. Output R_j.
    LD1   COUNT,3        N    i ← COUNT[K_j].
    LDA   INPUT,2        N    rA ← R_j.
    STA   OUTPUT,1       N    S_i ← rA.
    DEC1  1              N
    ST1   COUNT,3        N    COUNT[K_j] ← i - 1.
    DEC2  1              N
    J2P   6B             N    N ≥ j > 0.  ∎
```

The running time is $(10M + 22N + 10)u$.

10. In order to avoid using N extra "tag" bits [see Section 1.3.3 and *Cybernetics* **1** (1965), 95], yet keep the running time essentially proportional to N, we may use the following algorithm based on the cycle structure of the permutation:

P1. [Loop on i.] Do step P2 for $1 \le i \le N$; then terminate the algorithm.
P2. [Is $p(i) = i$?] Do steps P3 through P5, if $p(i) \neq i$.
P3. [Begin cycle.] Set $t \leftarrow R_i$, $j \leftarrow i$.
P4. [Fix R_j.] Set $k \leftarrow p(j)$, $R_j \leftarrow R_k$, $p(j) \leftarrow j$, $j \leftarrow k$. If $p(j) \neq i$, repeat this step.
P5. [End cycle.] Set $R_j \leftarrow t$, $p(j) \leftarrow j$. ∎

This algorithm changes $p(i)$, since the sorting application lets us assume that $p(i)$ is stored in memory. On the other hand, there are applications such as matrix transposition where $p(i)$ is a function of i which is to be computed (not tabulated) in order to save memory space. In such a case we can use the following method, performing steps B1 through B3 for $1 \le i \le N$:

B1. Set $k \leftarrow p(i)$.
B2. If $k > i$, set $k \leftarrow p(k)$ and repeat this step.
B3. If $k < i$, do nothing; but if $k = i$ (this means that i is smallest in its cycle), we permute the cycle containing i as follows: Set $t \leftarrow R_i$; then while $p(k) \neq i$ repeatedly set $R_k \leftarrow R_{p(k)}$ and $k \leftarrow p(k)$; finally set $R_k \leftarrow t$. ∎

This algorithm is similar to the procedure of J. Boothroyd [*Comp. J.* **10** (1967), 310], but requires less data movement; some refinements have been suggested by I. D. G. MacLeod [*Australian Comp. J.* **2** (1970), 16–19]. For random permutations the analysis in exercise 1.3.3–14 shows that step B2 is performed $(N + 1)H_N - N$ steps on the average. Cf. D. E. Knuth [*Proc. IFIP Congress* 1971, **1**, 19–27]. Similar algorithms

can be designed to replace $(R_{p(1)}, \ldots, R_{p(N)})$ by $(R_1, \ldots, R_N)$, e.g. if the rearrangement in exercise 4 were to be done with OUTPUT = INPUT.

11. Let rI1 $\equiv i$; rI2 $\equiv j$; rI3 $\equiv k$; rX $\equiv t$.

```
1H    ENT1   N            1          P1. Loop on i.
2H    CMP1   P,1          N          P2. Is p(i) = i?
      JE     8F           N          Jump if p(i) = i.
3H    LDX    INPUT,1      A — B      P3. Begin cycle.  t ← R_i.
      ENT2   0,1          A — B      j ← i.
4H    LD3    P,2          N — A      P4. Fix R_j.  k ← p(j).
      LDA    INPUT,3      N — A
      STA    INPUT,2      N — A      R_j ← R_k.
      ST2    P,2          N — A      p(j) ← j.
      ENT2   0,3          N — A      j ← k.
      CMP1   P,2          N — A
      JNE    4B           N — A      Repeat if p(j) ≠ i.
5H    STX    INPUT,2      A — B      P5. End cycle.  R_j ← t.
      ST2    P,2          A — B      p(j) ← j.
8H    DEC1   1            N
      J1P    2B           N          N ≥ i ≥ 1.  ∎
```

The running time is $(17N - 5A - 7B + 1)u$, where A is the number of cycles in the permutation $p(1) \ldots p(N)$, and B is the number of fixed points (1-cycles). By Eqs. 1.3.3–21 and 28, A = (min 1, ave H_N, max N, dev $\sqrt{H_N - H_N^{(2)}}$); B = (min 0, ave 1, max N, dev 1), for $N \geq 2$.

12. The following "direct" method is due to M. D. MacLaren. (Assume for convenience that $0 \leq$ LINK(P) $\leq N$, for $1 \leq$ P $\leq N$, where $\Lambda \equiv 0$.)

M1. [Initialize.] Set P $\leftarrow$ HEAD, $k \leftarrow 1$.

M2. [Done?] If P $= \Lambda$ (or equivalently if $k = N + 1$), the algorithm terminates.

M3. [Ensure P $\geq k$.] If P $< k$, set P $\leftarrow$ LINK(P) and repeat this step.

M4. [Exchange.] Interchange R_k and R[P]. (Assume that LINK(k) and LINK(P) are also interchanged in this process.) Then set Q $\leftarrow$ LINK(k), LINK(k) $\leftarrow$ P, P $\leftarrow$ Q, $k \leftarrow k + 1$, and return to step M2. ∎

A proof that MacLaren's method is valid can be based on an inductive verification of the following property which holds at the beginning of step M2: The entries which are $\geq k$ in the sequence P, LINK(P), LINK(LINK(P)), ..., Λ are $a_1, a_2, \ldots, a_{N+1-k}$, where $R_1 \leq \cdots \leq R_{k-1} \leq R_{a_1} \leq \cdots \leq R_{a_{N+1-k}}$ is the desired final order of the records. Furthermore LINK(j) $\geq j$ for $1 \leq j < k$, so that LINK(j) $= \Lambda$ implies $j \geq k$.

It is quite interesting to analyze the above algorithm; one of its remarkable properties is that it can be run backwards, reconstructing the original set of links from the final values of LINK(1) ... LINK(N). Each of the $N!$ possible output configurations with $j \leq$ LINK(j) $\leq N$ corresponds to exactly one of the $N!$ possible input configurations. If A is the number of times P $\leftarrow$ LINK(P) in step M3, then $N - A$ is the number of j such that LINK(j) $= j$ at the conclusion of the algorithm; this occurs if and only if j was largest in its cycle; hence $N - A$ is the number of cycles in the permutation, and A = (min 0, ave $N - H_N$, max $N - 1$).

13. D5′. Set $r \leftarrow N$.

D6′. If $r = 0$, stop. Otherwise if $\texttt{COUNT}[K_r] < r$ set $r \leftarrow r - 1$ and repeat this step; if $\texttt{COUNT}[K_r] = r$, decrease both $\texttt{COUNT}[K_r]$ and r by 1 and repeat this step. Otherwise set $R \leftarrow R_r$, $j \leftarrow \texttt{COUNT}[K_r]$, $\texttt{COUNT}[K_r] \leftarrow j - 1$.

D7′. Set $S \leftarrow R_j$, $k \leftarrow \texttt{COUNT}[K_j]$, $\texttt{COUNT}[K_j] \leftarrow k - 1$, $R_j \leftarrow R$, $R \leftarrow S$, $j \leftarrow k$. Then if $j \neq r$ repeat this step; if $j = r$ set $R_j \leftarrow R$, $r \leftarrow r - 1$, and go back to D6′. ∎

To prove that this is valid, observe that at the beginning of step D6′ all records R_j such that $j > r$ which are not in their final resting place must move to the left; when $r = 0$ there can't be any such records since *somebody* must move right. The algorithm is elegant but not "stable" for equal keys; it is intimately related to Foata's construction in Theorem 5.1.2B.

1. Yes, equal elements are never moved across each other.

2. Yes, but the running time would be slower when equal elements are present and the sorting would be just the opposite of stable.

3. The following eight-liner is conjectured to be the shortest MIX sorting routine, although it is not recommended for speed. We assume that the numbers appear in locations $1, \ldots, N$ (i.e. `INPUT EQU 0`; otherwise another line of code is necessary).

```
2H      LDA   0,1      B
        CMPA  1,1      B
        JLE   1F       B
        MOVE  1,1      A
        STA   0,1      A
START   ENT1  N        A + 1
1H      DEC1  1        B + 1
        J1P   2B       B + 1
```

Note: To estimate the running time of this program, note that A is the number of inversions. B is a reasonably simple function of the inversion table, and (assuming distinct inputs in random order) it has the generating function

$$z^{N-1}(1+z)(1+z^2+z^{2+1})(1+z^3+z^{3+2}+z^{3+2+1})\cdots\Big(1+z^{N-1}+z^{2N-3}+\cdots+z^{\binom{N}{2}}\Big)\Big/N!.$$

The mean value of B is $N-1+\sum_{1\le k\le N}(k-1)(2k-1)/6 = (N-1)(4N^2+N+36)/36$; hence the average running time of this program is roughly $\frac{7}{9}N^3u$.

4. Consider the inversion table $B_1 \ldots B_N$ of the given input permutation, in the sense of exercise 5.1.1–7. A is one less than the number of B_j's which are equal to $j-1$, and B is the sum of the B_j's. Hence both $B-A$ and B are maximized when the input permutation is $N \ldots 2\ 1$; they both are minimized when the input is $1\ 2 \ldots N$. Hence the minimum achievable time occurs for $A=0$, $B=0$, namely $(10N-9)u$; the maximum occurs for $A=N-1$, $B=\binom{N}{2}$, namely $4.5N^2+2.5N-6$.

5. The generating function is z^{10N-9} times the generating function for $9B-3A$. By considering the inversion table as in the previous exercise, remembering that individual entries of the inversion table are independent of each other, the desired

generating function is $z^{10N-9}\prod_{1<j\le N}((1+z^9+\cdots+z^{9j-18}+z^{9j-12})/j)$. The variance comes to $2.25N^3+3.375N^2-32.625N+36H_N-9H_N^{(2)}$.

6. Treat the input area as a circular list, with position N adjacent to position 1. Take new elements to be inserted from either the left or the right of the current segment of unsorted elements, according as the previously inserted element fell to the right or left of the center of the sorted elements, respectively. Afterwards it will usually be necessary to "rotate" the area, moving each record k places around the circle for some fixed k; this can be done in gcd (N, k) passes (cyclically permute R_i, R_{i+k}, ..., R_{N+i-k} for $1 \le i \le$ gcd (N, k); cf. W. Fletcher and R. Silver, *CACM* **9** (1966), 326).

7. The average value of $|a_j - j|$ is

$$\frac{1}{n}(|1-j|+|2-j|+\cdots+|n-j|) = \frac{1}{n}\left(\binom{j}{2}+\binom{n-j+1}{2}\right);$$

summing on j gives

$$\frac{1}{n}\left(\binom{n+1}{3}+\binom{n+1}{3}\right) = \frac{1}{3}(n^2-1).$$

8. No; for example, consider the keys 2 1 1 1 1 1 1 1 1 1 1.

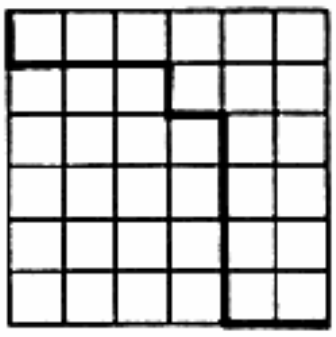
9. For Table 3, $A = 3+0+2+1 = 6$, $B = 3+1+4+21 = 29$; in Table 4, $A = 4+2+2+0 = 8$, $B = 4+3+8+10 = 25$; hence the running time of Program D comes to $786u$ and $734u$, respectively. Although the number of moves has been cut from 41 to 25, the running time is not competitive with Program S since the bookkeeping time for four passes is wasted when $N = 16$. When sorting 16 items we will be better off using only two passes; a two-pass Program D begins to beat Program S at about $N = 13$, although they are fairly equal for awhile (and for such small N the length of the program is perhaps significant).

10. Insert "`INC1 INPUT; ST1 3F(0:2)`" between lines 07 and 08, and change lines 10–17 to:

```
3H   CMPA   INPUT+N-H,1        NT - S
     JGE    8F                 NT - S
4H   ENT2   N-H,1              NT - S - C
5H   LDX    INPUT,2            B
6H   STX    INPUT+H,2          B
     DEC2   0,4                B
     J2NP   7F                 B
     CMPA   INPUT,2            B - A
     JL     5B                 B - A
7H   STA    INPUT+H,2          NT - S - C
```

For a net increase of four instructions, this saves $3(C-T)$ units of time, where C is the number of times $K_j \ge K_{j-h}$. In Tables 3 and 4 the time savings is approximately 87 and 88, respectively; empirically the value of $C/(NT-S)$ seems to be about 0.4 when $h_{s+1}/h_s \approx 2$ and about 0.3 when $h_{s+1}/h_s \approx 3$, so the improvement is worth while. (On the other hand, the analogous change to Program S is not desirable, since the savings in that case is only proportional to $\log N$ unless the input is known to be pretty well ordered.)

11.



12. When the path includes the line segment from (i, j) to $(i+1, j)$, we have $a_{2i+1} = i+j+1$ in the corresponding permutation $a_1 \ldots a_n$. If $i \le j$, there are $j-i$ elements to the right of a_{2i+1} which are less than it, but all elements to its left are less; a similar argument for the case $i \ge j$ shows that a_{2i+1} contributes to exactly $|i-j|$ inversions. This completes the proof, since each inversion involves one element with an odd subscript.

13. Put the weight $|i-j|$ on the segment from $(i, j-1)$ to (i, j). A path using this segment has $a_{2j} = i+j$, so we may argue as in exercise 12.

14. (a) Interchange i, j in the sum for A_{2n} and add these two sums. (b) Taking half of this result, we see that

$$A_{2n} = \sum_{0 \le i \le j} (j-i) \binom{i+j}{i} \binom{2n-i-j}{n-j};$$

hence $\sum A_{2n} z^n = z/(1-4z)^2$ by two applications of the stated identity.

The above proof was suggested to the author by Leonard Carlitz. Another proof can be based on interplay between horizontal and vertical weights (cf. exercise 13); but no simple combinatorial derivation of the formula $A_n = \lfloor n/2 \rfloor 2^{n-2}$ is apparent.

15. For $n > 0$,

$$\bar{g}_n(z) = z^{n-1} g_{n-1}(z); \; \bar{h}_n(z) = \bar{g}_n(z) + z^{-n} \bar{g}_n(z);$$
$$g_n(z) = \sum_{1 \le k \le n} \bar{g}_k(z) g_{n-k}(z); \; h_n(z) = \sum_{1 \le k \le n} \bar{h}_k(z) h_{n-k}(z).$$

Letting $G(w, z) = \sum_n g_n(z) w^n$, we find that $wzG(w, z)G(wz, z) = G(w, z) - 1$. From this representation we can deduce that, if $t = \sqrt{1-4w} = 1 - 2w - 2w^2 - 4w^3 - \cdots$, we have $G(w, 1) = (1-t)/(2w)$; $G_{,}(w, 1) = 1/(wt) - (1-t)/(2w^2)$; $G'(w, 1) = 1/(2t^2) - 1/(2t)$; $G_{,,}(w, 1) = 2/(wt^3) - 2/(w^2 t) + (1-t)/w^3$; $G'_{,}(w, 1) = 2/t^4 - 1/t^3$; and $G''(w, 1) = 1/t^3 - (1-2w)/t^4 + 10w^2/t^5$. Here lower primes denote differentiation with respect to the first parameter, and upper primes denote differentiation with respect to the second parameter. Similarly from the formula

$$w(zG(wz, z) + G(w, z))H(w, z) = H(w, z) - 1$$

we deduce that

$$H'(w, 1) = w/t^4, \qquad H''(w, 1) = -w/t^3 - w/t^4 + 2w/t^5 + (2w^2 + 20w^3)/t^7.$$

The formula manipulation summarized here was done by hand, but it should have been done by computer. In principle all moments of the distribution are obtainable in this way.

16. For $h = 2$ the maximum clearly occurs for the path that goes through the upper right corner of the lattice diagram, namely

$$\binom{\lfloor n/2 \rfloor + 1}{2}.$$

For general h the corresponding number is

$$\bar{f}(n, h) = \binom{h}{2}\binom{q+1}{2} + \binom{r}{2}(q+1),$$

where q and r are defined in Theorem H; for the permutation with $a_{i+jh} = (n+1 - i(q+1) + j$, if $1 \le i \le r$; $n+1-r-iq+j$, if $r < i \le h)$ maximizes the number of inversions between each of the $\binom{h}{2}$ pairs of sorted subsequences. The maximum number of moves is obtained if we replace f by $\bar{f}$ in (6).

17. The only two-ordered permutation of $\{1, 2, \ldots, 2n\}$ which has as many as $\binom{n+1}{2}$ inversions is $(n+1)\ 1\ (n+2)\ 2\ \ldots\ (2n)\ n$. Using this idea recursively, we obtain the permutation defined by adding unity to each element of the sequence $(2^t - 1)^R \ldots 1^R 0^R$, where R denotes the operation of writing an integer as a t-bit binary number, then reversing the left-to-right order of the digits!

18. Take out a common factor and let $h_{t+1} = 4N/\pi$; we want to minimize $\sum_{t \ge s \ge 1} h_{s+1}^{1/2}/h_s$, when $h_1 = 1$. Differentiation yields $h_s^3 = 4h_{s+1}^2 h_{s-1}$, which has the solution $h_s = 2^{s-2+s/(2^s-1)} h_{s+1}^{(2^{s-1}-1)/(2^s-1)}$. The minimum value of the originally-given estimate comes to $(1 - 2^{-t})\pi^{(2^{t-1}-1)/(2^t-1)} N^{1+(2^{t-1}+1)/(2^t-1)}/2^{1+(t-1)/(2^t-1)}$ which rapidly approaches $N\sqrt{\pi N}/2$ as $t \to \infty$.

Typical examples of "optimum" h's when $N = 1000$ (see also Table 6):

$$h_3 = 57.64, \quad h_2 = 6.13, \quad h_1 = 1;$$

$$h_5 = 284.5, \quad h_4 = 67.23, \quad h_3 = 16.34, \quad h_2 = 4.028, \quad h_1 = 1;$$

$$h_{10} = 9165, \quad h_9 = 12294, \quad h_8 = 7120, \quad h_7 = 2709, \quad h_6 = 835.5, \quad \ldots, \quad h_2 = 3.97, \quad h_1 = 1.$$

19. Let $g(n, h) = H_r - 1 + \sum_{r < j \le h} q/(qj + r)$, where q and r are defined in Theorem H; then replace f by g in (6).

20. (This is much harder to write down than to understand.) Assume that a k-ordered file $R_1, \ldots, R_N$ has been h-sorted, and let $1 \le i \le N - k$; we want to show that $K_i \le K_{i+k}$. Find u, v such that $1 \le u, v \le h$; $i \equiv u$ and $i + k \equiv v$ (modulo h); and apply Lemma L with $m = (u - v + k)/h$, $r = \lfloor (N - k + h - u)/h \rfloor$, $n + r = \lfloor (N + h - u)/h \rfloor$, $x_j = K_{v+(j-1)h}$, $y_j = K_{u+(j-1)h}$.

21. If $ah + bk = hk - h - k$, then $a \bmod k = k - 1$ and $b \bmod h = h - 1$; hence $ah + bk \ge (k-1)h + (h-1)k > hk - h - k$, a contradiction. Conversely, if $n \ge (h-1)(k-1)$, choose a so that $0 \le a < k$ and $ah \equiv n$ (modulo k). Then $(n - ah)/k$ is a nonnegative integer, b; hence n is representable.

A slight strengthening of this argument shows that exactly $\frac{1}{2}(h-1)(k-1)$ positive integers are unrepresentable in the stated form. (See R. Z. Norman, *AMM* **67** (1960), 594.)

22. To avoid cumbersome notation, consider $h = 4$, which is representative of the general case. Let n_k be the smallest number representable in the form $15a_0 + 31a_1 + \cdots$ which is congruent to k (modulo 15); then we find easily that

$k =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$n_k =$	15	31	62	63	94	125	126	127	158	189	190	221	252	253	254.

Hence $239 = 2^4(2^4 - 1) - 1$ is the largest unrepresentable number, and the total number of unrepresentables is

$$\begin{aligned} x_4 &= (n_1 - 1 + n_2 - 2 + \cdots + n_{14} - 14)/15 \\ &= (2+4+4+6+8+8) + 8 + (10+12+12+14+16+16) + 16 \\ &= 2x_3 + 8 \cdot 9; \end{aligned}$$

in general, $x_h = 2x_{h-1} + 2^{h-1}(2^{h-1} + 1)$.

For the other problem the answers are $2^{2h} + 2^h + 2$ and $2^{h-1}(2^h + h - 1) + 2$, respectively.

23. Each of the N numbers has at most $\lceil (h_{s+2} - 1)(h_{s+1} - 1)/h_s \rceil$ inversions in its subfile.

24. (Solution obtained jointly with V. Pratt.) Construct the "h-recidivous permutation" of $\{1, 2, \ldots, N\}$ as follows. Start with $a_1 \ldots a_N$ blank; then for $j = 2, 3, 4, \ldots$ do Step j: Fill in all blank positions a_i from left to right, using the smallest number that has not yet appeared in the permutation, whenever $(2^h - 1)j - i$ is a positive integer representable as in exercise 22. Continue until all positions are filled. Thus the 2-recidivous permutation for $N = 20$ is

6 2 1 9 4 3 12 7 5 15 10 8 17 13 11 19 16 14 20 18.

The h-recidivous permutation is $(2^k - 1)$-ordered for all $k \geq h$. When $2^h < j \leq N/(2^h - 1)$, exactly $2^h - 1$ positions are filled during step j; the $(k+1)$st of these adds at least $2^{h-1} - 2k$ to the number of moves required to $(2^{h-1} - 1)$-sort the permutation. Hence the number of moves to sort the h-recidivous permutation with increments $h_s = 2^s - 1$ when $N = 2^{h+1}(2^h - 1)$ is $> 2^{3h-4} > \frac{1}{128}N^{3/2}$. Pratt has generalized this construction to a large family of similar sequences, including (8), in his Ph.D. thesis (Stanford University, 1972).

25. F_{n+1} [this result is due to H. B. Mann, *Econometrica* **13** (1945), 256]; for the permutation must begin either 1 . . . or 2 1 . . . . There are at most $\lfloor n/2 \rfloor$ inversions; and the total number of inversions is

$$\frac{n-1}{5} F_n + \frac{2n}{5} F_{n-1}.$$

(See exercise 1.2.8–12.) Note that the F_{n+1} permutations can conveniently be represented by "Morse code" sequences of dots and dashes, where a dash corresponds to an inversion; thus when $n = 4$ the permutations

1 2 3 4, 1 2 4 3, 1 3 2 4, 2 1 3 4, 2 1 4 3

correspond respectively to the Morse code sequences

••••, ••—, •—•, —••, ——

of length 4. Hence we have found the total number of dashes among all Morse code sequences of length n.

Since 3-sorting followed by 2-sorting doesn't leave a "random" 3- and 2-ordered permutation, the total number of inversions does not tell us anything precise about the average number obtained in Algorithm D.

26. Yes; a shortest example is 4 1 3 7 2 6 8 5, which has nine inversions. In general, the construction $a_{3k+s} = 3k + 4s$ for $-1 \le s \le 1$ yields files which are 3-, 5-, and 7-ordered, having approximately $\frac{4}{3}N$ inversions. When $N \bmod 3 = 2$ this construction is best possible.

28. 255 63 15 7 3 1. But better sequences are possible, see exercise 29.

29. Experiments by C. Tribolet in 1971 resulted in the choices 373 137 53 19 7 3 1 ($B_{\text{ave}} \approx 7100$) and 317 101 31 11 3 1 ($B_{\text{ave}} \approx 8300$). [The first of these resulted in a sorting time of $126457u$, compared to $131002u$ when the same data were sorted using increments (8).] In general, Tribolet suggests letting h_s be the nearest prime number to $N^{(s-1)/t}$. Experiments by Shelby Siegel in 1972 indicate that the best number of increments in such a method, for $N \le 10000$, is $t \approx \frac{4}{3} \ln (N/5.75)$.

Another good sequence, found by Robert L. Tomlinson, Jr., is 199 79 31 11 5 1 ($B_{\text{ave}} \approx 7600$).

The best three-increment sequence, according to extensive tests by Carole M. McNamee, appears to be 45 7 1 ($B_{\text{ave}} \approx 17600$). For four increments, 91 23 7 1 was the winner in her tests ($B_{\text{ave}} \approx 11500$), but a rather broad range of increments gave roughly the same performance.

30. The number of integer points in the triangular region

$$\{x \ln 2 + y \ln 3 < \ln N,\ x \ge 0,\ y \ge 0\} \qquad \text{is} \qquad \tfrac{1}{2}(\log_2 N)(\log_3 N) + O(\log N).$$

While we are h-sorting, the file is already $2h$-ordered and $3h$-ordered, by Theorem K; hence exercise 25 applies.

31.

```
01    START  ENT3   T                    1
02    1H     LD4    H,3                  T
03           ENN2   INPUT-N,4            T
04           ST2    6F(0:2)              T
05           ST2    7F(0:2)              T
06           ST2    4F(0:2)              T
07           ENT2   0,4                  T
08           JMP    9F                   T
09    2H     LDA    INPUT+N,1            NT - S - B + A
10    4H     CMPA   INPUT+N-H,1          NT - S - B + A
11           JGE    8F                   NT - S - B + A
12    6H     LDX    INPUT+N-H,1          B
13           STX    INPUT+N,1            B
14    7H     STA    INPUT+N-H,1          B
15           INC1   0,4                  B
16    8H     INC1   0,4                  NT - B + A
17           J1NP   2B                   NT - B + A
18           DEC2   1                    S
19    9H     ENT1   -N,2                 T + S
20           J2P    8B                   T + S
21           DEC3   1                    T
22           J3P    1B                   T
```

Here A is the number of right-to-left maxima, in the sense that A in Program D is the number of left-to-right minima; both quantities have the same statistical behavior.

The simplifications in the inner loop have cut the running time to $7NT + 7A - 2S + 1 + 15T$ units, curiously independent of B!

When $N = 8$ the increments are 6, 4, 3, 2, 1, and we have $A_{\text{ave}} = 3.892$, $B_{\text{ave}} = 6.762$; the average total running time is $276.24u$. (Cf. Table 5.) Both A and B are maximized in the permutation 7 3 8 4 5 1 6 2. When $N = 1000$ there are 40 increments, 972, 864, 768, 729, . . . , 8, 6, 4, 3, 2, 1; empirical tests like those in Table 6 give $A \approx 875$, $B \approx 4250$, and a total time of about $268000u$ (about twice as long as Program D with the increments of exercise 28).

Instead of storing the increments in an auxiliary table, it is convenient to generate them as follows on a binary machine:

P1. Set $m \leftarrow 2^{\lceil \log_2 N \rceil - 1}$, the largest power of 2 less than N.

P2. Set $h \leftarrow m$.

P3. Use h as the increment for one sorting pass.

P4. If h is even, set $h \leftarrow h + h/2$; then if $h < N$, return to P3.

P5. Set $m \leftarrow \lfloor m/2 \rfloor$ and if $m \geq 1$ return to P2.

Although the increments are not being generated in descending order, the order specified here is sufficient to make the sorting algorithm valid.

32. 4 12 11 13 2 0 8 5 10 14 1 6 3 9 16 7 15.

33. Two separate types of improvements can be made. First, by assuming that the artificial key K_0 is ∞, we can omit testing whether or not $p > 0$. (This idea has been used, for example, in Algorithm 2.2.4A.) Secondly, a standard "optimization" technique: We can make two copies of the inner loop with the register assignments for p and q interchanged; this avoids the assignment $q \leftarrow p$. (This idea has been used in exercise 1.1–3.)

Thus we assume that location INPUT contains the largest possible value in its (0:3) field, and we replace lines 07 and following of Program L by:

```
07   8H   LD3    INPUT,2(LINK)    B'     p ← L_q.  (Here p ≡ rI2, q ≡ rI3.)
08        CMPA   INPUT,3(KEY)     B'
09        JG     4F               B'     To L4 if K > K_p.
10   7H   ST1    INPUT,2(LINK)    N'     L_q ← j.
11        ST3    INPUT,1(LINK)    N'     L_j ← p.
12        JMP    6F               N'     Go to decrease j.
13   4H   LD2    INPUT,3(LINK)    B''    p ← L_q.  (Here p ≡ rI2, q ≡ rI3.)
14        CMPA   INPUT,2(KEY)     B''
15        JG     8B               B''    To L4 if K > K_p.
16   5H   ST1    INPUT,3(LINK)    N''    L_q ← j.
17        ST2    INPUT,1(LINK)    N''    L_j ← p.
18   6H   DEC1   1                N      j ← j − 1.
19        ENT3   0                N      q ← 0.
20        LDA    INPUT,1          N      K ← K_j.
21        J1P    4B               N      N > j ≥ 1.  ∎
```

Here $B' + B'' = B + N - 1$, $N' + N'' = N - 1$, so the total running time is $5B + 14N + N' - 3$ units. (N' is the number of elements with an odd number of

lesser elements to their right; so it has the statistics (min 0, ave $\frac{1}{2}N + \frac{1}{4}H_{\lfloor N/2 \rfloor} - \frac{1}{2}H_N$, max $N - 1$).)

The trick of setting $K_0 \leftarrow -\infty$ makes a similar saving in the running time of Program S, reducing it to $(8B + 11N - 10)u$.

34. There are $\binom{N}{n}$ sequences of N choices in which the given list is chosen n times; each such sequence has probability $(1/M)^n(1 - 1/M)^{N-n}$ of occurring, since the given list is chosen with probability $1/M$.

35. Program L: $A = 3$, $B = 41$, $N = 16$, time $= 496u$. Program M: $A = 2 + 1 + 1 + 3 = 7$, $B = 2 + 0 + 3 + 3 = 8$, $N = 16$, time $= 549u$. (Because of the 16 multiplications. But the improved Program L in exercise 33 takes only $358u$.)

36. To minimize $AC/M + BM$ we need $M = \sqrt{AC/B}$, so M is one of the integers just above or below this quantity. (In the case of Program M we would choose M proportional to N.)

37. The stated identity is equivalent to

$$g_{NM}(z) = M^{-N} \sum_{n_1+\cdots+n_M=N} \left(\frac{N!}{n_1! \ldots n_M!}\right) g_{n_1}(z) \ldots g_{n_M}(z),$$

which is proved as in exercise 34. It may be of interest to tabulate some of these generating functions, to indicate the trend for increasing M:

$$\begin{aligned}
g_{41}(z) &= (216 + 648z + 1080z^2 + 1296z^3 + 1080z^4 + 648z^5 + 216z^6)/5184,\\
g_{42}(z) &= (945 + 1917z + 1485z^2 + 594z^3 + 135z^4 + 81z^5 + 27z^6)/5184,\\
g_{43}(z) &= (1704 + 2264z + 840z^2 + 304z^3 + 40z^4 + 24z^5 + 8z^6)/5184.
\end{aligned}$$

Differentiating the stated identity twice with respect to z and then setting $z = 1$ yields

$$G''_{NM}(1) = M^{-N}\left(M(M-1)\sum_{m,n}\binom{N}{m,\,n,\,N-m-n}\right.$$
$$\left.\times (M-2)^{N-m-n} g'_m(1) g'_n(1) + M\sum_n \binom{N}{n}(M-1)^{N-n} g''_n(1)\right).$$

Using the formula

$$g''_n(1) = \frac{3}{2}\binom{n}{4} + \frac{5}{3}\binom{n}{3},$$

these sums present no special difficulty, and we find that

$$G''_{NM}(1) = \left(\frac{3}{2}\binom{N}{4} + \frac{5}{3}\binom{N}{3}\right) \Big/ M^2.$$

Finally since

$$G'_{NM}(1) = \frac{1}{2}\binom{N}{2} \Big/ M, \qquad G'_{NM}(1)^2 = \left(\frac{3}{2}\binom{N}{4} + \frac{3}{2}\binom{N}{3} + \frac{1}{4}\binom{N}{2}\right) \Big/ M^2,$$

the variance is

$$\left(\frac{1}{6}\binom{N}{3}+\frac{(2M-1)}{4}\binom{N}{2}\right)\Big/ M^2.$$

38. $\sum_{j,n}\binom{N}{n}p_j^n(1-p_j)^{N-n}\binom{n}{2} = \binom{N}{2}\sum_j p_j^2$; setting $p_j = F(j/M) - F((j-1)/M)$, and $F'(x) = f(x)$, this converges to $\binom{N}{2}/M$ times $\int_0^1 f(x)^2\,dx$ when F is reasonably well behaved.

39. (Solution by R. W. Floyd.) A deletion-insertion operation essentially moves only a_i. In a sequence of such operations, unmoved elements retain their relative order. Therefore if π can be sorted with k deletion-insertions, it has an increasing subsequence of length $n - k$; and conversely. Hence dis $(\pi) = n -$ (length of longest increasing subsequence of π). (A longest increasing subsequence can be found in $O(n \log n)$ steps; cf. Section 5.1.4.)

SECTION 5.2.2

1. No, it has $2m + 1$ *less* inversions, where $m \geq 0$ is the number of elements a_k such that $i < k < j$ and $a_i > a_k > a_j$. (Hence *all* exchange-sorting methods will eventually converge to a sorted permutation.)

2. (a) 6. (b) [A. Cayley, *Philos. Mag.* **34** (1849), 527–529.] Consider the cycle representation of π: Any exchange of elements in the *same* cycle increases the number of cycles by 1; any exchange of elements in *different* cycles decreases the number by 1. (This is essentially the content of exercise 2.2.4–3.) A completely sorted permutation is characterized by having n cycles. Hence xch (π) is n minus the number of cycles in π.

3. Yes, equal elements are never moved across each other.

4. The probability that $b_1 > b_2, \ldots, b_n$ in the inversion table, namely

$$\left(\sum_{1\leq k<n} k! \cdot k^{n-k-1}\right) \Big/ n! = \sqrt{\pi/2n} + O(n^{-1}) = \text{negligible.}$$

5. We may assume that $r > 0$. Let $b'_i = (b_i \geq r \Rightarrow b_i - r + 1; 0)$ be the inversion table after $r - 1$ passes. If $b'_i > 0$, element i is preceded by b'_i larger elements, the largest of which will bubble up at least to position $b'_i + i$ (in view of the i elements $\leq i$). Furthermore if element j is the rightmost to be exchanged, we have $b'_j > 0$ and BOUND $= b'_j + j - 1$ after the rth pass.

6. Solution 1: An element displaced farthest to the right of its final position moves one step left on each pass except the last. Solution 2 (higher level): By exercise 5.1.1–8, answer (f), $a'_i - i = b_i - c_i$, for $1 \leq i \leq n$, where $c_1\, c_2 \ldots c_n$ is the dual inversion table. If $b_j = \max(b_1, \ldots, b_n)$ then $c_j = 0$.

7. $(2(n+1)(1 + P(n) - P(n+1)) - P(n) - P(n)^2)^{1/2} = \sqrt{(2 - \pi/2)n} + O(1)$.

8. For $i < k + 2$ there are $j + k - i + 1$ choices for b_i; for $k + 2 \leq i < n - j + 2$ there are $j - 1$ choices; and for $i \geq n - j + 2$ there are $n - i + 1$.

10. (a) If $i = 2k - 1$, from $(k - 1, a_i - k)$ to $(k, a_i - k)$. If $i = 2k$, from $(a_i - k, k - 1)$ to $(a_i - k, k)$. (b) Step a_{2k-1} is above the diagonal iff $k \leq a_{2k-1} - k$ iff $a_{2k-1} \geq 2k$ iff $a_{2k-1} > a_{2k}$ iff $a_{2k} \leq 2k - 1$ iff $a_{2k} - k \leq k - 1$ iff step a_{2k} is above the diagonal. Exchanging them interchanges horizontal and vertical steps. (c) Step a_{2k+d} is at least m below the diagonal iff $k + m - 1 \geq a_{2k+d} - (k + m) + m$ iff

$a_{2k+d} < 2k + m$ iff $a_{2k} \geq 2k + m$ iff $a_{2k} - k \geq k + m$ iff step a_{2k} is at least m below the diagonal. (If $a_{2k+d} < 2k + m$ and $a_{2k} < 2k + m$, there are at least $(k + m) + k$ elements less than $2k + m$; that's impossible. If $a_{2k+d} \geq 2k + m$ and $a_{2k} \geq 2k + m$, one of the $\geq$ must be $>$; but we can't fit all of the elements $\leq 2k + m$ into less than $(k + m) + k$ positions. Hence $a_{2k+d} < a_{2k}$ iff $a_{2k+d} < 2k + m$ iff $2k + m \leq a_{2k}$. A rather unexpected result!)

11. 16 10 13 5 14 6 9 2 15 8 11 3 12 4 7 1 (61 exchanges), by considering the lattice diagram. In general, when $N = 2^t$ the set $\{K_2, K_4, \ldots\}$ should be $\{1, 2, \ldots, \lfloor N/3 \rfloor + 1, \lfloor N/3 \rfloor + 3, \lfloor N/3 \rfloor + 5, \ldots\}$, permuted so as to maximize the exchanges for $N/2$ elements.

12. In this program, TT $\equiv 2^{t-1}$, $p \equiv$ rI1, $r \equiv$ rI2, $i \equiv$ rI3, $i + d - N \equiv$ rI4; assume that $N \geq 2$.

```
01       START   ENT1    TT              1       M1. Initialize p. p ← 2^(t-1).
02               ST1     P               1
03       2H      ENT2    TT              T       M2. Initialize q, r, d.
04               ST2     Q(1:2)          T       q ← 2^(t-1).
05               ENT2    0               T       r ← 0.
06               ENT4    0,1             T       rI4 ← p.
07       3H      DEC4    N               A       M3. Loop on i.
08               ENT3    0               A       i ← 0.
09       8H      ENTA    0,3             C+E     (At this point rI4 < 0.)
10               AND     P               C+E     rA ← i ∧ p.
11               DECA    0,2             C+E
12               JAZ     4F              C+E     Jump if i ∧ p = r.
13               INC3    0,1             D       Otherwise i ← i + p.
14               INC4    0,1             D
15               J4NN    5F              D       Exit loop if i + d ≥ N.
16       4H      LDA     INPUT+1,3       C       M4. Compare/exchange
17               CMPA    INPUT+N+1,4     C         R_{i+1} : R_{i+d+1}.
18               JLE     *+4             C
19               LDX     INPUT+N+1,4     B
20               STX     INPUT+1,3       B
21               STA     INPUT+N+1,4     B
22               INC3    1               C       i ← i + 1.
23               INC4    1               C
24               J4N     8B              C       Repeat loop if i + d < N.
25       5H      ENT2    0,1             A       M5. Loop on q. r ← p.
26       Q       ENT4    *               A       rI4 ← q.
27               ENTA    0,4             A
28               SRB     1               A
29               STA     Q(1:2)          A       q' ← q/2.
30               DEC4    0,1             A       rI4 ← q - p.
31               J4P     3B              A       To M3 if q ≠ p.
32       6H      ENTA    0,1             T       M6. Loop on p.
33               SRB     1               T
34               STA     P               T
35               LD1     P               T       p ← ⌊p/2⌋.
36               JANZ    2B              T       To M2 if p ≠ 0.  ∎
```

The running time depends on six quantities, only one of which depends on the input data (the remaining five are functions of N alone): $T = t$, the number of "major cycles"; $A = t(t+1)/2$, the number of passes or "minor cycles"; B = the (variable) number of exchanges; C = the number of comparisons; D = the number of times $i \wedge p \neq r$ in step M2; and E = the number of times $i \wedge p \neq r$ and $i + p + d \geq N$ in step M2. When $N = 2^t$, it can be shown that $D = 2^t(t-2) + 2 + E$, and $E = \binom{t}{2}$. For Table 1, $T = 4$, $A = 10$, $B = 3+0+1+4+0+0+8+0+4+5 = 25$, $C = 63$, $D = 40$, $E = 6$, so the total running time is $11A + 6B + 13C + 3D + 5E + 13T + 3 = 1284u$.

13. No, nor are Algorithms Q, R.

14. (a) When $p = 1$ we do $(2^{t-1} - 0) + (2^{t-1} - 1) + (2^{t-1} - 2) + (2^{t-1} - 4) + \cdots + (2^{t-1} - 2^{t-2}) = (t-1)2^{t-1} + 1$ comparisons for the final merge. (b) $x_t = x_{t-1} + \frac{1}{2}(t-1) + 2^{-t} = \cdots = x_0 + \sum_{0 \leq k < t} (\frac{1}{2}k + 2^{-k-1}) = \frac{1}{2}\binom{t}{2} + 1 - 2^{-t}$. Hence $c(2^t) = 2^{t-2}(t^2 - t + 4) - 1$.

15. (a) Consider the number of comparisons such that $i + d = N$; then use induction on r. (b) If $b(n) = c(n+1)$, we have $b(2n) = a(1) + \cdots + a(2n) = a(0) + a(1) + a(1) + \cdots + a(n-1) + a(n) + x(1) + x(2) + \cdots + x(2n) = 2b(n) + y(2n) - a(n)$; similarly $b(2n+1) = 2b(n) + y(2n+1)$. (c) Cf. exercise 1.2.4–42. (d) A rather laborious calculation of $(z(N) + 2z(\lfloor N/2 \rfloor) + \cdots) - a(N)$, using formulas such as

$$\sum_{0 \leq k \leq n} 2^k(n-k) = 2^{n+1} - n - 2, \qquad \sum_{0 \leq k \leq n} 2^k \binom{n-k}{2} = 2^{n+1} - \binom{n+2}{2} - 1,$$

leads to the result

$$c(N) = N\left(\frac{1}{2}\binom{e_1}{2} + 2e_1 - 1\right) - 2^{e_1}(e_1 - 1) - 1 + \sum_{1 \leq j \leq r} 2^{e_j}\left(e_1 + \cdots + e_{j-1} - j(e_1 - 1) + \frac{1}{2}\binom{e_1 - e_j}{2}\right).$$

16. Consider the $\binom{2n}{n}$ lattice paths from $(0, 0)$ to (n, n). For $k \geq 0$, the number of times a line $(i, i+k) - (i, i+k+1)$ is traversed for some i, summed over all paths, is

$$\sum_i \binom{k+2i}{i}\binom{2n-1-k-2i}{n-i},$$

the coefficient of x^{2n} in $x^{k+1}y^{2k+1}/(1-4x)$, where $y = (1 - \sqrt{1-4x})/2x$. (Cf. exercise 5.2.1–14.) It can now be shown that the number of exchanges during the final merge when $N = 2n$, summed over all paths, is the coefficient of x^N in

$$\frac{x}{(1-4x)^{3/2}} \frac{1}{1-z}\left(1 + \frac{2z}{1+z^2} + \frac{2z^2}{1+z^4} + \frac{2z^4}{1+z^8} + \cdots\right), \qquad z = xy^2 = y - 1.$$

The asymptotic value of this coefficient is desired.

17. K_{N+1} is inspected when we are sorting a subfile with $r = N$ and K_l the largest key. K_0 is inspected during a straight-insertion sorting phase with $l = 1$, when left-to-right minima sink to position R_1.

18. Each comparison is followed by a transfer; the partitioning process for $R_l \ldots R_r$ ends with $i = \lceil (l+r)/2 \rceil$ in step Q7, bisecting the subfile as perfectly as possible. (Quantitatively speaking, we replace Eqs. (17) by $A = 1$, $B = \lfloor (N-1)/2 \rfloor$, $C = N$, $X = 1 - (N \bmod 2)$; this puts us essentially in the *best* case of the algorithm, as in exercise 27 below, except that $B \approx \frac{1}{2}C$.) If the "$<$" signs in steps Q3 and Q5 are changed, we need "$<$" signs in (13); and the algorithm behaves terribly, letting j run from N down to 0 each time and making the slowest conceivable progress. Thus the extra transfers are well worth making. [Another way to handle equal keys is suggested in exercise 30.]

19. Yes, the other subfiles may be processed in any order (without increasing the size of the storage area); but a stack is easiest to work with.

20. $\lfloor \log_2 (N+1)/(M+2) \rfloor$. (The worst case occurs when $N = 2^k(M+1) - 1$ and all subfiles are perfectly bisected when they are partitioned.)

21. If $s > 1$, the first occurrence of step Q4 removes s from the keys in memory, and no comparison in steps Q3 and Q5 will come out $=$. Exactly t records are transferred from the area $R_2 \ldots R_s$ to $R_{s+1} \ldots R_N$; and $t + h$ are transferred from $R_s \ldots R_N$ to $R_1 \ldots R_{s-1}$. If $h = 1$, the last comparison is $K_i{:}K$ with $i = s+1$ and $j = s$; if $h = 0$ it is $K{:}K_j$ with $j = s - 1$ and $i = s$. So we obtain (17); in fact, $C' = N + 2 - s - h - \delta_{s1}$ and $C'' = s - 1 + h$.

22. The stated relations for $A_N(z)$ follow easily because $A_{s-1}(z)\,A_{N-s}(z)$ is the generating function for the value of A after independently sorting randomly ordered files of sizes $s - 1$ and $N - s$. Similarly, we obtain the relations

$$B_N(z) = \sum_{1 \le s \le N} \sum_{0 \le k \le s} b_{stN}\, z^t\, B_{s-1}(z) B_{N-s}(z),$$

$$C_N(z) = \frac{1}{N} \sum_{1 \le s \le N} z^{N+1-\delta_{s1}} C_{s-1}(z) C_{N-s}(z),$$

$$D_N(z) = \frac{1}{N} \sum_{1 \le s \le N} D_{s-1}(z) D_{N-s}(z),$$

$$E_N(z) = \frac{1}{N} \sum_{1 \le s \le N} E_{s-1}(z) E_{N-s}(z),$$

$$L_N(z) = \frac{1}{N} \sum_{1 \le s \le N} L_{s-1}(z) L_{N-s}(z),$$

$$X_N(z) = \frac{1}{N} \sum_{1 \le s \le N} (1 + (z-1)h_{sN}) X_{s-1}(z) X_{N-s}(z),$$

for $N > M$. Here b_{stN} is the probability that s and t have given values in a file of length N, namely

$$\binom{s-1}{t}\binom{N-s}{t} \Big/ N\binom{N-1}{s-1},$$

which is $(1/N!)$ times the $(s-1)!$ ways to permute $\{1, \ldots, s-1\}$ times the $(N-s)!$

ways to permute $\{s+1, \ldots, N\}$ times the $\binom{s-1}{t}\binom{N-s}{t}$ patterns with t displaced elements on each side; and h_{sN} is the probability that $h = 1$ when N and s are given, namely $(s-1)/(N-1)$. For $0 \le N \le M$, we have $B_N(z) = C_N(z) = X_N(z) = 1$; $L_N(z) = 1$, except $L_0(z) = L_1(z) = z$; $D_N(z) = z^{N-1}$, except $D_0(z) = 1$; and $E_N(z)$ is $\prod_{1\le k\le N} ((1+z+\cdots+z^{k-1})/k)$.

23. When $N > M$, $A_N = 1 + (2/N)\sum_{0\le k<N} A_k$; $B_N = \sum_{0\le t<s\le N} b_{stN}(t + B_{s-1} + B_{N-s}) = (1/N)\sum_{1\le s\le N} ((s-1)(N-s)/(N-1) + B_{s-1} + B_{N-s}) = (N-2)/6 + (2/N)\sum_{0\le k<N} B_k$ [cf. exercise 22]; $D_N = (2/N)\sum_{0\le k<N} D_k$; E_N and L_N are similar; $X_N = (1/N)\sum_{1\le s\le N} (h_{sN} + X_{s-1} + X_{N-s}) = \frac{1}{2} + (2/N)\sum_{0\le k<N} X_k = \frac{1}{2}A_N$. Each of these recurrences has the form (20) for some function f_n.

24. The recurrence $C_N = N - 1 + (2/N)\sum_{0\le k<N} C_k$, $N > M$, has the solution $(N+1)\big(2H_{N+1} - 2H_{M+2} + 1 - 4/(M+2) + 2/(N+1)\big)$, $N > M$. (So we save about $4N/M$ comparisons, but each comparison takes longer.)

25. (Use (17) repeatedly with $s = 1$.) $A = N - M$, $B = 0$, $C = \binom{N+1}{2} - \binom{M+1}{2}$, $D = M - 1$, $E = 0$, $L = N - M + \delta_{M1}$, $X = 0$.

26. $N\ M\ (M-1) \ldots 1\ (M+1)\ (M+2) \ldots (N-1)$. Using (17) repeatedly with $s = N$, we have $A = N - M$, $B = 0$, $C = \binom{N+2}{2} - \binom{M+2}{2}$, $D = M - 1$, $E = \binom{M}{2}$, $L = N - M + \delta_{M1}$, $X = N - M$. (Since $B = 0$ this may not be the worst possible case.)

27. Eq. (17) with $A = 1$, $B = 0$, $C = N + 1$, $X = 1$, and $s = \lfloor (N+1)/2 \rfloor$ is best for the partitioning; and $E = 0$ is best for straight insertion. When $N = (M+1)2^k - 1$, the totals are $A = X = 2^k - 1$, $B = E = 0$, $C = k(M+1)2^k$, $D = 2^k(M-1)$; when $N = (M+2)2^k - 1$, $A = X = 2^{k+1} - 1$, $C = (k+1)(M+2)2^k$, $D = 2^{k+1}\lfloor (M-1)/2 \rfloor$. In general, the approximate values are $A = X \approx$ between N/M and $2N/M$; $B = E = 0$; $C \approx N \log_2 (N/M)$; $D \approx N$.

28. The recurrence

$$C_n = n + 1 + \frac{2}{\binom{n}{3}} \sum_{1\le k\le n} (k-1)(n-k)C_{k-1}$$

can be transformed into

$$\binom{n}{3}C_n - 2\binom{n-1}{3}C_{n-1} + \binom{n-2}{3}C_{n-2} = 2(n-1)(n-2) + 2(n-2)C_{n-2}.$$

29. In general, consider the recurrence

$$C_n = n + 1 + \frac{2}{\binom{n}{2t+1}} \sum_{1\le k\le n} \binom{k-1}{t}\binom{n-k}{t} C_{k-1},$$

which arises when the median of $2t+1$ elements governs the partitioning. Letting $C(z) = \sum_n C_n z^n$, the recurrence can be transformed to $(1-z)^{t+1}C^{(2t+1)}(z)/(2t+2)! = 1/(1-z)^{t+2} + C^{(t)}(z)/(t+1)!$. Let $f(x) = C^{(t)}(1-x)$; then $p_t(\vartheta)f(x) = (2t+2)!/(x)^{\underline{t+2}}$, where ϑ denotes the operator $x(d/dx)$, and $p_t(x) = (t-x)^{\underline{t+1}} - (2t+2)^{\underline{t+1}}$. The general solution to $(\vartheta - \alpha)g(x) = x^\beta$ is $g(x) = x^\beta/(\beta-\alpha) + Cx^\alpha$, for $\alpha \ne \beta$; $g(x) = x^\beta(\ln x + C)$ for $\alpha = \beta$. We have $p_t(-t-2) = 0$; so the general solution to our differential equation is

$$C^{(t)}(z) = (2t+2)!\ln(1-z)/p_t'(-t-2)(1-z)^{t+2} + \sum_{0\le j\le t} c_j(1-z)^{\alpha_j}$$

where $\alpha_0, \ldots, \alpha_t$ are the roots of $p_t(x) = 0$, and the constants c_i depend on the initial values $C_t, \ldots, C_{2t}$. The handy identity

$$\frac{1}{(1-z)^{m+1}} \ln\left(\frac{1}{1-z}\right) = \sum_{n\ge 0} (H_{n+m} - H_m)\binom{n+m}{m} z^n, \qquad m \ge 0,$$

now leads to the surprisingly simple *closed form solution*

$$C_n = (H_{n+1} - H_{t+1})(n+1)/(H_{2t+2} - H_{t+1}) + \left(\sum_{0\le j\le t} c_j(-\alpha_j)^{\overline{n-t}}\right)/n!,$$

from which the asymptotic formula is easily deduced. (The leading term $n \ln n/(H_{2t+2} - H_{t+1})$ was discovered by M. H. van Emden [*CACM* **13** (1970), 563–567] using an information-theoretic approach. In fact, suppose we wish to analyze any partitioning process such that the left subfile contains at most xN elements with asymptotic probability $\int_0^x f(x)\,dx$, as $N \to \infty$, for $0 \le x \le 1$; van Emden has proved that the average number of comparisons required to sort the file completely is $\sim n \ln n/\alpha$, where $\alpha = -1/\int_0^1 \big(f(x) + f(1-x)\big)x \ln x\,dx$. This formula applies to radix exchange as well as quicksort and various other methods. See also H. Hurwitz, *CACM* **14** (1971), 99–102.)

30. Each subfile may be identified by four quantities (l, r, k, X), where l and r are the boundaries (as presently), k indicates the number of words of the keys which are known to be equal throughout the subfile, and X is a lower bound for the $(k+1)$st words of the keys. Assuming nonnegative keys, we have $(l, r, k, X) = (1, N, 0, 0)$ initially. When partitioning a file, we let K be the $(k+1)$st word of the test key K_q. If $K > X$, partitioning takes place with all keys $\ge K$ at the right and all keys $< K$ at the left (looking only at the $(k+1)$st word of the key each time; the partitioned subfiles get the respective identifications $(l, i-1, k, X)$ and (i, r, k, K). But if $K = X$, partitioning takes place with all keys $> K$ at the right and all keys $\le K$ [actually $= K$] at the left; the partitioned subfiles get the respective identifications $(l, i, k+1, 0)$ and $(i+1, r, k, K)$. In both cases we are unsure that R_i is in its final position since we haven't looked at the $(k+2)$nd words. Obvious further changes are made to handle boundary conditions properly. By adding a fifth "upper bound" component, the method could be made symmetrical between left and right.

31. Go through a normal partitioning process, with R_1 finally falling into position R_k. If $k = m$, stop; if $k > m$, use the same technique to find the $(m-k)$th smallest element of the right-hand subfile; and if $k < m$ find the mth smallest element of the left-hand subfile. [*CACM* **4** (1961), 321–322.]

32. The recurrence is $C_{nm} = n - 1 + (A_{nm} + B_{nm})/n$, where

$$A_{nm} = \sum_{1\le k<m} C_{n-k,m-k} \qquad \text{and} \qquad B_{nm} = \sum_{m\le k<n} C_{km},$$

for $1 \le m \le n$. Since $A_{n+1,m+1} = A_{nm} + C_{nm}$ and $B_{n+1,m} = B_{nm} + C_{nm}$, we can first find a formula for $(n+1)C_{n+1,m+1} - nC_{nm}$, then sum this to obtain the answer $2\big((n+1)H_n - (n+3-m)H_{n+1-m} - (m+2)H_m + n + 3\big)$. When $n = 2m-1$,

it becomes $4m(H_{2m-1} - H_m) + 4m - 8H_m + 4 = (4 + 4\ln 2)m - 8\ln m - 8\gamma + 1 + O(m^{-1}) \approx 3.39n$.

33. Proceed as in the first stage of radix exchange, using the sign instead of bit 1.

34. We can avoid testing whether or not $i \le j$, as soon as we have found at least one 0 bit and at least one 1 bit in each stage, i.e., after making the first exchange in each stage. This saves approximately $2C$ units of time in Program R.

35. $A = N - 1$, $B = (\min 0,\ \text{ave}\ \frac{1}{4}N\log_2 N,\ \max \frac{1}{2}N\log_2 N)$, $C = N\log_2 N$, $G = \frac{1}{2}N$, $K = L = R = 0$, $S = \frac{1}{2}N - 1$, $X = (\min 0, \text{ave}\ \frac{1}{2}(N-1), \max N - 1)$. [It is interesting to note that, in general, the quantities A, C, G, K, L, R, and S depend only on the set of keys in the file, not on their initial order; only B and X are influenced by the initial order of the keys.]

36. (a) $\sum\binom{n}{k}\binom{k}{j}(-1)^{k+j}a_j = \sum\binom{n}{j}\binom{n-j}{k-j}(-1)^{k-j}a_j = \sum\binom{n}{j}\delta_{nj}a_j = a_n$. (b) $\langle\delta_{n0}\rangle$; $\langle-\delta_{n1}\rangle$; $\langle(-1)^n\delta_{nm}\rangle$; $\langle(1-a)^n\rangle$; $\langle\binom{n}{m}(-a)^m(1-a)^{n-m}\rangle$. (c) Writing the relations to be proved as $x_n = y_n = a_n + z_n$, we have $y_n = a_n + z_n$ by part (a); also $2^{1-n}\sum_{k\ge 2}\binom{n}{k}y_k = z_n$, so y_n satisfies the same recurrence as x_n. [See exercises 53 and 6.3–17 for some generalizations of this result. It does not appear to be easy to prove *directly* that $\hat{x}_n = \hat{a}_n 2^{n-1}/(2^{n-1} - 1)$.]

37. $\langle\sum_m c_m\binom{n}{2m}2^{-n}\rangle$ for an arbitrary sequence of constants $c_0, c_1, c_2, \ldots$. [This answer, although correct, does not reveal immediately that $\langle 1/(n+1)\rangle$ and $\langle n - \delta_{n1}\rangle$ are such sequences! Sequences having the form $\langle a_n + \hat{a}_n\rangle$ are always self-dual. Note that, in terms of the generating function $A(z) = \sum a_n z^n/n!$, we have $\hat{A}(z) = e^z A(-z)$; hence $A = \hat{A}$ is equivalent to saying that $A(z)e^{-z/2}$ is an even function.

38. A partitioning stage which yields a left subfile of size s and a right subfile of size $N - s$ makes the following contributions to the total running time:

$$A = 1, \quad B = t, \quad C = N, \quad K = \delta_{s1}, \quad L = \delta_{s0}, \quad R = \delta_{sN}, \quad X = h,$$

where t is the number of keys $K_1, \ldots, K_s$ with bit b equal to 1, and h is bit b of K_{s+1}; if $s = N$, then $h = 0$. (Cf. (17).) This leads to recurrence equations such as

$$B_N = 2^{-N}\sum_{0\le t\le s\le N}\binom{s}{t}\binom{N-s}{t}(t + B_s + B_{N-s})$$

$$= \frac{1}{4}(N-1) + 2^{1-N}\sum_{s\ge 2}\binom{N}{s}B_s, \qquad \text{for } N \ge 2;\quad B_0 = B_1 = 0.$$

(Cf. exercise 23.) Solving these recurrences by the method of exercise 36 yields the formulas $A_N = V_N - U_N + 1$, $B_N = \frac{1}{4}(U_N + N - 1)$, $C_N = V_N + N$, $K_N = N/2$, $L_N = R_N = \frac{1}{2}(V_N - U_N - N) + 1$, $X_N = \frac{1}{2}(A_N - L_N)$. Clearly $G_N = 0$.

39. Each stage of quicksort puts at least one element into its final position, but this need not happen during radix exchange (cf. Table 3).

40. If we switch to straight insertion whenever $r - l < M$ in step R2, the problem doesn't arise unless more than M equal elements occur. If the latter is a likely prospect, we can test whether or not $K_l = \cdots = K_r$ whenever $j < l$ or $j = r$ in step R8.

43. As $a \to 0+$, $\int_0^1 y^{a-1}(e^{-y} - 1)\,dy + \int_1^\infty y^{a-1}e^{-y}\,dy = \Gamma(a) - 1/a = \big(\Gamma(a+1) - \Gamma(1)\big)/a \to \Gamma'(1) = -\gamma$, by exercise 1.2.7–24.

44. For $k \geq 0$, we have $r_k(m) \sim \frac{1}{2}(2m)^{(k+1)/2}\Gamma\big((k+1)/2\big) - \sum_{j\geq 0}(-1)^j B_{k+2j+1}/\big((k+2j+1)j!(2m)^j\big)$. When $k = -1$, the contributions from $f_k^{(j-1)}(m)$ in (36) cancel with similar terms in the expansion of H_{m-1}, and we have $r_{-1}(m) = H_{m-1} + (1/\sqrt{2m})\sum_{t\geq 0} f_{-1}(t) \sim \frac{1}{2}\big(\ln(2m) + \gamma\big) - \sum_{j\geq 1}(-1)^j B_{2j}/(2j)j!(2m)^j$. Therefore the contribution to W_{m-1} from the term N^t/t of (33) is obtained from $m\sum_{t\geq 1} t^{-1}\exp(-t^2/2m)(1 - t^3/3m^2 + t^6/18m^4)(1 - t^4/4m^3)(1 - t/2m - t^2/8m^2) + O(m^{-1/2}) = \frac{1}{2}m\ln m + \frac{1}{2}(\ln 2 + \gamma)m - \frac{5}{12}\sqrt{2\pi m} + \frac{4}{9} + O(m^{-1/2})$. The term $-\frac{1}{2}N^{t-1}$ contributes $-\frac{1}{2}\sum_{t\geq 1}\exp(-t^2/2m)(1 - t^3/3m^2)(1 - t/2m)(1 + t/m) + O(m^{-1/2}) = -\frac{1}{4}\sqrt{2\pi m} - \frac{1}{6}$. The term $\frac{1}{2}\delta_{t1}$ yields $\frac{1}{2}$. And finally the term $\frac{1}{2}(t-1)B_2N^{t-2}$ contributes $\frac{1}{12}m^{-1}\sum_{t\geq 1} t\exp(-t^2/2m) + O(m^{-1/2}) = \frac{1}{12} + O(m^{-1/2})$.

45. The argument used to derive (42) is also valid for (43), except that we leave out the residues at $z = -1$ and $z = 0$.

46. Proceeding as we did with (45), we obtain $(s-1)!/\ln 2 + f_s(n)$, where

$$f_s(n) = \frac{2}{\ln 2}\sum_{k\geq 1}\Re\big(\Gamma(s - 2\pi ik/\ln 2)\exp(2\pi ik\log_2 n)\big).$$

[Note that $|\Gamma(s+it)|^2 = \big(\prod_{0\leq k<s}(k^2+t^2)\big)\pi/(t\sinh\pi t)$, for integer $s \geq 0$, so we can bound $f_s(n)$.]

47. In fact, $\sum_{j\geq 1} e^{-n/2^j}(n/2^j)^s$ equals the integral in exercise 46, for all $s > 0$.

48. Making use of the intermediate identity

$$1 - e^{-x} = \frac{-1}{2\pi i}\int_{-1/2-i\infty}^{-1/2+i\infty}\Gamma(z)x^{-z}\,dz,$$

we proceed as in the text, with $1 - e^{-x}$ playing the role of $e^{-x} - 1 + x$; $V_{n+1}/(n+1) = (-1/2\pi i)\int_{-1/2-i\infty}^{-1/2+i\infty}\Gamma(z)n^{-z}\,dz/(2^{-z}-1) + O(n^{-1})$, and the integral equals $\log_2 n + \gamma/(\ln 2) - \frac{1}{2} - f_0(n)$ in the notation of exercise 46. [Thus A_N in exercise 38 is $N\big(1/(\ln 2) - f_0(N-1) - f_{-1}(N)\big) + O(1)$.]

49. The right-hand side of Eq. (40) can be improved to $e^{-x}\big(n/x + \frac{1}{2}x + x^3O(n^{-1})\big)$. The effect is to subtract $\frac{1}{2}$ times the sum in exercise 47, replacing $O(1)$ in (47) by $2 - \frac{1}{2}\big(1/(\ln 2) + f_1(n)\big) + O(n^{-1})$. (The "2" comes from the "$2/n$" in (45).)

50. $U_{mn} = n\log_m n + n\big((\gamma-1)/(\ln m) - \frac{1}{2} + f_{-1}(n)\big) + m/(m-1) - 1/(2\ln m) - \frac{1}{2}f_1(n) + O(n^{-1})$, where $f_s(n)$ is defined as in exercise 46 but with $\ln 2$ and $\log_2$ replaced by $\ln m$ and $\log_m$. [*Note:* For $m = 2, 3, 4, 5, 10, 100, 1000, 10^6$ we have $f_{-1}(n) <$ .0000001725, .00041227, .000296, .00085, .00627, .068, .153, .341, respectively.]

51. Let $N = 2m$. We may extend the sum (35) over all $t \geq 1$, when it equals

$$\sum_{t\geq 1}(1/2\pi i)\int_{a-i\infty}^{a+i\infty}\Gamma(z)(t^2/N)^{-z}t^k\,dz = (1/2\pi i)\int_{a-i\infty}^{a+i\infty}\Gamma(z)N^z\zeta(2z-k)\,dz,$$

provided that $a > (k+1)/2$. So we need to know properties of the zeta function. When $\Re(w) \geq -q$, $\zeta(w) = O(|w|^{q+1})$ as $|w| \to \infty$; hence we can shift the line of integration to the left as far as we please if we only take the residues into account. The factor $\Gamma(z)$ has poles at $0, -1, -2, \ldots$, and $\zeta(2z-k)$ has a pole only at $z = (k+1)/2$. The residue at $z = -j$ is $N^{-j}(-1)^j\zeta(-2j-k)/j!$, and $\zeta(-n) =$

$(-1)^{n+1}B_{n+1}/(n+1)$. The residue at $z = (k+1)/2$ is $\frac{1}{2}\Gamma\big((k+1)/2\big)N^{(k+1)/2}$. But when $k = -1$ there is a double pole at $z = 0$; $\zeta(z) = 1/(z-1) + \gamma + O(|z-1|)$, so the residue at 0 in this case is $\gamma + \frac{1}{2}\ln N - \frac{1}{2}\gamma$. We therefore obtain the asymptotic series mentioned in the answer to exercise 44.

52. Set $x = t/n$; then

$$\binom{2n}{n+t}\Big/\binom{2n}{n} = \exp\big(-2n(x^2/1\cdot 2 + x^4/3\cdot 4 + \cdots) + (x^2/2 + x^4/4 + \cdots) - (1/6n)(x^2 + x^4 + \cdots) + \cdots\big);$$

the desired sum can now be expressed in terms of $\sum_{t\ge 1} t^k\, d(t)e^{-t^2/n}$, for various k. Proceeding as in exercise 51, since $\zeta(z)^2 = \sum_{t\ge 1} d(t)t^{-z}$, we wish to evaluate the residues of $\Gamma(z)n^z\zeta(2z-k)^2$ when $k \ge 0$. At $z = -j$ the residue is $n^{-j}(-1)^j\big(B_{2j+k+1}/(2j+k+1)\big)^2/j!$, and at $z = (k+1)/2$ the residue is $n^{(k+1)/2}\Gamma\big((k+1)/2\big)\big(\gamma + \frac{1}{4}\ln n + \frac{1}{4}\psi\big((k+1)/2\big)\big)$, where $\psi(z) = \Gamma'(z)/\Gamma(z) = H_{z-1} - \gamma$; thus, for example, when $k = 0$, $\sum_{t\ge 1}e^{-t^2/n}\,d(t) = \frac{1}{4}\sqrt{\pi n}\ln n + (\frac{3}{4}\gamma - \frac{1}{2}\ln 2)\sqrt{\pi n} + \frac{1}{4} + O(n^{-M})$ for all M. For $S_n/\binom{2n}{n}$, add $(\frac{1}{32}\ln n + \frac{3}{32}\gamma + \frac{1}{24} - \frac{1}{16}\ln 2)\sqrt{\pi/n} + O(n^{-1})$ to this quantity. (Cf. exercise 1.2.7–23, 1.2.9–19.)

53. Let $q = 1 - p$. Generalizing exercise 36(c), if

$$x_n = a_n + \sum_{k\ge 2}\binom{n}{k}(p^k q^{n-k} + q^k p^{n-k})x_k,$$

then

$$x_n = a_n + \sum_{k\ge 2}\binom{n}{k}(-1)^k \hat{a}_k(p^k + q^k)/(1 - p^k - q^k).$$

We can therefore find B_N and C_N as before (the factor $\frac{1}{4}$ in B_N should be replaced by pq). The asymptotic examination of U_N proceeds essentially as in the text, with

$$\begin{aligned} T_n &= \sum_{r\ge 1,\, s\ge 0}\binom{r}{s}(e^{-np^sq^{r-s}} - 1 + np^sq^{r-s}) \\ &= (1/2\pi i)\int_{-3/2-i\infty}^{-3/2+i\infty}\Gamma(z)n^{-z}(p^{-z} + q^{-z})\,dz/(1 - p^{-z} - q^{-z}) \\ &= (n/h_p)\big(\ln n + \gamma - 1 + h_p^{(2)}/2h_p - h_p + f(n)\big) + O(1), \end{aligned}$$

where

$$h_p = -(p\ln p + q\ln q), \qquad h_p^{(2)} = p(\ln p)^2 + q(\ln q)^2,$$

and

$$f(n) = \sum \Gamma(z)n^{-1-z}/h_p$$

summed over all complex $z \ne -1$ such that $p^{-z} + q^{-z} = 1$. The latter set of points seems to be difficult to analyze in general; but when $p = \phi^{-1}$, $q = \phi^{-2}$, the solutions are $z = (-1)^{k+1} + k\pi i/(\ln\phi)$. The dominant term, $(n\ln n)/h_p$, could also have been obtained from van Emden's general formula quoted in the answer to exercise 29. For $p = \phi^{-1}$ we have $1/h_p = 1.521$, compared to $1/h_{1/2} = 1.4427$.

54. Let C be a circle of radius $(M+\frac{1}{2})b$, so that the integral vanishes on C as $M\to\infty$. (The asymptotic form of U_n can now be derived in a new way, expanding $\Gamma(n+1)/\Gamma(n+ibm)$. The method of this exercise applies to *all* sums of the form $\sum_k\binom{n}{k}(-1)^k f(k)$, when f is reasonably well behaved!)

55. Analysis shows that it is actually a little better to use $\lceil(l+r)/2\rceil$ in place of $\lfloor(l+r)/2\rfloor$, because of the way Program Q is written. Thus, we replace lines 09 and 10 of Program Q by

```
ENTA  0,3                 LDA   INPUT,5               JGE   Line13
INCA  1,4                 LDX   INPUT,3               LDA   INPUT,5
SRB   1                   CMPA  INPUT,4               LDX   INPUT,3
STA   TEMP                JL    5F                    CMPA  INPUT,4
LD5   TEMP                STX   INPUT,5               JG    5F
LDA   INPUT,3             JMP   Line19                STX   INPUT,5
CMPA  INPUT,4        5H   LDA   INPUT,4               JMP   Line13
JL    1F                  STX   INPUT,4          5H   LDA   INPUT,4
CMPA  INPUT,5             JMP   Line13                STX   INPUT,4
JLE   Line19         1H   CMPA  INPUT,5               JMP   Line21
```

The first three of these lines should be replaced by "`ENTX 0,3; INCX 1,4; ENTA 0; DIV =2=`" if binary shifting is not available. In this code, "`Line19`" refers to line 19 of Program Q, etc.

56. We may solve the recurrence $\binom{n}{3}x_n = b_n + 2\sum_{1\le k\le n}(k-1)(n-k)x_{k-1}$, $n>m$, by letting $y_n = nx_n$, $u_n = ny_{n+1} - (n+2)y_n$, $v_n = nu_{n+1} - (n-5)u_n$; it follows that $v_n = 6(b_{n+2} - 2b_{n+1} + b_n)$, for $n>m$. *Example:* Let $x_n = \delta_{n1}$ for $n\le m$, and let $b_n\equiv 0$. Then $v_n = 0$ for all $n>m$, hence $n^{\underline{5}}u_{n+1} = m^{\underline{5}}u_{m+1}$. Since $y_{m+1} = 12/m$, $y_{m+2} = 12/(m+1)$, we ultimately find $x_n = \frac{48}{7}(n+1)/m(m+1)(m+2) + \frac{36}{7}(m-1)^{\underline{4}}/n^{\underline{6}}$ for $n>m$. In general, when b_n is identically zero, let $f_n = \big(12/(n-1)(n-2)\big)\sum_{1\le k\le n}(k-1)(n-k)x_{k-1}$; the solution for $n>m$ is $x_n = (n+1)\big((m+1)f_{m+2} - (m-4)f_{m+1}\big)/7(m+1)(m+2) + \big((m+1)f_{m+2} - (m+3)f_{m+1}\big)m^{\underline{5}}/7n^{\underline{6}}$. When $b_n = \binom{n}{3}/n^{\underline{p}}$ and $x_n = 0$ for $n\le m$, the solution is

$$x_n/(n+1) = (p-3)(p-2)/(p-6)(p+1)(n+1)^{\underline{p+1}} + 12/7(p+1)(m+2)^{\underline{p+1}} - 12(m+1-p)^{\underline{6-p}}/7(p-6)(n+1)^{\underline{7}},$$

for $n>m$; except that when $p=-1$, $x_n/(n+1) = \frac{12}{7}(H_{n+1} - H_{m+2}) + \frac{37}{49} + \frac{12}{49}(m+2)^{\underline{7}}/(n+1)^{\underline{7}}$, and when $p=6$, $x_n/(n+1) = -\frac{12}{7}(H_{n-6} - H_{m-5})/(n+1)^{\underline{7}} + \frac{12}{49}/(m+2)^{\underline{7}} + \frac{37}{49}/(n+1)^{\underline{7}}$.

The above recurrence arises in the following ways in this problem, if we use $\lceil(l+r)/2\rceil$ in place of $\lfloor(l+r)/2\rfloor$ (cf. exercise 55):

Quantity	Value for $n\le m$	$b_n/\binom{n}{3}$ for $n>m$	Solution for $n>m$.
A_n	0	1	$(n+1)\big(\frac{12}{7}/(m+2)\big) - 1$
B_n	0	$\frac{2n-3}{10} + \frac{3}{32}\frac{\pi(\pi-2)}{(\pi-1)(\pi-3)}$	$(n+1)\big(\frac{12}{35}(H_{n+1} - H_{m+2}) + \frac{37}{245} - \frac{24}{56}/(m+2) + \cdots\big)$
C_n	0	$n+1$	$(n+1)\big(\frac{12}{7}(H_{n+1} - H_{m+2}) + \frac{37}{49}\big)$
D_n	$n-1$	0	$(n+1)\big(1 - \frac{24}{7}/(m+2)\big)$
E_n	$n(n-1)/4$	0	$(n+1)\big(\frac{6}{35}m - \frac{17}{35} + \frac{6}{7}/(m+2)\big)$
L_n	δ_{n1}	0	$(n+1)\big(\frac{24}{7}/(m+2)(m+1)m\big)$
X_n	0	$\frac{1}{2} - \frac{3}{4}\frac{n+1-\pi}{(\pi-1)(\pi-3)}$	$(n+1)\big(\frac{6}{7}/(m+2) - \cdots\big) - \frac{1}{2}$

Here $\bar{n} = 2\lceil n/2 \rceil$, and the solutions involve further terms of $O(n^{-6})$. The total average running time of the program in exercise 55 is $52\frac{1}{6}A_N + 14B_N + 4C_N + 12D_N + 8E_N - L_N + 8X_N + 15$; $M = 9$ and $M = 10$ are about equally good. With DIV instead of SRB, add $13A_N$ to the running time and take $M = 10$.

57. The asymptotic series for

$$\sum_{n>N} n^{-1}(1 - \alpha/N)^{n-N} = -N^{-1} + \sum_{k\geq 0} (N+k)^{-1}(1 - \alpha/N)^k$$

can be obtained by restricting k to $O(N^{1+\epsilon})$, expanding $(1 - \alpha/N)^k$ as $e^{-\alpha k/N}$ times $(1 - k\alpha^2/2N^2 + \cdots)$, and using Euler's summation formula; it comes to $e^\alpha E_1(\alpha)(1 + \alpha^2/2N) - (1+\alpha)/2N + O(N^{-2})$. Hence the asymptotic value of 5.2.1–11 is $N(\ln\alpha + \gamma + E_1(\alpha))/\alpha + (1 - e^{-\alpha}(1+\alpha))/2\alpha + O(N^{-1})$. [The coefficient of N is 0.7966, 0.6596, 0.2880, respectively, for $\alpha = 1, 2, 10$.] Note that $\ln\alpha + \gamma + E_1(\alpha) = \int_0^\alpha (1-e^{-t})t^{-1}dt$.

SECTION 5.2.3

1. No; but the method using ∞ (described just before Algorithm S) is stable.

2. Traversing a linear list stored sequentially in memory is often slightly faster if we scan the list from higher indices to lower, since it is usually easier to test if an index is zero than to test if it exceeds N. (For the same reason, the search in step S2 runs from j down to 1; but see exercise 8!)

3. (a) The permutation $a_1 \ldots a_{N-1}\ N$ occurs for inputs $N\ a_2 \ldots a_{N-1}\ a_1$, $a_1\ N\ a_3 \ldots a_{N-1}\ a_2$, $\ldots$, $a_1\ a_2 \ldots a_{N-2}\ N\ a_{N-1}$, $a_1 \ldots a_{N-1}\ N$. (b) The average number of times the maximum is changed during the first iteration of step S2 is $H_N - 1$, as shown in Section 1.2.10. [Hence B_N can be found from Eq. 1.2.7–8.]

4. If the input is a permutation of $\{1, 2, \ldots, N\}$, the number of times $i = j$ in step S3 is exactly one less than the number of cycles in the permutation. (Indeed, it is not hard to show that steps S2 and S3 simply remove element j from its cycle; hence S3 is inactive only when j was the smallest element in its cycle.) By Eq. 1.3.3–21 we could save $H_N - 1$ of the $N - 1$ executions of step S3, on the average.

Thus it is inefficient to insert an extra test "$i = j$?" before step S3. Instead of testing i vs. j, however, we could lengthen the program for S2 slightly, duplicating part of the code, so that S3 never is encountered if the initial guess K_j is not changed during the search for the maximum; this would make Program S a wee bit faster.

5. $(N-1) + (N-3) + \cdots = \lfloor N^2/4 \rfloor$.

6. (a) If $i \neq j$ in step S3, that step decreases the number of inversions by $2m - 1$, where m is one more than the number of keys in $K_{i+1} \ldots K_{j-1}$ lying between K_i and K_j; clearly m is not less than the contribution to B on the previous step S2. Now apply the observation of exercise 4, connecting cycles to the condition $i = j$. (b) Every permutation can be obtained from $N \ldots 2\ 1$ by successive interchanges of adjacent elements that are out of order. (Apply, in reverse sequence, the interchanges that sort the permutation into decreasing order.) Each such operation decreases I by one and changes C by ± 1. Hence no permutation has a value of $I - C$ which exceeds the corresponding value for $N \ldots 2\ 1$. [By exercise 5 the inequality $B \leq \lfloor N^2/4 \rfloor$ is best possible.]

7. [When $N = 5$ the generating function is $\frac{1}{120}(1 + 4z + 15z^2 + 30z^3 + 37z^4 + 26z^5 + 7z^6)$.]

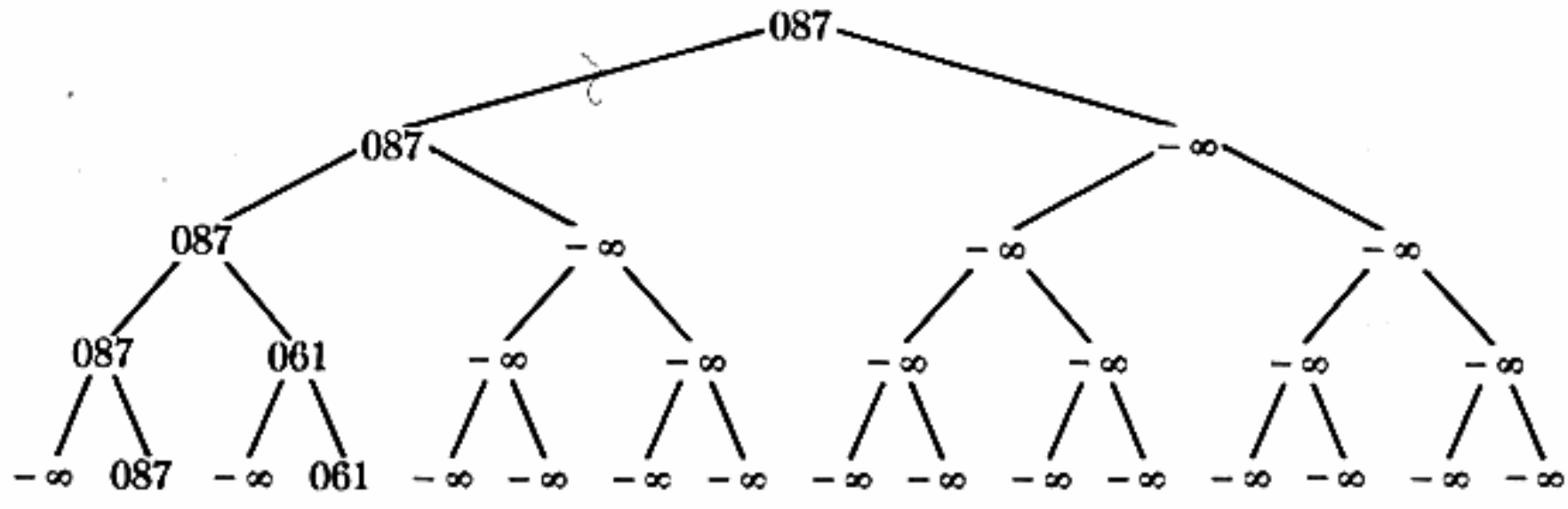
8. We can start the next iteration of step S2 at position K_i, provided that we have remembered max $(K_1, \ldots, K_{i-1})$. One way to keep all of this auxiliary information is to use a link table $L_1 \ldots L_N$ such that K_{L_k} is the previous boldface element whenever K_k is boldface; $L_1 = 0$. [We could also use less auxiliary storage, at the expense of some redundant comparisons.]

The following MIX program uses address modification so that the inner loop is fast. rI1 $\equiv j$, rI2 $\equiv k - j$, rI3 $\equiv i$, rA $\equiv K_i$.

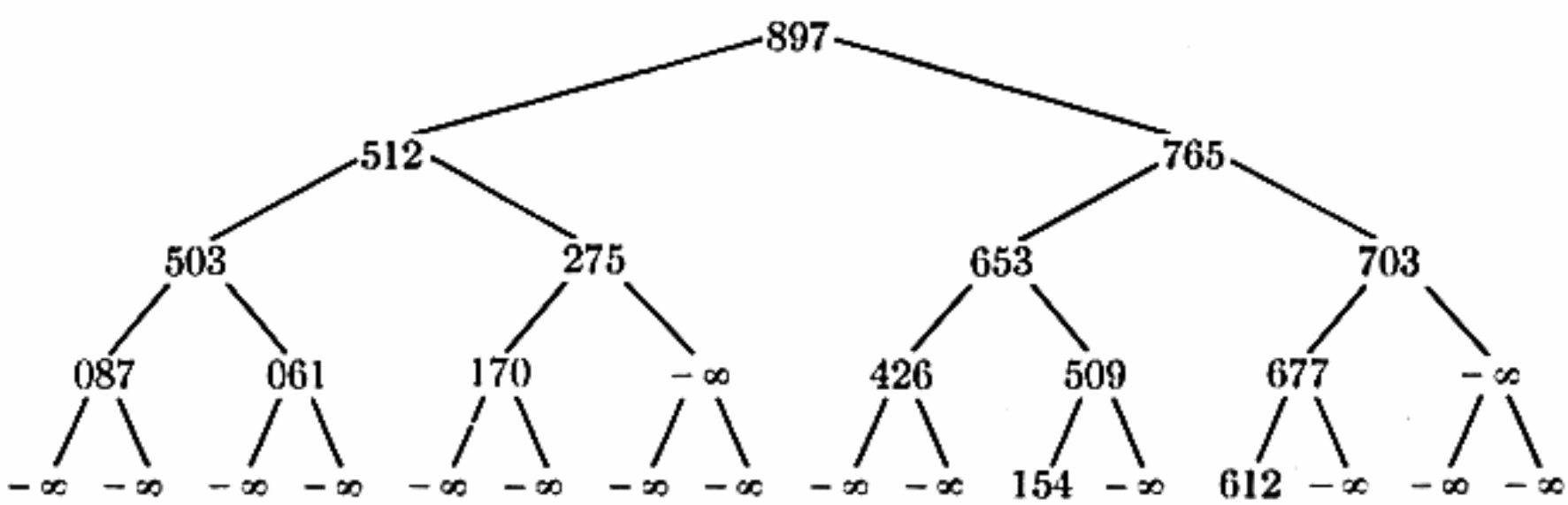
```
01  START  ENT1  N              1          j ← N.
02         STZ   LINK+1         1
03         JMP   9F             1
04  1H     ST1   6F(0:2)        N - D      Modify addresses in loop.
05         ENT4  INPUT,1        N - D
06         ST4   7F(0:2)        N - D
07         ENT4  LINK,1         N - D
08         ST4   8F(0:2)        N - D
09  7H     CMPA  INPUT+J,2      A                          [Address modified]
10         JGE   *+4            A          Jump if K_i ≥ K_k.
11  8H     ST3   LINK+J,2       N + 1 - C  Otherwise L_k ← i, [Address modified]
12  6H     ENT3  J,2            N + 1 - C  i ← k.          [Address modified]
13         LDA   INPUT,3        N + 1 - C
14         INC2  1              A          k ← k + 1
15         J2NP  7B             A          Jump if k ≤ j.
16  4H     LDX   INPUT,1        N
17         STX   INPUT,3        N          R_i ← R_j.
18         STA   INPUT,1        N          R_j ← former R_i.
19         DEC1  1              N          j ← j - 1.
20         ENT2  0,3            N          rI2 ← i.
21         LD3   LINK,3         N          i ← L_i.
22         J3NZ  5F             N          If i > 0, k will start at i.
23  9H     ENT3  1              C          Otherwise i ← 1;
24         ENT2  2              C          k will start at 2.
25  5H     DEC2  0,1            N + 1
26         LDA   INPUT,3        N + 1      rA ← K_i.
27         J2NP  1B             N + 1      Jump if k ≤ j.
28         J1P   4B             D + 1      Jump if j > 0.  ∎
```

9. $N - 1 + \sum_{N \ge k \ge 2} ((k-1)/2 - 1/k) = \frac{1}{2}\binom{N}{2} + N - H_N$. [The average values of C and D are, respectively, $H_N + 1$ and $H_N - \frac{1}{2}$; hence the average running time of the program is $(1.25N^2 + 31.75N - 15H_N + 14.5)u$.] Program H is much better.

10.



11.



12. $2^n - 1$, once for each "$-\infty$" in a branch node.

13. If $K \geq K_{r+1}$, then step H4 may go to step H5 if $j = r$. (Step H5 is inactive unless $K_r < K_{r+1}$, when step H6 will go to H8 anyway.) To ensure that $K \geq K_{r+1}$ throughout the algorithm, we may start with $K_{N+1} \leq \min(K_1, \ldots, K_N)$; instead of setting $R_r \leftarrow R_1$ in step H2, set $R_{r+1} \leftarrow R_{N+1}$ and $R_{N+1} \leftarrow R_1$; also set $R_2 \leftarrow R_{N+1}$ after $r = 1$. (This trick does not speed up the algorithm nor does it make Program H any shorter.)

14. When inserting an element, give it a key which is less (or greater) than all previously assigned keys, to get the effect of a simple queue (or stack, respectively).

15. For efficiency, the following solution is a little bit tricky, avoiding all multiples of 3 [*CACM* **10** (1967), 570].

a) Set $p[1] \leftarrow 2$, $p[2] \leftarrow 3$, $k \leftarrow 2$, $n \leftarrow 5$, $d \leftarrow 2$, $r \leftarrow 1$, $t \leftarrow 25$, and place (25, 10, 30) in the priority queue. (In this algorithm, $p[i] =$ ith prime; $k =$ number of primes found so far; $n =$ prime candidate; $d =$ distance to next candidate; $r =$ number of elements in the queue; $t = p[r+2]^2$, the next n for which we should increase r. The queue entries have the form $(u, v, 6p)$, where p is the smallest prime divisor of u, $v = 2p$ or $4p$, and $u + v$ is not a multiple of 3.)

b) Let (q, q', q'') be the queue element having the smallest first component. If $n \neq q$, set $k \leftarrow k+1$, $p[k] \leftarrow n$, $n \leftarrow n + d$, $d \leftarrow 6 - d$, and repeat this step until either $n > N$ or $n = q$.

c) If $n = t$, set $r \leftarrow r+1$, $u \leftarrow p[r+2]$, $t \leftarrow u^2$ and insert $(t, 2u, 6u)$ or $(t, 4u, 6u)$ into the queue according as $u \bmod 3 = 2$ or $u \bmod 3 = 1$.

d) Replace (q, q', q'') in the queue by $(q + q', q'' - q', q'')$. Set $n \leftarrow n + d$, $d \leftarrow 6 - d$, and return to (b).

Thus the computation begins as follows:

Queue contents	Primes found
(25, 10, 30)	5, 7, 11, 13, 17, 19, 23
(35, 20, 30)(49, 28, 42)	29, 31
(49, 28, 42)(55, 10, 30)	37, 41, 43, 47
(55, 10, 30)(77, 14, 42)(121, 22, 66)	53

Note that the queue entries always have distinct first components. If the queue is maintained as a heap, we can find all primes $\leq N$ in $O(N \log N)$ steps; the length of the heap is at most the number of primes $\leq \sqrt{N}$. The sieve of Eratosthenes (exercise 4.5.4–8) is a $O(N)$ method requiring considerably more random-access storage.

16. *Step 1.* Set $K \leftarrow$ key to be inserted; $j \leftarrow n + 1$.
Step 2. Set $i \leftarrow \lfloor j/2 \rfloor$.
Step 3. If $i = 0$ or $K_i \geq K$, set $K_j \leftarrow K$ and terminate the algorithm.
Step 4. Set $K_j \leftarrow K_i$, $j \leftarrow i$, and return to step 2.

17. The file 1 2 3 goes into the heap 3 2 1 with Algorithm H, but into 3 1 2 with exercise 16. (*Note:* The latter method of heap creation has a worst case of order $N \log N$; but empirical tests have shown that the number of iterations of step 2 during the creation of a heap is only about $2.27N - 32$, for random input.)

18. (For convenience we assume the presence of an artificial key $K_0 \geq \max(K_1, \ldots, K_N)$.) Delete step H6, and replace H8 by:

H8′. [Move back up.] Set $j \leftarrow i$, $i \leftarrow \lfloor j/2 \rfloor$.

H9′. [Does K fit?] If $K \leq K_i$, set $K_j \leftarrow K$ and return to H2. Otherwise set $K_j \leftarrow K_i$ and return to H8′. ∎

The method is essentially the same as in exercise 16, but with a different starting place in the heap. The net change to the file is the same as in Algorithm H. Empirical tests on this method show that the number of times $K_j \leftarrow K_i$ occurs per siftup during the selection phase is (0, 1, 2) with respective probabilities (.848, .135, .016). This method makes Program H somewhat longer but improves its asymptotic speed to $13N \log_2 N + O(N)$. An instruction to halve the value of an index register would be desirable.

19. Proceed as in the revised sift-up algorithm of exercise 18, with $K = K_N$, $l = 1$, $r = N - 1$, starting with a given value of j in step H3.

20. For $0 \leq k \leq n$, the number of positive integers $\leq N$ whose binary representation has the form $(b_n \ldots b_k a_1 \ldots a_q)_2$ for some $q \geq 0$ is clearly $\sum_{0 \leq q < k} 2^q + (b_{k-1} \ldots b_0)_2 + 1 = (1 b_{k-1} \ldots b_0)_2$.

21. Let $j = (c_r \ldots c_0)_2$ be in the range $\lfloor N/2^{k+1} \rfloor = (b_n \ldots b_{k+1})_2 < j < (b_n \ldots b_k)_2 = \lfloor N/2^k \rfloor$. Then s_j is the number of positive integers $\leq N$ whose binary representation has the form $(c_r \ldots c_0 a_1 \ldots a_q)_2$ for some $q \geq 0$, namely $\sum_{0 \leq q < k} 2^q = 2^{k+1} - 1$. Hence the number of nonspecial subtrees of size $2^{k+1} - 1$ is

$$\lfloor N/2^k \rfloor - \lfloor N/2^{k+1} \rfloor - 1 = \lfloor (N - 2^k)/2^{k+1} \rfloor.$$

[For the latter identity, use the replicative law in exercise 1.2.4–38 with $n = 2$ and $x = N/2^{k+1}$.]

22. Before $l = 1$ the five possibilities are 5 3 4 1 2, 3 5 4 1 2, 4 3 5 1 2, 1 5 4 3 2, 2 5 4 1 3. Each of these possibilities $a_1 a_2 a_3 a_4 a_5$ leads to three possible permutations $a_1 a_2 a_3 a_4 a_5$, $a_1 a_4 a_3 a_2 a_5$, $a_1 a_5 a_3 a_4 a_2$ before $l = 2$.

23. (a) After B iterations, $j \geq 2^B l$; hence $2^B l \leq r$. (b) $\sum_{1 \leq l \leq N} \lfloor \log_2 (N/l) \rfloor = (\lfloor N/2 \rfloor - \lfloor N/4 \rfloor) + 2(\lfloor N/4 \rfloor - \lfloor N/8 \rfloor) + 3(\lfloor N/8 \rfloor - \lfloor N/16 \rfloor) + \cdots = \lfloor N/2 \rfloor + \lfloor N/4 \rfloor + \lfloor N/8 \rfloor + \cdots = N - \nu(N)$, where $\nu(N)$ is the number of ones in the binary representation of N. Also by exercise 1.2.4–42 we have $\sum_{1 \leq r < N} \lfloor \log_2 r \rfloor = N \lfloor \log_2 N \rfloor - 2^{\lfloor \log_2 N \rfloor + 1} + 2$. We know by Theorem H that this upper bound on B is best possible during the heap creation phase. Furthermore it is interesting to note that there is a unique heap containing the keys $\{1, 2, \ldots, N\}$ such that K is identically equal to 1 throughout the selection phase of Algorithm H. (For example, when $N = 7$ that heap is 7 5 6 2 4 3 1; it is not difficult to pass from N to $N + 1$.) This heap gives the maximum value of B (as well as the maximum value $\lfloor N/2 \rfloor$ of D) for the selection phase of heapsort, so the best possible upper bound on B for the entire sort is $N - \nu(N) + N \lfloor \log_2 N \rfloor - 2^{\lfloor \log_2 N \rfloor + 1} + 2$.

24. $\sum_{1 \leq k \leq N} \lfloor \log_2 k \rfloor^2 = (N + 1 - 2^n) n^2 + \sum_{0 \leq k < n} k^2 2^k = (N + 1) n^2 - 6 -$

$(2n-3)2^{n+1}$, where $n = \lfloor \log_2 N \rfloor$ (cf. exercise 4.5.2–23); hence the variance of the last sift-up is $\beta_N = \big((N+1)n^2 - (2n-3)2^{n+1} - 6\big)/N - \big((N+1)n + 2 - 2^{n+1}\big)^2/N^2 = O(1)$. The standard deviation of B'_N is $(\sum_{s \in M_N} \beta_s)^{1/2} = O(\sqrt{N})$.

25. The sift-up is "uniform," and each comparison $K_j : K_{j+1}$ has probability $\frac{1}{2}$ of coming out $<$. The average contribution to C in this case is just one-half the sum of the average contributions to A and B, namely $\big((2n-3)2^{n-1} + \frac{1}{2}\big)/(2^{n+1}-1)$.

26. (a) $(\frac{10}{25} + \frac{1}{2} + 1\frac{3}{9} + \frac{1}{2} + 1\frac{1}{2} + 1\frac{2}{5} + 2\frac{1}{2} + \frac{1}{2} + 1\frac{1}{2} + 1\frac{1}{2} + 2\frac{1}{2} + 1\frac{1}{2} + 2 + 2 + 3 + 0 + 1 + 1 + 2 + 1 + 2 + 2 + 3 + 1 + 2 + 2)/26$. (b) $\big(\sum_{1 \le k \le N} \nu(k) - N + \frac{1}{2}\lfloor N/2 \rfloor - \frac{1}{2}n + \sum_{1 \le k < n} \min(\alpha_{k-1}, \alpha_k - \alpha_{k-1} - 1)/(\alpha_k - 1)\big)/N$, where $\nu(k)$ is the number of one bits in the binary representation of k, and $\alpha_k = (1b_k \ldots b_0)_2$. If $N = 2^{e_1} + 2^{e_2} + \cdots + 2^{e_t}$, with $e_1 > e_2 > \cdots > e_t \ge 0$, it can be shown that $\sum_{0 \le k \le N} \nu(k) = \frac{1}{2}\big((e_1+2)2^{e_1} + (e_2+4)2^{e_2} + \cdots + (e_t+2t)2^{e_t}\big) + t - N$.

27. In general, the Lambert series $\sum_{n \ge 1} a_n x^n/(1-x^n) = \sum_{N \ge 1} (\sum_{d \backslash n} a_d) x^N = \sum_{m \ge 1} x^{m^2}(a_m + \sum_{k \ge 1} (a_m + a_{m+k}) x^{km})$.

[Note that when $a_n = n$ and $x = \frac{1}{2}$ we obtain the relation

$$\beta = \sum_{n \ge 1} \frac{1}{2^n - 1} = \sum_{n \ge 1} 2^{-n^2} \left(n \left(\frac{2^n + 1}{2^n - 1} \right) + \frac{2^n}{(2^n-1)^2} \right) = 2.74403\ 38887\ 59488\ldots;$$

this constant arises in (20), where we have $B'_N \sim (\beta - 2)N$ and $C'_N \sim (\frac{1}{2}\beta - \frac{1}{4}\alpha - \frac{1}{2})N$.]

28. The sons of node k are nodes $3k-1$, $3k$, and $3k+1$. A MIX program analogous to Program H will take asymptotically $21\frac{2}{3} N \log_3 N \approx 13.7 N \log_2 N$ units of time. Using the idea of exercise 18 lowers this to $18\frac{2}{3} N \log_3 N \approx 11.8 N \log_2 N$, although the division by 3 will add a large $O(N)$ term.

31. Let A, B be arrays of n elements each such that $A[\lfloor i/2 \rfloor] \le A[i]$ and $B[i] \le B[\lfloor i/2 \rfloor]$ for $1 < i \le n$; furthermore we require that $A[i] \le B[i]$ for $1 \le i \le n$. (The latter condition holds for all i iff it holds for $n/2 < i \le n$, because of the heap structure.) This "twin-heap" contains $2n$ elements; to handle an odd number of elements, we simply keep one element off to the side. Appropriate modifications of the other algorithms in this section can be used to maintain twin heaps, and it is interesting to work out the details.

32. Let P, Q point to the given priority queues; for brevity, the following algorithm uses the convention DIST(Λ) = 0.

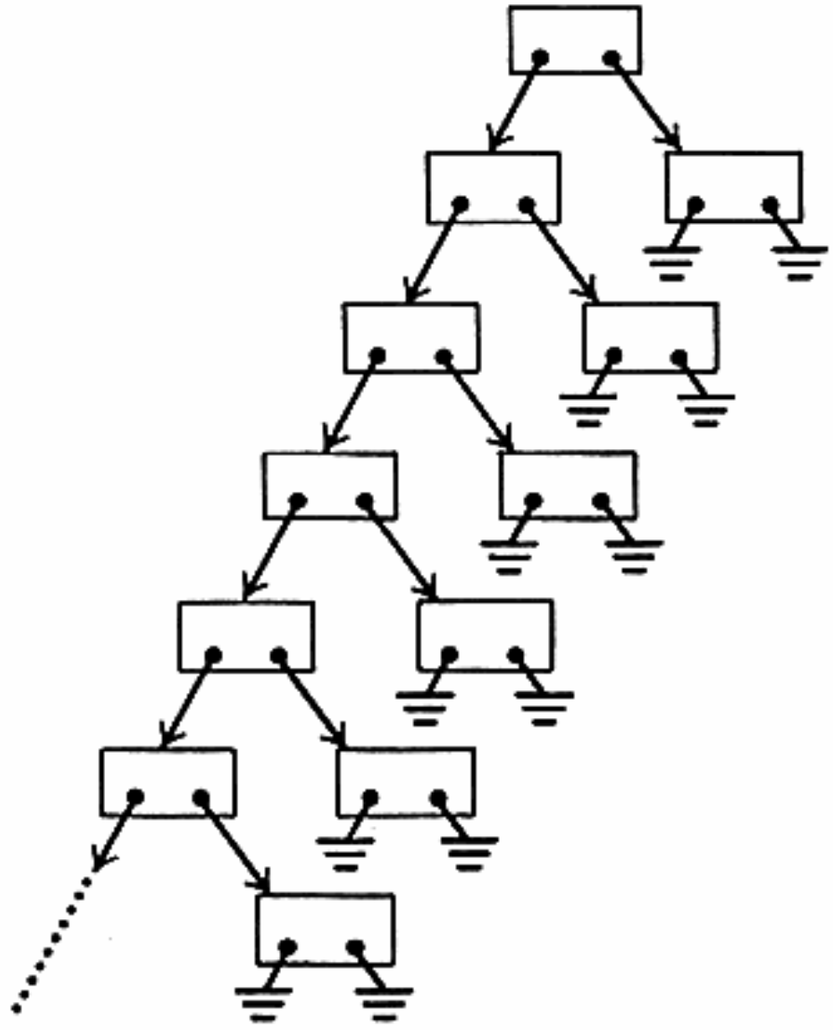
M1. [Initialize.] Set R ← Λ.

M2. [List merge.] If Q = Λ, set D ← DIST(P) and go to M3. If P = Λ, set P ← Q, D ← DIST(P), and go to M3. Otherwise if KEY(P) ≥ KEY(Q), set T ← RIGHT(P), RIGHT(P) ← R, R ← P, P ← T and repeat step M2. If KEY(P) > KEY(Q), set T ← RIGHT(Q), RIGHT(Q) ← R, R ← Q, Q ← T and repeat step M2. (This step essentially merges the two "right lists" of the given trees, temporarily inserting upward pointers into the RIGHT fields.)

M3. [Done?] If R = Λ, terminate the algorithm; P points to the answer.

M4. [Fix DISTs.] Set Q ← RIGHT(R). If DIST(LEFT(R)) < D, then set D ← DIST(LEFT(R)) + 1, RIGHT(R) ← LEFT(R), LEFT(R) ← P; otherwise set D ← D + 1, RIGHT(R) ← P. Finally set DIST(R) ← D, R ← Q, P ← R, and return to M3. ∎

33. We might have to promote about half the nodes in a lop-sided tree such as



34. Let the number be L_{N+1}. It is not difficult to prove that $L_{m+n} \leq L_m L_n$, hence (taking logarithms) $\lim L_n^{1/n}$ exists. (This suggestion is due to D. A. Klarner. Some calculations by E. Logg suggest that $L_{n+1}/L_n \geq L_n/L_{n-1}$ may hold for all n.)

35. Let the DIST field of the deleted node be d_0, and let the DIST field of the merged subtrees be d_1. If $d_0 = d_1$, we need not go up at all. If $d_0 > d_1$, and if we go up n levels, the new DIST fields of the ancestors of P must be, respectively, $d_1 + 1$, $d_1 + 2$, $\ldots$, $d_1 + n$. If $d_0 < d_1$, the upward path must go only leftwards.

36. Instead of a general priority queue, it is simplest to use a doubly-linked list; when "using" a node, move it to one end of the list, and delete nodes from the other end. [See the discussion of "self-organizing files" in Section 6.1.]

SECTION 5.2.4

1. Start with $i_1 = \cdots = i_k = 1$, $j = 1$. Repeatedly find $\min(x_{1i_1}, \ldots, x_{ki_k}) = x_{ri_r}$ and set $z_j \leftarrow x_{ri_r}$, $j \leftarrow j + 1$, $i_r \leftarrow i_r + 1$. (In this case the use of $x_{i(m_i+1)} = \infty$ is a decided convenience.)

When k is moderately large, it is desirable to keep the keys $x_{1i_1}, \ldots, x_{ki_k}$ in a tree structure suited to repeated selection, as discussed in Section 5.2.3, so that only $\lfloor \log_2 k \rfloor$ comparisons are needed to find the new minimum each time after the first. Indeed this is a typical application of the principle of "smallest in, first out" in a priority queue. The keys can be maintained as a heap, and ∞ can be avoided entirely. See the further discussion in Section 5.4.1.

2. Let C be the number of comparisons; we have $C = m + n - S$, where S is the number of elements transmitted in step M4 or M6. The probability that $S \geq s$ is easily seen to be

$$q_s = \left(\binom{m+n-s}{m} + \binom{m+n-s}{n} \right) \Big/ \binom{m+n}{m}$$

for $1 \le s \le m+n$; $q_s = 0$ for $s > m+n$. Hence the mean of S is $\mu_{mn} = q_1 + q_2 + \cdots = m/(n+1) + n/(m+1)$ [cf. exercises 3.4.2–5, 6], and the variance is $\sigma^2_{mn} = (q_1 + 3q_2 + 5q_3 + \cdots) - \mu^2_{mn} = m(2m+n)/(n+1)(n+2) + (m+2n)n/(m+1)(m+2) - \mu^2_{mn}$. Thus

$$C = (\min\ \min(m, n), \quad \text{ave}\ m+n-\mu_{mn}, \quad \max\ m+n-1, \quad \text{dev}\ \sigma_{mn}).$$

When $m = n$ the average was first computed by H. Nagler, *CACM* **3**(1960), 618–620; it is asymptotically $2n - 2 + O(n^{-1})$, with a standard deviation of $\sqrt{2} + O(n^{-1})$. Thus C hovers close to its maximum value.

3. M2′. If $K_i < K'_j$, go to M3′; if $K_i = K'_j$, go to M7′; if $K_i > K'_j$, go to M5′.

M7′. Set $K''_k \leftarrow K'_j$, $k \leftarrow k+1$, $i \leftarrow i+1$, $j \leftarrow j+1$. If $i > M$, go to M4′; otherwise if $j > N$, go to M6′; otherwise return to M2′. ▮

(Appropriate modifications are made to other steps of Algorithm M. Again many special cases disappear if we insert artificial keys $K_{M+1} = K'_{N+1} = \infty$ at the end of the files.)

4. The sequence of elements which appears at a fixed internal node of the selection tree, as time passes, is obtained by merging the sequences of elements which appear at the sons of that node. (The discussion in Section 5.2.3 is based on selecting the *largest* element, but it could equally well have reversed the order.) So the operations involved in tree selection are essentially the same as those involved in merging, but they are performed in a different sequence and using different data structures.

Another relation between merging and tree selection is indicated in exercise 1. Note that an N-way merge of one-element files is a selection sort; compare also four-way merging of (A, B, C, D) to two-way merging of (A, B), (C, D), then (AB, CD).

5. In step N6 we always have $K_i < K_{i-1} \le K_j$; in N10, $K_j < K_{j+1} < K_i$.

6. 2 6 4 10 8 14 12 16 15 11 13 7 9 3 5 1. After one pass we have 1 2 5 6 7 8 13 14 16 15 12 11 10 9 4 3 (two of the expected stepdowns disappear). This possibility was noted by D. A. Bell, *Comp. J.* **1** (1958), 74. Quirks like this make it almost hopeless to carry out a precise analysis of Algorithm N.

7. $\lceil \log_2 N \rceil$, if $N > 1$. (Consider how many times p must be doubled until it is $\ge N$.)

8. If N is not a multiple of $2p$, there is one short run on the pass, and it is always near the middle; letting its length be t, we have $0 \le t < p$, and the situation is essentially $x_1 \le x_2 \le \cdots \le x_p \mid y_t \ge \cdots \ge y_1$. If $x_p \le y_t$, the left-hand run is exhausted first, and step S6 will take us to S13 after x_p has been transmitted. On the other hand, if $t = 0$ or $x_p > y_t$, the right-hand side will be artificially exhausted, but $K_j = x_p$ will never be $< K_i$ in step S3! Thus S6 will eventually take us to S13 in all cases.

10. For example, Algorithm M can merge elements $x_{j+1} \ldots x_{j+m}$ with $x_{j+m+1} \ldots x_{j+m+n}$ into positions $x_1 \ldots x_{m+n}$ of an array without conflict, if $j \ge n$. With care we can exploit this idea so that $N + 2^{\lfloor \log_2 N \rfloor - 1}$ locations are required for an entire sort. But the program seems to be rather complicated compared to Algorithm S. [*Comp. J.* **1** (1958), 75; see also L. S. Lozinskiĭ, *Kibernetika* **1**, 3 (1965), 58–62.]

11. Yes. This can be seen, for example, by considering the relation to tree selection mentioned in exercise 4. But Algorithms N and S are obviously not stable.

12. Set $L_0 \leftarrow 1$, $t \leftarrow N+1$; then for $p = 1, 2, \ldots, N-1$, do the following:

If $K_p \le K_{p+1}$ set $L_p \leftarrow p+1$; otherwise set $L_t \leftarrow -(p+1)$, $t \leftarrow p$.

Finally set $L_t \leftarrow 0$, $L_N \leftarrow 0$, $L_{N+1} \leftarrow |L_{N+1}|$.

(Stability is preserved. The number of passes is $\lceil \log_2 r \rceil$, where r is the number of ascending runs in the input; the exact distribution of r is analyzed in Section 5.1.3. We may conclude that "natural" merging is preferable to "straight" merging when linked allocation is being used, although it was inferior for sequential allocation.)

13. The running time for $N \ge 3$ is $(11A + 6B + 3B' + 9C + 2C'' + 4D + 5N + 9)u$, where A is the number of passes; $B = B' + B''$ is the number of subfile-merge operations performed, where B' is the number of such merges in which the p subfile was exhausted first; $C = C' + C''$ is the number of comparisons performed, where C' is the number of such comparisons with $K_p \le K_q$; $D = D' + D''$ is the number of elements remaining in subfiles when the other subfile has been exhausted, where D' is the number of such elements belonging to the q subfile.

Algorithm L does a sequence of merges on subfiles whose sizes (m, n) can be determined as follows: Let $N - 1 = (b_k \ldots b_1 b_0)_2$ in binary notation. There are $(b_k \ldots b_{j+1})_2$ "ordinary" merges with $(m, n) = (2^j, 2^j)$, for $0 \le j < k$; and there are "special" merges with $(m, n) = (2^j, 1 + (b_{j-1} \ldots b_0)_2)$ whenever $b_j = 1$, for $0 \le j \le k$. For example when $N = 14$, there are six ordinary (1, 1) merges, three ordinary (2, 2) merges, one ordinary (4, 4) merge, and the special merges deal with subfiles of sizes (1, 1), (4, 2), (8, 6).

It follows that, regardless of the input distribution, we have $A = \lceil \log_2 N \rceil$, $B = N - 1$, $C' + D'' = \sum_{0 \le j \le k} b_j 2^j (1 + \frac{1}{2}j)$, $C'' + D' = \sum_{0 \le j \le k} b_j (1 + 2^j(\frac{1}{2}j + b_{j+1} + \cdots + b_k))$; hence only B', C', D' need to be analyzed further.

If the input to Algorithm L is random, each of the merging operations satisfies the conditions of exercise 2, and is independent of the behavior of the other merges; so the distribution of B', C', D' is the convolution of their individual distributions for each subfile merge. The average values for such a merge are $B' = n/(m+n)$, $C' = mn/(n+1)$, $D' = n/(m+1)$. Sum these over all relevant (m, n) to get the exact average values.

When $N = 2^k$ we have, of course, the simplest situation; $B'_{\text{ave}} = \frac{1}{2}B$, $C'_{\text{ave}} = \frac{1}{2}C_{\text{ave}}$, $C + D = kN$, and $D_{\text{ave}} = \sum_{1 \le j \le k} (2^{k-j} \cdot 2^j/(2^{j-1} + 1)) = \alpha' N + O(1)$, where $\alpha' = \sum_{n \ge 0} 1/(2^n + 1) = \alpha + \frac{1}{2} - 2\sum_{n \ge 1} 1/(4^n - 1) = 1.26449\ 97803\ 48444\ 20919\ 13197\ 47255\ 49848\ 25577$– can be evaluated to high precision as in exercise 5.2.3–27. This special case was first analyzed by A. Gleason [unpublished, 1956] and H. Nagler [*CACM* **3** (1960), 618–620].

In Table 3 we have $A = 4$, $B' = 6$, $B'' = 9$, $C' = 22$, $C'' = 22$, $D' = 10$, $D'' = 10$, total time $= 761u$. (The comparable Program 5.2.1L takes only $433u$, when improved as in exercise 5.2.1–33, so we can see that merging isn't especially efficient when N is small.)

14. Set $D = B$ in exercise 13 to maximize C.

15. Make extra copies of steps L3, L4, L6 for the cases that L_s is known to equal p or q. [A *further* improvement can also be made, removing the assignment $s \leftarrow p$ (or $s \leftarrow q$) from the inner loop, by simply renaming the registers! For example, change lines 20, 21 to "`LD3 INPUT,1(L)`" and continue with p in rI3, s in rI1 and L_s known

to equal p. With twelve copies of the inner loop, corresponding to the different permutations of (p, q, r) with respect to (rI1, rI2, rI3), and the different knowledge about L_s, we can cut the average running time to $8N \log_2 N + O(N)$.]

16. (The result will be slightly faster than Algorithm L, cf. exercise 5.2.3–28.)

17. Consider the new record as a subfile of length 1. Repeatedly merge the smallest two subfiles if they have the same length. (The resulting sorting algorithm is essentially the same as Algorithm L, but the subfiles are merged at different relative times.)

18. Yes, but it seems to be a complicated job. The simplest known method uses the following ingenious construction [*Dokladi Akad. Nauk SSSR* **186** (1969), 1256–1258]: Let n be $\approx \sqrt{N}$. Divide the file into $m+2$ "zones" $Z_1 \ldots Z_m Z_{m+1} Z_{m+2}$, where Z_{m+2} contains (N mod n) records while each other zone contains exactly n records. Interchange the records of Z_{m+1} with the zone containing R_M; the file now takes the form $Z_1 \ldots Z_m A$, where each of $Z_1 \ldots Z_m$ contains exactly n records in order and where A is an auxiliary area containing s records, for some s, where $n \le s < 2n$.

Find the zone with smallest leading element, and exchange that entire zone with Z_1. (This takes $O(m+n)$ operations.) Then find the zone with next smallest leading element, and exchange it with Z_2, etc. Finally in $O(m(m+n)) = O(N)$ operations we will have rearranged the m zones so that their leading elements are in order. Furthermore, because of our original assumptions about the file, each of the keys in $Z_1 \ldots Z_m$ now has less than n inversions.

We can merge Z_1 with Z_2, using the following trick: Interchange Z_1 with the first n elements A' of A; then merge Z_2 with A' in the usual way but exchanging elements with the elements of $Z_1 Z_2$ as they are output. For example, if $n = 3$ and $x_1 < y_1 < x_2 < y_2 < x_3 < y_3$, we have

	Zone 1	Zone 2	Auxiliary
Initial contents:	$x_1\ x_2\ x_3$	$y_1\ y_2\ y_3$	$a_1\ a_2\ a_3$
Exchange Z_1:	$a_1\ a_2\ a_3$	$y_1\ y_2\ y_3$	$x_1\ x_2\ x_3$
Exchange x_1:	$x_1\ a_2\ a_3$	$y_1\ y_2\ y_3$	$a_1\ x_2\ x_3$
Exchange y_1:	$x_1\ y_1\ a_3$	$a_2\ y_2\ y_3$	$a_1\ x_2\ x_3$
Exchange x_2:	$x_1\ y_1\ x_2$	$a_2\ y_2\ y_3$	$a_1\ a_3\ x_3$
Exchange y_2:	$x_1\ y_1\ x_2$	$y_2\ a_2\ y_3$	$a_1\ a_3\ x_3$
Exchange x_3:	$x_1\ y_1\ x_2$	$y_2\ x_3\ y_3$	$a_1\ a_3\ a_2$

(The merge is always complete when the nth element of the auxiliary area has been exchanged; this method generally permutes the auxiliary records.)

The above trick is used to merge Z_1 with Z_2, then Z_2 with Z_3, ..., Z_{m-1} with Z_m, requiring a total of $O(mn) = O(N)$ operations. Since no element has more than n inversions, the $Z_1 \ldots Z_m$ portion of the file has been completely sorted.

For the final "cleanup," we sort $R_{N+1-2s} \ldots R_N$ by insertion, in $O(s^2) = O(N)$ steps; this brings the s largest elements into area A. Then we merge $R_1 \ldots R_{N-2s}$ with $R_{N+1-2s} \ldots R_{N-s}$, using the above trick with auxiliary storage area A (but interchanging the roles of right and left, less and greater, throughout). Finally, we sort $R_{N+1-s} \ldots R_N$ by insertion.

19. We may number the input cars so that the final permutation has them in order, 1 2 . . . 2^n; so this is essentially a sorting problem. First move the first 2^{n-1} cars through $n - 1$ stacks, putting them in decreasing order, and transfer them to the nth stack so that the smallest is on top. Then move the other 2^{n-1} cars through $n - 1$ stacks, putting them into increasing order and leaving them positioned just before the nth stack. Finally, merge the two sequences together in the obvious way.

20. For further information, see R. E. Tarjan, *JACM* **19** (1972), 341–346.

SECTION 5.2.5

1. No, because radix sorting doesn't work at all unless the distribution sorting is stable, after the first pass. (But the suggested distribution sort *could* be used in a most-significant-digit-first radix sorting method, generalizing radix exchange, as suggested in the last paragraph of the text.)

2. It is "anti-stable," just the opposite; elements with equal keys appear in reverse order, since the first pass goes through the records from R_N to R_1. (This proves to be convenient because of lines 28 and 20 of Program R, equating Λ with 0, but of course it is not necessary to make the first pass go backwards.)

3. If pile 0 is not empty, `BOTM[0]` already points to the first element; if it is empty, we set `P` ← `LOC(BOTM[0])` and later make `LINK(P)` point to the bottom of the first nonempty pile.

4. When there are an even number of passes remaining, take pile 0 first (top to bottom), followed by pile 1, . . . , pile $(M - 1)$; the result will be in order with respect to the digits examined so far. When there are an odd number of passes remaining, take pile $(M - 1)$ first, then pile $(M - 2)$, . . . , pile 0; the result will be in *reverse* order with respect to the digits examined so far. (This rule was apparently first published by E. H. Friend [*JACM* **3** (1956), 156, 165–166]. See also an article by Donald W. Johnson [*Library Resources and Technical Services* **3** (1959), 300–310], who proposed radix sorting as a manual aid for putting library cards in order. Johnson says the University of California library at Berkeley found that radix sorting made it possible to alphabetize 2000 cards by hand in only 4.8 hours instead of 8.5 hours by previous methods.)

5. (a) There are k empty piles after the $(N + 1)$st element is placed iff (i) the $(N + 1)$st element falls into a previously empty pile, with probability $((k + 1)/M) \times p_{MN(k+1)}$; or (ii) it falls into a nonempty pile, with probability $((M - k)/M)p_{MNk}$. The recurrence

$$p_{M,N+1,k} = \frac{k+1}{M} p_{M,N,k+1} + \frac{M-k}{M} p_{M,N,k}$$

is equivalent to the stated formula. (b) The nth derivative satisfies $g^{(n)}_{M,N+1}(z) = (1 - n/M)g^{(n)}_{M,N}(z) + ((1 - z)/M)g^{(n+1)}_{M,n}(z)$, by induction on n. Setting $z = 1$, we find $g^{(n)}_{M,N}(1) = (1 - n/M)^N M^{\underline{n}}$, since $g_{M0}(z) = z^M$. Hence mean $(g_{MN}) = (1 - 1/M)^N M$, var $(g_{MN}) = (1 - 2/M)^N M(M - 1) + (1 - 1/M)^N M - (1 - 1/M)^{2N} M^2$. (Note that the generating function for E in Program R is $(g_{MN}(z))^p$.)

6. Change line 04 to "`ENT3 7`", and change the `R3SW` and `R5SW` tables to:

```
R3SW    LD2  KEY,1(1:1)
        LD2  KEY,1(2:2)
        LD2  KEY,1(3:3)
        LD2  KEY,1(4:4)
        LD2  KEY,1(5:5)
        LD2  INPUT,1(1:1)
        LD2  INPUT,1(2:2)
        LD2  INPUT,1(3:3)
R5SW    LD1  INPUT,1(LINK)
        ...  (repeat previous line six more times)
        DEC1 1
```

The new running time is found by changing "3" to "8" everywhere; it amounts to $(11p - 1)N + 16pM + 12p - 4E + 2$, for $p = 8$.

7. Let R = radix sort, RX = radix exchange. Some of the important similarities and differences: RX goes from most significant digit to least significant, while R goes the other way. Both methods sort by digit inspections, without making comparisons of keys. RX always has $M = 2$ (but see exercise 1). The running time for R is almost unvarying, while RX is sensitive to the distribution of the digits. In both cases the running time is $O(N \log K)$, where K is the range of keys, but the constant of proportionality is higher for RX; on the other hand, when the keys are uniformly distributed in their leading digits, RX has an average running time of $O(N \log N)$ regardless of the size of K. R requires link fields while RX runs in "minimal storage." The inner loop of R is more suited to "pipeline" computers.

8. On the final pass, the piles should be hooked together in another order; for example, if $M = 256$, pile $(10000000)_2$ comes first, then pile $(10000001)_2$, . . . , pile $(11111111)_2$, pile $(00000000)_2$, pile $(00000001)_2$, . . . , pile $(01111111)_2$. This change in hooking order can be done easily by modifying Algorithm H, or (in Table 1) by changing the storage allocation strategy, on the last pass.

9. We could first separate the negative keys from the positive keys, as in exercise 5.2.2–33; or we could change the keys to complement notation on the first pass. Alternatively, after the last pass we could separate the positive keys from the negative ones, reversing the order of the latter, although the method of exercise 5.2.2–33 no longer applies.

11. Without the first pass the method would still sort perfectly, because (by coincidence) 503 already precedes 509. Without the first two passes, the number of inversions would be $1 + 1 + 0 + 0 + 0 + 1 + 1 + 1 + 0 + 0 = 5$.

12. After exchanging R_k with $R[P]$ in step M4 (exercise 5.2–12), we can compare K_k to K_{k-1}. If K_k is less, we compare it to K_{k-2}, K_{k-3}, . . . , until finding $K_k \geq K_j$. Then set $(R_{j+1}, \ldots, R_{k-1}, R_k) \leftarrow (R_k, R_{j+1}, \ldots, R_{k-1})$, *without* changing the LINK fields. It is convenient to place an artificial key K_0, which is $\leq$ all other keys, at the left of the file.

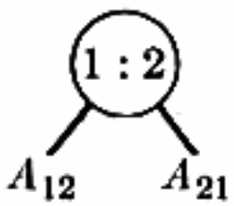
14. If the original permutation of the cards requires k readings, in the sense of exercise 5.1.3–20, and if we use m piles per pass, we must make at least $\lceil \log_m k \rceil$ passes. (Consider going back from a sorted deck to the original one; the number of readings increases

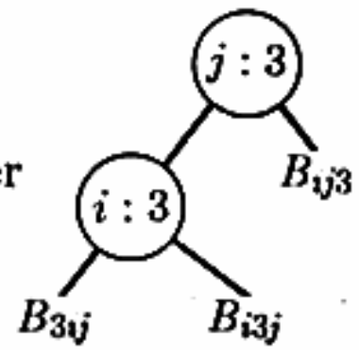
by at most a factor of m on each pass.) The given permutation requires 4 increasing readings, 10 decreasing readings, so decreasing order requires 4 passes with two piles or 3 passes with three piles.

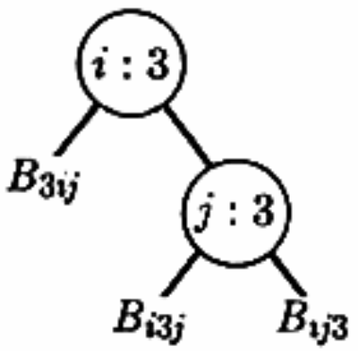
Conversely, this optimum number of passes can be achieved: Number the cards from 0 to $k - 1$ according to which reading it belongs to, and use a radix sort (least significant digit first in radix m). [Cf. *Martin Gardner's Sixth Book of Mathematical Games* (San Francisco: W. H. Freeman, 1971), 111–112.]

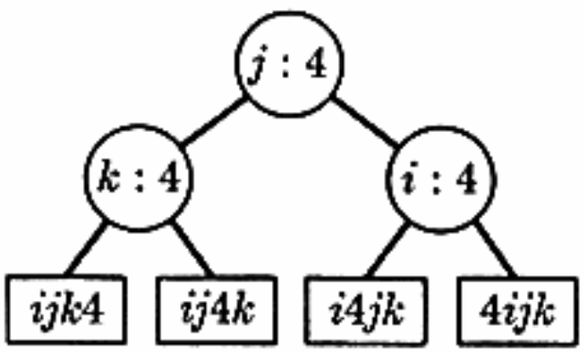
15. Let there be k readings and m piles. The order is reversed on each pass; if there are k readings in one order, the number of readings in the opposite order is $n + 1 - k$. The minimum number of passes is either the smallest even number greater than or equal to $\log_m k$ or the smallest odd number greater than or equal to $\log_m (n + 1 - k)$. (Going backwards, there are at most m decreasing readings after one pass, m^2 increasing readings after two passes, etc.) The example can be sorted into increasing order in $\min(2, 5) = 2$ passes, into decreasing order in $\min(3, 4) = 3$ passes, using only two piles.

SECTION 5.3.1

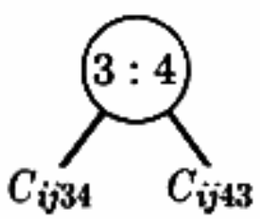
1. a) 

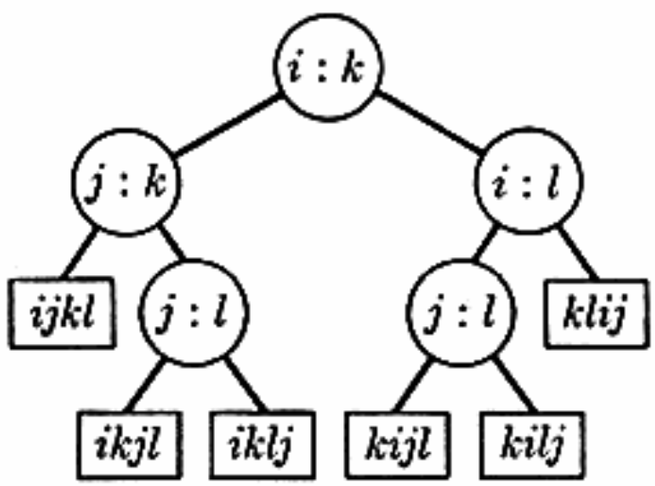
where A_{ij} = either 

or 

and B_{ijk} = 

External path length is 112 (optimum).

b) Here A_{ij} = 

where C_{ijkl} = 

External path length is 112 (optimum).

2. In the notation of exercise 5.2.4–14, $L(n) - B(n) = \sum_{1\le k\le t}\big((e_k+k-1)2^{e_k} - (e_1+1)2^{e_k}\big) + 2^{e_1+1} - 2^{e_t} = 2^{e_1} - 2^{e_t} - \sum_{2\le k\le t}(e_1 - e_k + 2 - k)2^{e_k} \ge 2^{e_1} -$

$(2^{e_1-1}+\cdots+2^{e_1-t+1}+2^{e_t})\geq 0$, with equality iff $n=2^k-2^j$ for some $k>j\geq 0$.

3. When $n>0$, the number of outcomes such that the smallest key appears exactly k times is $\binom{n}{k}P_{n-k}$. Thus $2P_n=\sum_k\binom{n}{k}P_{n-k}$, for $n>0$, and we have $2P(z)=e^zP(z)+1$. (Cf. Eq. 1.2.9–10.)

Another proof comes from the fact that $P_n=\sum_{k\geq 0}\left\{{n\atop k}\right\}k!$, since $\left\{{n\atop k}\right\}$ is the number of ways to partition n elements into k nonempty parts and these parts can be permuted in $k!$ ways. Thus $\sum_{n\geq 0}P_nz^n/n!=\sum_{k\geq 0}(e^z-1)^k=1/(2-e^z)$ by Eq. 1.2.9–23.

Still *another* proof, perhaps the most interesting, arises if we arrange the elements in sequence in a "stable" manner, so that K_i precedes K_j if and only if $K_i<K_j$ or $(K_i=K_j$ and $i<j)$. Among all P_n outcomes, a given arrangement $K_{a_1}\ldots K_{a_n}$ now occurs exactly 2^{k-1} times if the permutation $a_n\ldots a_1$ contains k runs; hence P_n can be expressed in terms of the Eulerian numbers, $P_n=\sum_k\left\langle{n\atop k}\right\rangle 2^{k-1}$. Eq. 5.1.3–20 with $z=2$ now establishes the desired result.

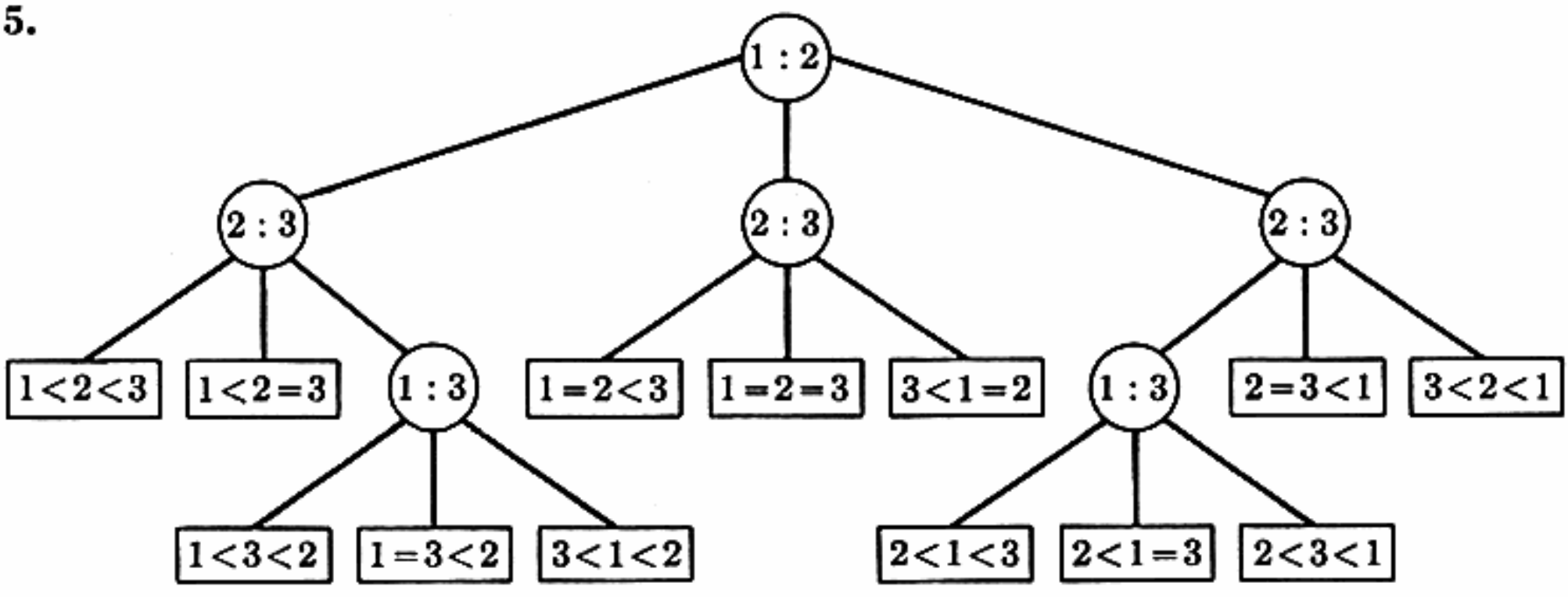
This generating function was obtained by A. Cayley [*Phil. Mag.* **18** (1859), 374–378] in connection with the enumeration of an imprecisely defined class of trees. See also J. Touchard, *Ann. Soc. Sci. Bruxelles* **53** (1933), 21–31. A table of $P_1,\ldots,P_{14}$ appears in O. A. Gross, *AMM* **69** (1962), 4–8; Gross gives the interesting formula $P_n=\sum_{k\geq 1}k^n/2^{1+k}$, $n\geq 1$.

4. The representation

$$2P(z)=\tfrac{1}{2}(1-i\cot(i(z-\ln 2)/2))$$
$$=\frac{1}{2}-\frac{1}{z-\ln 2}-\sum_{k\geq 1}\left(\frac{1}{z-\ln 2-2\pi ik}+\frac{1}{z-\ln 2+2\pi ik}\right)$$

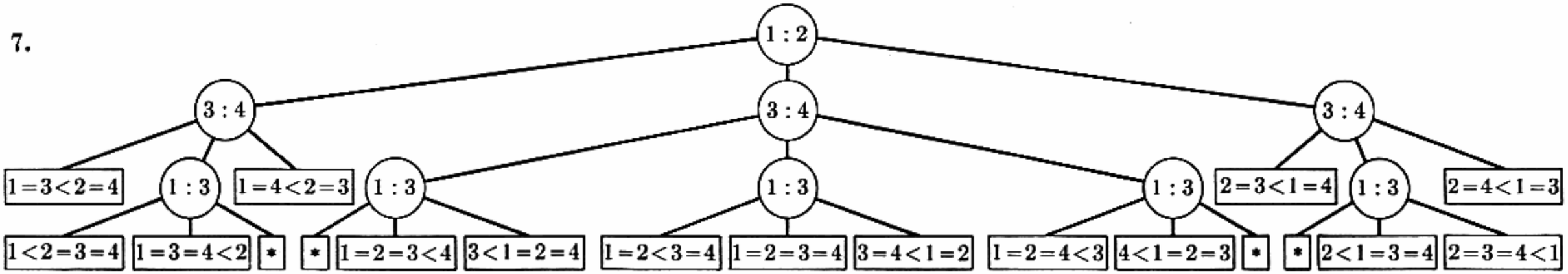
yields a convergent series $P_n/n!=\frac{1}{2}(\ln 2)^{-n-1}+\sum_{k\geq 1}\Re((\ln 2+2\pi ik)^{-n-1})$.

5.



6. $S'(n)\geq S(n)$, since the keys might all be distinct; thus we must show that $S'(n)\leq S(n)$. Given a sorting algorithm which takes $S(n)$ steps on distinct keys, we can construct a sorting algorithm for the general case by defining the $=$ branch to be identical to the $<$ branch, removing redundancies. When an external node appears, we know all of the equality relations, since we have $K_{a_1}\leq K_{a_2}\leq\cdots\leq K_{a_n}$ and an explicit comparison $K_{a_i}:K_{a_{i+1}}$ has been made for $1\leq i<n$.

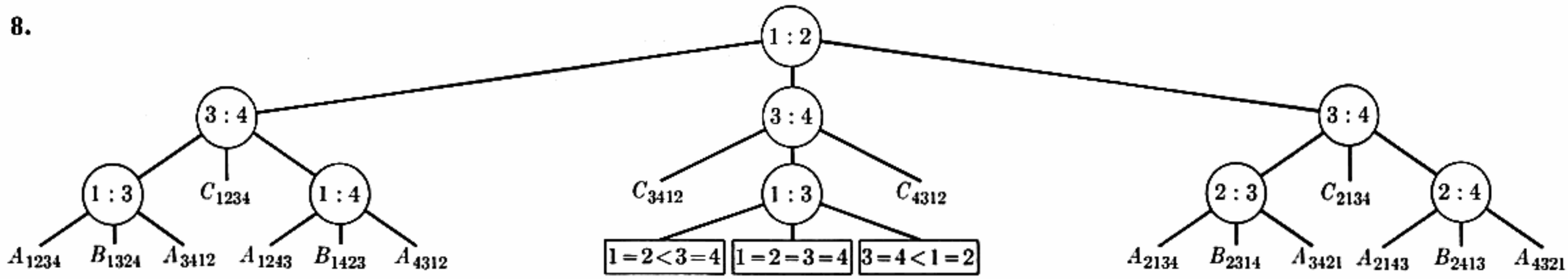
7.



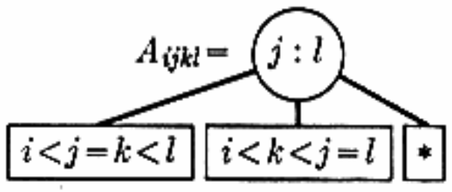
(* denotes an impossible case.)

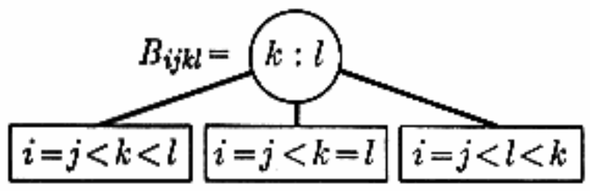
The average number of comparisons is $(2+3+3+2+3+3+3+6+3+3+3+2+3+3+2)/16 = 2\frac{3}{4}$.

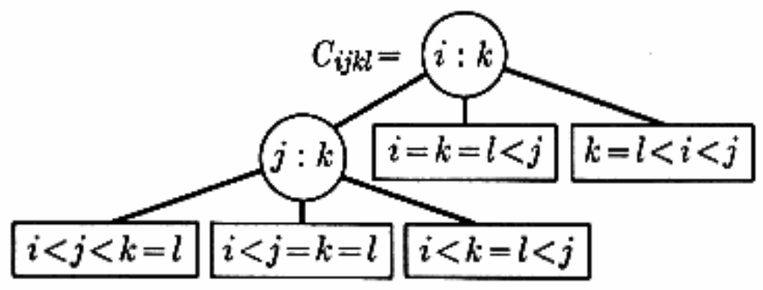
8.



where







Average number of comparisons is $3\frac{56}{81}$.

9. We need at least $n - 1$ comparisons to discover that all keys are equal, if they are. Conversely, $n - 1$ always suffices, since we can always deduce the final ordering after comparing K_1 with all of the other keys!

10. Let $f(n)$ be the desired function, and let $g(n)$ be the minimum average number of comparisons needed to sort $n + k$ elements when $k > 0$ and exactly k of the elements have known values (0 or 1). Then $f(0) = f(1) = g(0) = 0$, $g(1) = 1$; $f(n) = 1 + \frac{1}{2}f(n-1) + \frac{1}{2}g(n-2)$, $g(n) = 1 + \min\left(g(n-1), \frac{1}{2}g(n-1) + \frac{1}{2}g(n-2)\right) = 1 + \frac{1}{2}g(n-1) + \frac{1}{2}g(n-2)$, for $n \geq 2$. (Thus the best strategy is to compare two unknown elements whenever possible.) It follows that $f(n) - g(n) = \frac{1}{2}\big(f(n-1) - g(n-1)\big)$ for $n \geq 2$, and $g(n) = \frac{2}{3}\big(n + \frac{1}{3}(1 - (-\frac{1}{2})^n)\big)$ for $n \geq 0$. Hence the answer is $\frac{2}{3}n + \frac{2}{9} - \frac{2}{9}(-\frac{1}{2})^n - (\frac{1}{2})^{n-1}$, for $n \geq 1$. (This exact formula may be compared with the information-theoretic lower bound, $\log_3 (2^n - 1) \approx 0.6309n$.)

11. Binary insertion proves that $S_m(n) \leq B(m) + (n - m)\lceil \log_2 (m+1) \rceil$, for $n \geq m$. On the other hand $S_m(n) \geq \lceil \log_2 \sum_{1 \leq k \leq m} \left\{ {n \atop k} \right\} k! \rceil$ and this is asymptotically $n \log_2 m$ (cf. Eq. 1.2.6–49).

12. (a) If there are no redundant comparisons, we can arbitrarily assign an order to keys which are actually equal, when they are first compared, since no order can be deduced from previously made comparisons. (b) Assume that the tree strongly sorts every sequence of zeros and ones; we shall prove that it strongly sorts every permutation of $\{1, 2, \ldots, n\}$. Suppose it doesn't, so that there is a permutation for which it claims that $K_{a_1} \leq K_{a_2} \leq \cdots \leq K_{a_n}$ whereas in fact $K_{a_i} > K_{a_{i+1}}$ for some i. Replace all elements $< K_{a_i}$ by 0 and all elements $\geq K_{a_i}$ by 1; by assumption the method will now sort when we take the path which leads to $K_{a_1} \leq K_{a_2} \leq \cdots \leq K_{a_n}$, a contradiction.

13. If n is even, $F(n) - F(n-1) = 1 + F(\lfloor n/2 \rfloor) - F(\lfloor n/2 \rfloor - 1)$ so we must prove that $w_{k-1} < \lfloor n/2 \rfloor \leq w_k$; this is obvious since $w_{k-1} = \lfloor w_k/2 \rfloor$. If n is odd, $F(n) - F(n-1) = G(\lceil n/2 \rceil) - G(\lfloor n/2 \rfloor)$, so we must prove that $t_{k-1} < \lceil n/2 \rceil \leq t_k$; this is obvious since $t_{k-1} = \lceil w_k/2 \rceil$.

14. By exercise 1.2.4–42, the sum is $n\lceil \log_2 (\frac{3}{4}n) \rceil - (w_1 + \cdots + w_j)$ where $w_j < n \leq w_{j+1}$. The latter sum is $w_{j+1} - \lfloor j/2 \rfloor - 1$. $F(n)$ can therefore be written $n\lceil \log_2 (\frac{3}{4})n \rceil - \lfloor 2^{\lfloor \log_2 (6n) \rfloor}/3 \rfloor + \lfloor \frac{1}{2} \log_2 (6n) \rfloor$ (and in many other ways).

15. If $\lceil \log_2 (\frac{3}{4}n) \rceil = \log_2 (\frac{3}{4}n) + \theta$, $F(n) = n \log_2 n - (3 - \log_2 3)n + n(\theta + 1 - 2^\theta) + O(\log n)$. If $\lceil \log_2 n \rceil = \log_2 n + \theta$, $B(n) = n \log_2 n - n + n(\theta + 1 - 2^\theta) + O(\log n)$. [Note that $\log_2 n! = n \log_2 n - n/(\ln 2) + O(\log n)$; $1/(\ln 2) \approx 1.443$; $3 - \log_2 3 \approx 1.415$.]

17. The number of cases with $b_k < a_p < b_{k+1}$ is

$$\binom{m - p + n - k}{m - p} \binom{p - 1 + k}{p - 1},$$

and the number of cases with $a_j < b_q < a_{j+1}$ is

$$\binom{n - q + m - j}{n - q} \binom{q - 1 + j}{q - 1}.$$

18. No, since we are considering only the less efficient branch of the tree below each comparison. One of the more efficient branches might turn out to be harder to handle.

20. Let L be the maximum level on which an external node appears, and let l be the minimum such level. If $L \geq l+2$, we can remove two nodes from level L and place them below a node at level l; this decreases the external path length by $l+2L-(L-1+2(l+1)) = L-l-1 \geq 1$. Conversely, if $L \leq l+1$, let there be k external nodes on level $l-1$ and $N-k$ on level l, where $0 < k \leq N$. By exercise 2.3.4.5–3, $k2^{-l+1} + (N-k)2^{-l} = 1$; hence $N+k = 2^l$. The inequalities $N < 2^l \leq 2N$ now show that $l = \lfloor \log_2 N \rfloor + 1$; this defines k and yields the external path length (33).

21. Let $r(x)$ be the root of x's right subtree. We need $\lceil \log_2 t(l(x)) \rceil \leq \lceil \log_2 t(x) \rceil - 1$ and $\lceil \log_2 t(r(x)) \rceil \leq \lceil \log_2 t(x) \rceil - 1$. The first condition is equivalent to $2t(l(x)) - t(x) \leq 2^{\lceil \log_2 t(x) \rceil} - t(x)$, and the second condition is equivalent to $t(x) - 2t(l(x)) \leq 2^{\lceil \log_2 t(x) \rceil} - t(x)$.

22. By exercise 20, the conditions $\lfloor \log_2 t(l(x)) \rfloor, \lfloor \log_2 t(r(x)) \rfloor \geq \lfloor \log_2 t(x) \rfloor - 1$ and $\lceil \log_2 t(l(x)) \rceil, \lceil \log_2 t(r(x)) \rceil \leq \lceil \log_2 t(x) \rceil - 1$ are necessary and sufficient. Arguing as in exercise 21, we can prove them equivalent to the stated conditions. [Cf. Martin Sandelius, *AMM* **68** (1961), 133–134.] See exercise 33 for a generalization.

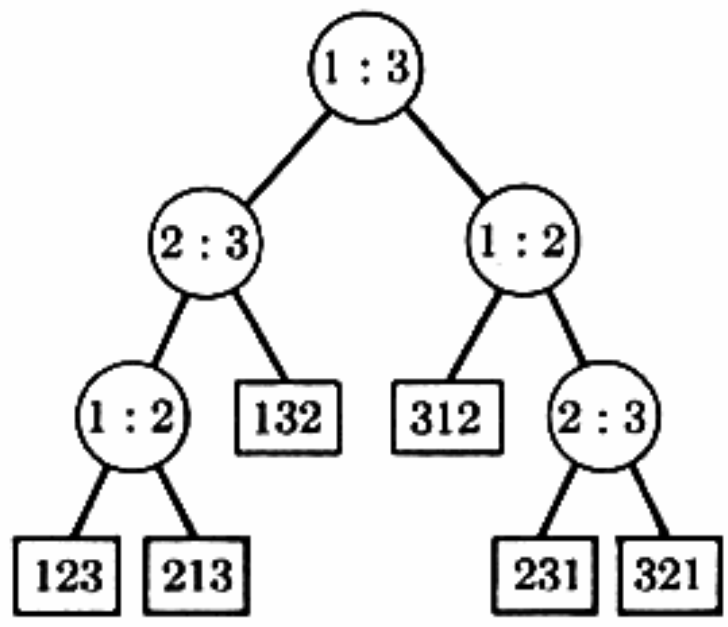
23. Multiple list insertion assumes that the keys are uniformly distributed in a known range, so it isn't a "pure comparison" method satisfying the restrictions considered in this section.

24. First proceed as if sorting five elements, until after five comparisons we reach one of the configurations in (6). In the first three cases, complete sorting the five elements in two more comparisons, then insert the sixth element f. In the latter case, first compare $f:b$, insert f into the main chain, then insert c. [*Théorie des Questionnaires*, p. 116.]

25. Since $N = 7! = 5040$ and $q = 13$, there would be $8192 - 5040 = 3152$ external nodes on level 12 and $5040 - 3152 = 1888$ on level 13.

26. (The best methods known to date have an external path length of 62416.)

27.



is the *only* way to recognize the two most frequent permutations with two comparisons, even though the first comparison produces a .27–.73 split!

28. Lun Kwan has constructed an 873-line program whose average running time is $38.925u$. Its maximum running time is $43u$; the latter appears to be optimal since it is the time for 7 compares, 7 tests, 6 loads, 5 stores.

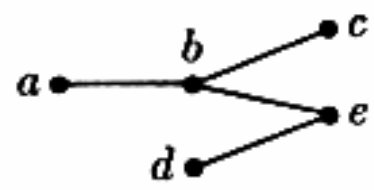
29. We must make at least $S(n)$ comparisons, because it is impossible to know whether a permutation is even or odd unless we have made enough comparisons to determine it uniquely. For we can assume that enough comparisons have been made to narrow things down to two possibilities that depend on whether or not a_i is less than a_j, for some i and j; one of the two possibilities is even, the other is odd. [On the other hand

there *is* an $O(n)$ algorithm for this problem, which simply counts the number of cycles and uses no comparisons at all; see exercise 5.2.2–2.]

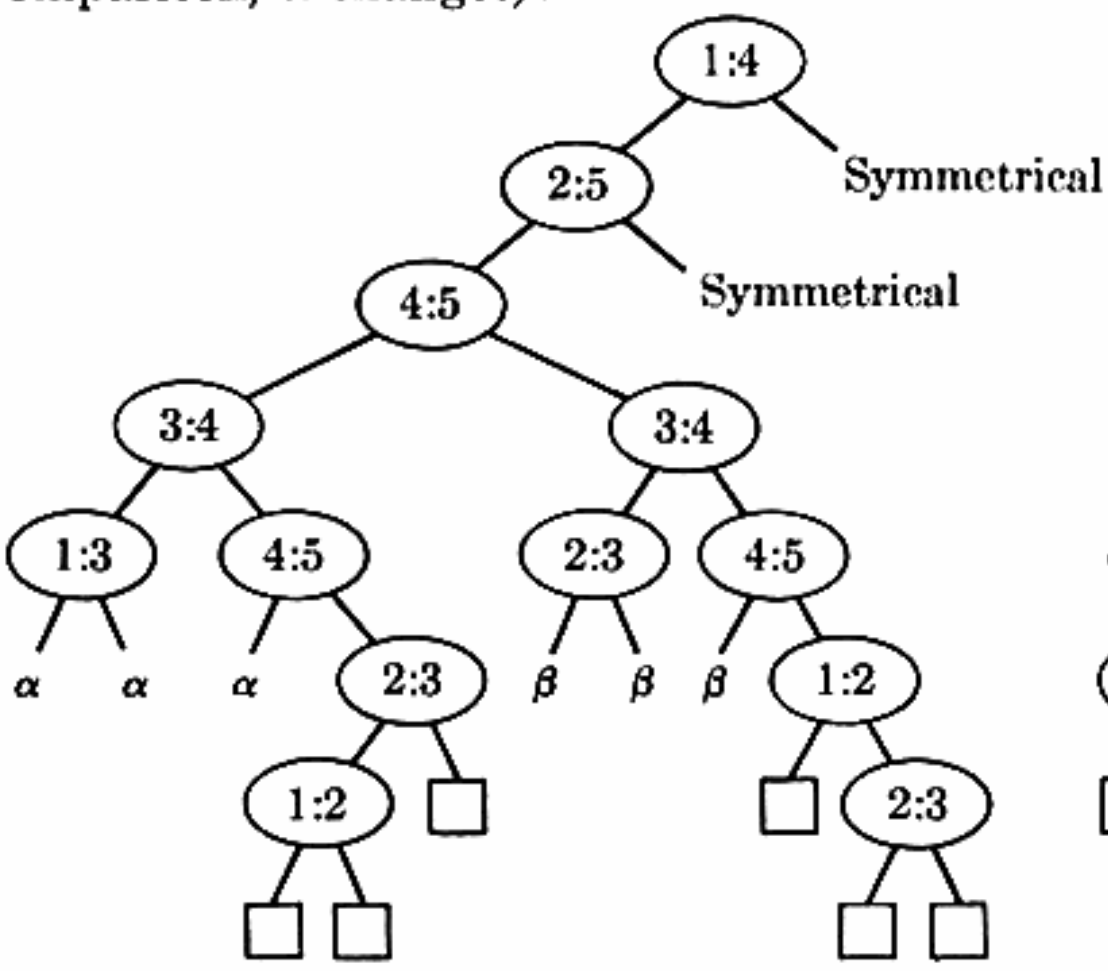
30. Start with an optimal comparison tree of height $S(n)$; repeatedly interchange $i \leftrightarrow j$ in the right subtree of a node labeled $i{:}j$, from top to bottom. Interpreting the result as a comparison-exchange tree, every terminal node defines a unique permutation which can be sorted by at most $n - 1$ more comparison-exchanges (by exercise 5.2.2–2).

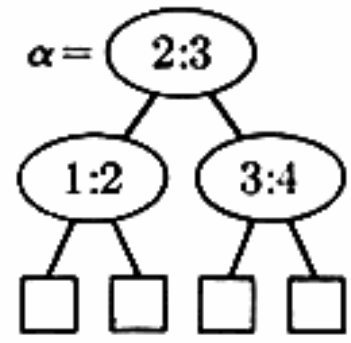
[The idea of a comparison-exchange tree is due to T. N. Hibbard.]

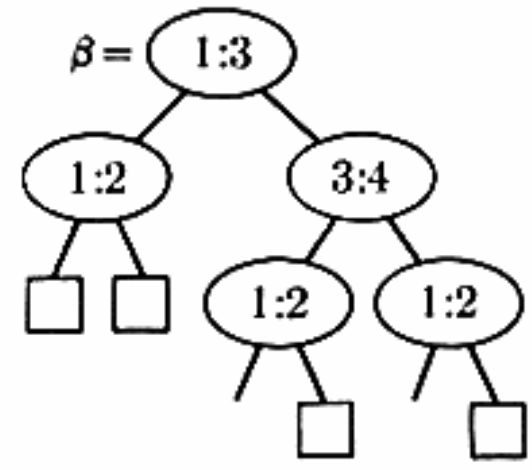
31. At least 8 are required, since every tree of height 7 will produce the configuration



(or its dual) in some branch after 4 steps, with $a \neq 1$. This configuration cannot be sorted in 3 more comparison/exchange operations. On the other hand the following tree achieves the desired bound (and perhaps also the minimum *average* number of comparison/exchanges):







33. Simple operations applied to any tree of order x and resolution 1 can be applied to yield another whose weighted path length is no greater and such that (a) all external nodes lie on levels k and $k - 1$ for some k; (b) at most one external node is non-integer, and if it is present it lies on level k. The weighted path length of any such tree has the stated value, so this must be minimal. Conversely if (iv) and (v) hold in any real-valued search tree it is possible to show by induction that the weighted path length has the stated value, since there is a simple formula for the weighted path length of a tree in terms of the weighted path lengths of the two subtrees of the root.

SECTION 5.3.2

1. $S(m+n) \leq S(m) + S(n) + M(m, n)$.

2. The internal node which is kth in symmetric order corresponds to the comparison $A_1 : B_k$.

3. Strategy $B(1, l)$ is no better than strategy $A(1, l+1)$, and strategy $B'(1, l)$ no

better than $A'(1, l-1)$; hence we must solve the recurrence

$$.M.(1,n) = \min_{1\le j\le n} \max\left(\max_{1\le l\le j} (1+.M.(1,l-1)), \max_{j\le l\le n} (1+.M.(1,n-l)) \right), \qquad n \ge 1;$$

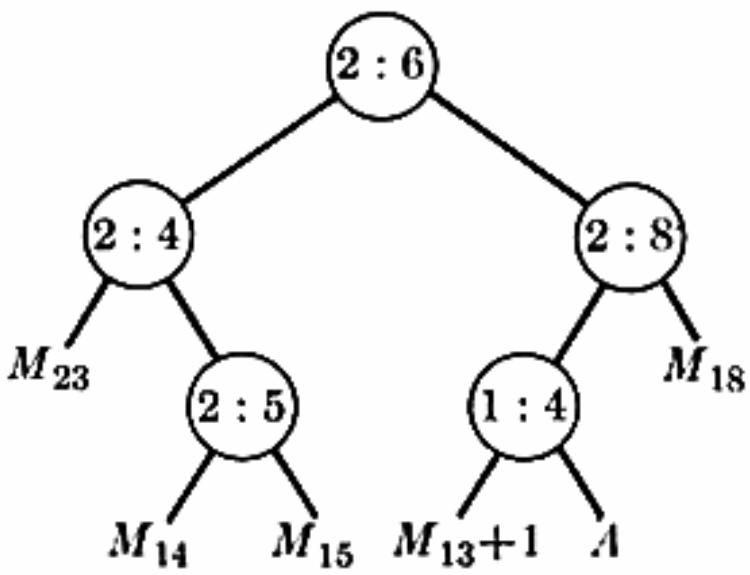
$.M.(1,0) = 0$. It is not difficult to verify that $\lceil \log_2 (n+1) \rceil$ satisfies this recurrence.

6. Strategy $A'(i, i+1)$ can be used when $j = i+1$, except when $i \le 2$. And we can use strategy $A(i, i+2)$ when $j \ge i+2$.

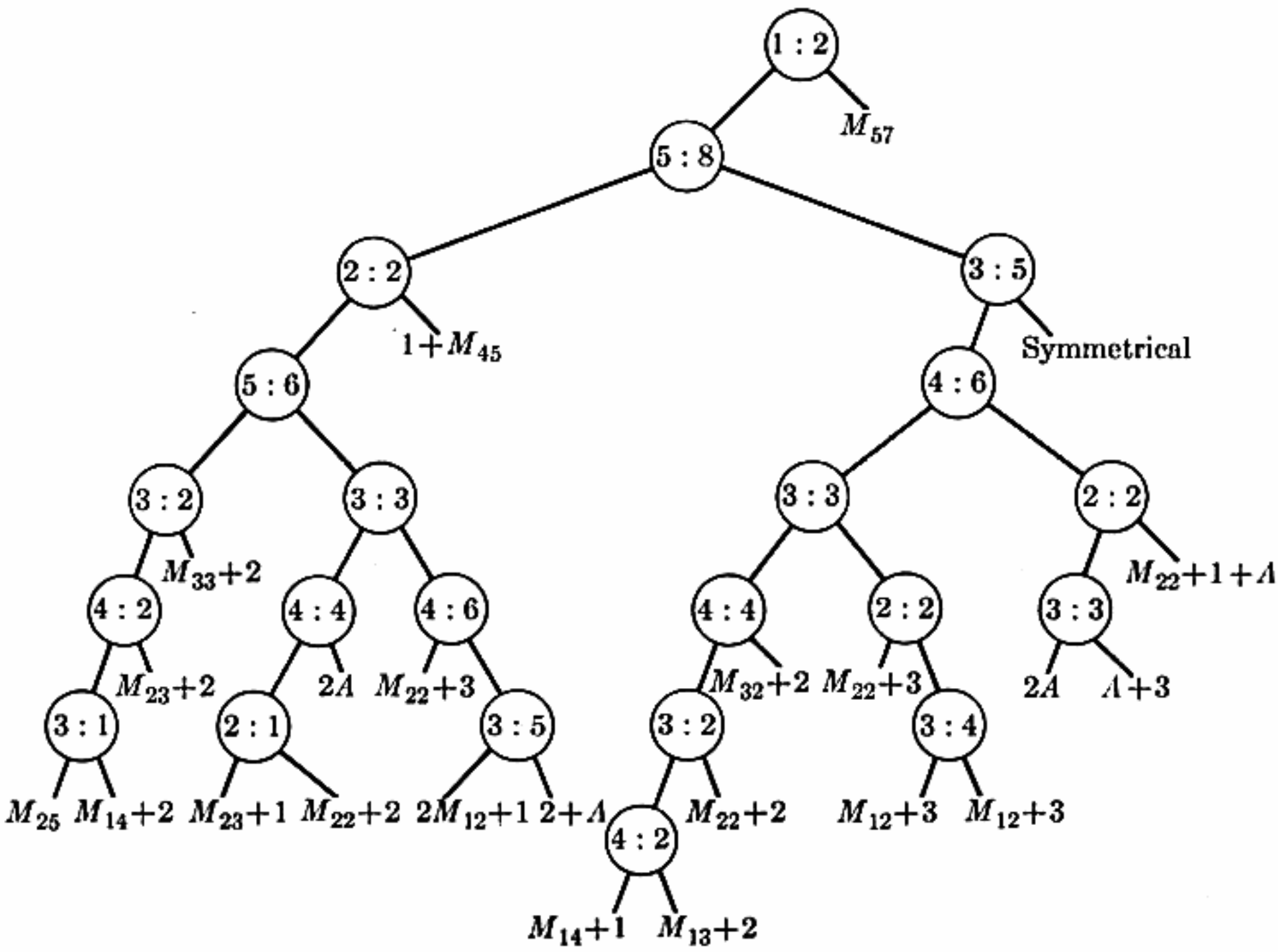
7. To insert $k+m$ elements among n others, independently insert k elements and m elements. (When k and m are large, an improved procedure is possible, see exercise 19.)

8. In the following diagrams, $i{:}j$ denotes the comparison $A_i{:}B_j$, M_{ij} denotes merging i elements with j in $M(i, j)$ steps, and A denotes sorting the pattern or in three steps.

9.



10.



11. Let $n = g_t$ as in the hint. We may assume that $t \geq 6$. Without loss of generality let $A_2{:}B_j$ be the first comparison. If $j > g_{t-1}$, the outcome $A_2 < B_j$ will require $\geq t$ more steps. If $j \leq g_{t-1}$, the outcome $A_2 < B_j$ would be no problem, so only the case $A_2 > B_j$ needs study, and we get the most information when $j = g_{t-1}$. If $t = 2k + 1$, we might have to merge A_2 with the $g_t - g_{t-1} = 2^{k-1}$ elements $> B_{g_{t-1}}$, and merge A_1 with the g_{t-1} others, but this requires $k + (k + 1) = t$ further steps. On the other hand if $n = g_t - 1$, we could merge A_2 with $2^{k-1} - 1$ elements, then A_1 with n elements, in $(k - 1) + (k + 1)$ further steps, hence $M(2, g_t - 1) \leq t$.

The case $t = 2k$ is considerably more difficult; note that $g_t - g_{t-1} \geq 2^{k-2}$. After $A_2 > B_{g_{t-1}}$, suppose we compare $A_1{:}B_j$. If $j > 2^{k-1}$ the outcome $A_1 < B_j$ requires $k + (k - 1)$ further comparisons (too many). If $j \leq 2^{k-1}$, we can argue as before that $j = 2^{k-1}$ gives most information. After $A_1 > B_{2^{k-1}}$, the next comparisons with A_1 might as well be with $B_{2^{k-1}+2^{k-2}}$, then $B_{2^{k-1}+2^{k-2}+2^{k-3}}$; since $2^{k-1} + 2^{k-2} + 2^{k-3} > g_{t-1}$, we are left with merging $\{A_1, A_2\}$ with $n - (2^{k-1} + 2^{k-2} + 2^{k-3})$ elements. Of course we needn't make any comparisons with A_1 right away; we could instead compare $A_2{:}B_{n+1-j}$. If $j \leq 2^{k-3}$, we consider the case $A_2 < B_{n+1-j}$, while if $j > 2^{k-3}$ we consider $A_2 > B_{n+1-j}$. The latter case requires at least $(k - 2) + (k + 1)$ more steps. Continuing, we find that the *only* potentially fruitful line is $A_2 > B_{g_{t-1}}$, $A_2 < B_{n+1-2^{k-3}}$, $A_1 > B_{2^{k-1}}$, $A_1 > B_{2^{k-1}+2^{k-2}}$, $A_1 > B_{2^{k-1}+2^{k-2}+2^{k-3}}$, but then we have exactly g_{t-5} elements left! Conversely, if $n = g_t - 1$, this line works. [*Acta Informatica* 1 (1971), 145–158.]

12. The first comparison must be either $\alpha{:}X_k$ for $1 \leq k \leq i$, or (symmetrically) $\beta{:}X_{n-k}$ for $1 \leq k \leq j$. In the former case the response $\alpha < X_k$ leaves us with $R_n(k - 1, j)$ more comparisons to make; the response $\alpha > X_k$ leaves us with the problem of sorting $\alpha < \beta$, $Y_1 < \cdots < Y_{n-k}$, $\alpha < Y_{i-k+1}$, $\beta > Y_{n-k-j}$, where $Y_r = X_{r-k}$.

13. [*Computers in Number Theory* (New York: Academic Press, 1971), 397–404.]

15. Double m until it exceeds n. This involves $\lfloor \log_2 (n/m) \rfloor + 1$ doublings.

16. All except $(m, n) = (2, 8)$, $(3, 6)$, $(3, 8)$, $(3, 10)$, $(4, 8)$, $(4, 10)$, $(5, 9)$, $(5, 10)$, when it's one over.

17. Assume that $m \leq n$ and let $t = \log_2 (n/m) - \theta$. Then $\log_2 \binom{m+n}{m} > \log_2 n^m - \log_2 m! \geq m \log_2 n - (m \log_2 m - m + 1) = m(t + \theta) + m - 1 = H(m, n) + \theta m - \lfloor 2^\theta m \rfloor \geq H(m, n) + \theta m - 2^\theta m \geq H(m, n) - m$. (The inequality $m! \leq m^m 2^{1-m}$ is a consequence of the fact that $k(m - k) \leq (m/2)^2$ for $1 \leq k < m$.)

19. First merge $\{A_1, \ldots, A_m\}$ with $\{B_2, B_4, \ldots, B_{2\lfloor n/2 \rfloor}\}$. Then we must insert the odd elements B_{2i-1} among a_i of the A's for $1 \leq i \leq \lceil n/2 \rceil$, where $a_1 + a_2 + \cdots + a_{\lceil n/2 \rceil} \leq m$. The latter operation requires $\leq a_i$ operations for each i, so at most m more comparisons will finish the job.

20. Apply (12).

23. The oracle keeps an $n \times n$ matrix X whose entries x_{ij} are initially all 1. When the algorithm asks if $A_i = B_j$, the oracle sets x_{ij} to 0. The answer is "No," unless the permanent of X has just become zero. In the latter case, the oracle answers "Yes," (as it must, lest the algorithm terminate immediately!), and deletes row i and column j from X; the resulting $(n - 1) \times (n - 1)$ matrix will have a nonzero permanent. The oracle continues in this way until only a 0×0 matrix is left.

One can show (for example by using Philip Hall's theorem, cf. Chapter 7) that at least n zeroes are deleted when the oracle first answers "Yes," and $n - 1$ the second

time, etc. The algorithm will terminate only after receiving n "Yes" answers to non-redundant questions, and after asking at least $n + (n - 1) + \cdots + 1$ questions.

SECTION 5.3.3

1. Player *11* lost to *05*, so *13* was known to be worse than *05*, *11*, *12*.

2. Let x be the tth largest, and let S be the set of all elements y such that the comparisons made are insufficient to prove either that $x < y$ or $y < x$. There are permutations, consistent with all the comparisons made, in which all elements of S are less than x; for we can stipulate that all elements of S are less than x and embed the resulting partial ordering in a linear ordering. Similarly there are consistent permutations in which all elements of S are greater than x. Hence we don't know the rank of x unless S is empty.

3. An oracle may regard the loser of the first comparison as the worst player of all. [R. W. Floyd has used a similar approach to show that the values in Table 1 are exact for all $n \leq 7$.]

4. There are at least $n^{\underline{t}}$ outcomes.

5. In fact, $W_t(n) \leq V_t(n) + S(t-1)$, by exercise 2.

6. Let $g(l_1, l_2, \ldots, l_m) = m - 2 + \log_2 (2^{l_1} + 2^{l_2} + \cdots + 2^{l_m})$, and assume that $f = g$ whenever $l_1 + l_2 + \cdots + l_m + 2m < N$. We shall prove that $f = g$ when $l_1 + l_2 + \cdots + l_m + 2m = N$. We may assume that $l_1 \geq l_2 \geq \cdots \geq l_m$. There are only a few possible ways to make the first comparison:

Strategy $A(j, k)$, *for* $j < k$. Compare the largest element of group j with the largest of group k. This gives the relation

$$\begin{aligned} f(l_1, \ldots, l_m) &\leq 1 + g(l_1, \ldots, l_{j-1}, l_j + 1, l_{j+1}, \ldots, l_{k-1}, l_{k+1}, \ldots, l_m) \\ &= g(l_1, \ldots, l_{j-1}, l_j, l_{j+1}, \ldots, l_{k-1}, l_j, l_{k+1}, \ldots, l_m) \geq g(l_1, \ldots, l_m). \end{aligned}$$

Strategy $B(j, k)$, *for* $l_k > 0$. Compare the largest element of group j with one of the small elements of group k. This gives the relation

$$f(l_1, \ldots, l_m) \leq 1 + \max(\alpha, \beta) = 1 + \beta,$$

where

$$\begin{aligned} \alpha &= g(l_1, \ldots, l_{j-1}, l_{j+1}, \ldots, l_m) \leq g(l_1, \ldots, l_m) - 1, \\ \beta &= g(l_1, \ldots, l_{k-1}, l_k - 1, l_{k+1}, \ldots, l_m) \geq g(l_1, \ldots, l_m) - 1. \end{aligned}$$

Strategy $C(j, k)$, *for* $j \leq k$, $l_j > 0$, $l_k > 0$. Compare a small element from group j with a small element from group k. The corresponding relation is

$$f(l_1, \ldots, l_m) \leq 1 + g(l_1, \ldots, l_{k-1}, l_k - 1, l_{k+1}, \ldots, l_m) \geq g(l_1, \ldots, l_m).$$

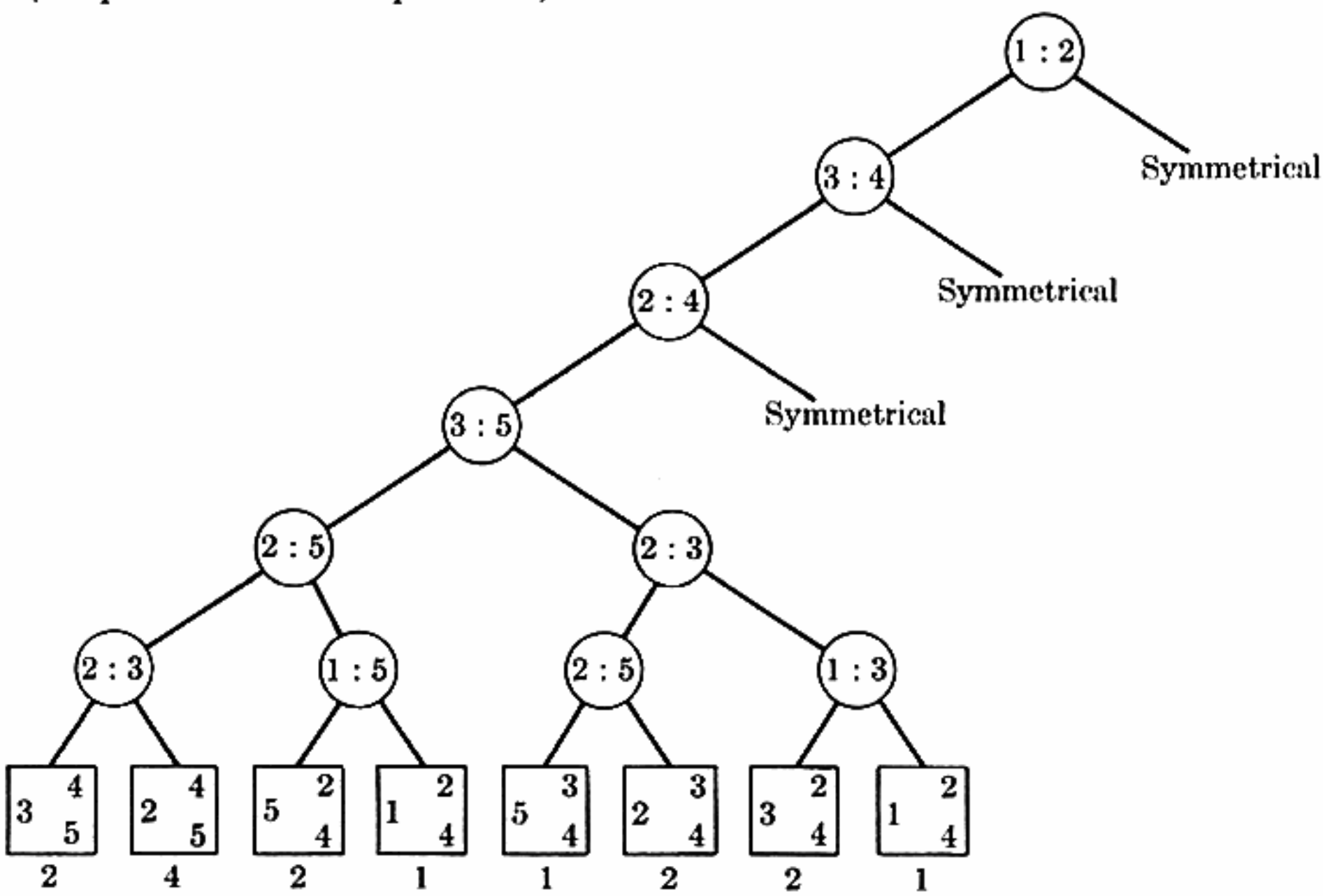
The value of $f(l_1, \ldots, l_m)$ is found by taking the minimum right-hand side over all these strategies; hence $f(l_1, \ldots, l_m) \geq g(l_1, \ldots, l_m)$. When $m > 1$, Strategy $A(m-1, m)$ shows that $f(l_1, \ldots, l_m) \leq g(l_1, \ldots, l_m)$, since $g(l_1, \ldots, l_{m-1}, l_m) = g(l_1, \ldots, l_{m-1}, l_{m-1})$ when $l_1 \geq \cdots \geq l_m$. (*Proof*: $\lceil \log_2 (M + 2^a) \rceil = \lceil \log_2 (M + 2^b) \rceil$ for $0 \leq a \leq b$, when M is a positive multiple of 2^b.) When $m = 1$, use Strategy $C(1, 1)$.

[S. S. Kislitsyn's paper determined the optimum strategy $A(m-1, m)$ and evaluated $f(l, l, \ldots, l)$ in closed form; the general formula for f and this simplified proof were discovered by Floyd in 1970.]

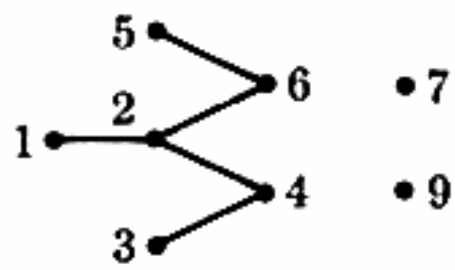
7. For $j > 1$, if $j+1$ is in α', c_j is 1 plus the number of comparisons needed to select the next largest element of α'. Similarly if $j+1$ is in α''; and c_1 is always 0, since the tree always looks the same at the end.

8. In other words, is there an extended binary tree with n external nodes such that the sum of the distances to the $t-1$ farthest internal nodes from the root is less than the corresponding sum for the complete binary tree? The answer is no, since it is not hard to show that the kth largest element of $\mu(\alpha)$ is $\geq \lfloor \log_2 (n-k) \rfloor$ for all α.

9. (All paths use six comparisons.)



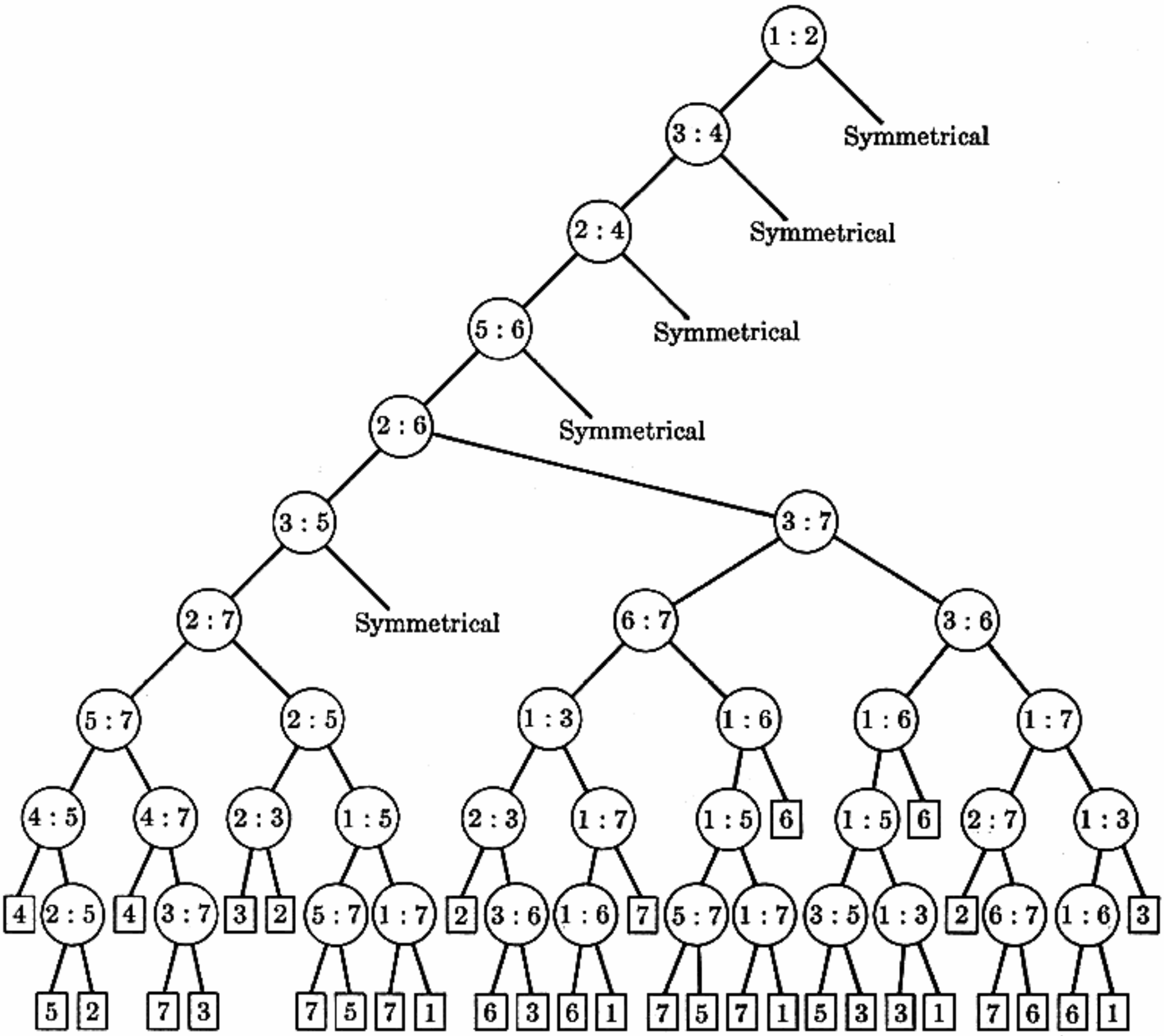
10. After the first seven comparisons in Doren's method, we may discard X_8, and so we must find the fourth largest of



This is more information than we knew after four steps of Doren's method, so at most $12 - 4 = 8$ more comparisons are required.

11. After discarding the smallest of $\{X_1, X_2, X_3, X_4\}$, we have the configuration •—• plus $n-3$ isolated elements; the third largest of these can be found in $V_3(n-1) - 1$ further steps.

12. (Found manually by trial and error, using exercise 6 to help find fruitful lines.)



13. After finding the median of the first $f(n)$ elements, say X_j, compare it to each of the others; this splits the elements into approximately $n/2 - k$ less than X_j and $n/2 + k$ greater than X_j, for some k. It remains to find the $|k|$th largest or smallest element of the bigger set, which requires $n/2 + O(|k| \log n)$ further comparisons. The average value of $|k|$ (consider points uniformly distributed in [0, 1]) is $O(1/\sqrt{n}) + O(n/\sqrt{f(n)})$. Let $T(n)$ be the average number of comparisons when $f(n) = n^{2/3}$; $T(n) - n = T(n^{2/3}) - n^{2/3} + n/2 + O(n^{2/3})$, and the result follows.

It is interesting to note that when $n = 5$, this method requires only $5\frac{13}{15}$ comparisons on the average, slightly better than the tree of exercise 9.

14. In general, the t largest can be found in $V_t(n-1) + 1$ comparisons, by finding the tth largest of $\{X_1, \ldots, X_{n-1}\}$ and comparing it with X_n, because of exercise 2.

15. $\min(t, n + 1 - t)$. Assuming that $t \leq n + 1 - t$, if we don't save each of the first t words when they are first read in, we may have forgotten the tth largest, depending on the subsequent values still unknown to us. Conversely, t locations are sufficient, since we can compare a newly input item with the previous tth largest, storing the register iff it is greater.

16. The algorithm starts with $(a, b, c, d) = (n, 0, 0, 0)$ and ends with $(0, 1, 1, n-2)$. If the oracle avoids "surprising" outcomes, the only transitions possible after each comparison are from (a, b, c, d) to itself or to

$$\begin{array}{cc} (a-2, b+1, c+1, d), & \text{if } a \geq 2; \\ (a-1, b, c+1, d) \quad \text{or} \quad (a-1, b+1, c, d), & \text{if } a \geq 1; \\ (a, b-1, c, d+1), & \text{if } b \geq 2; \\ (a, b, c-1, d+1), & \text{if } c \geq 2. \end{array}$$

It follows that $\lceil \frac{3}{2}a \rceil + b + c - 2$ comparisons are needed to get from (a, b, c, d) to $(0, 1, 1, a+b+c+d-2)$. [*CACM* **15** (1972), 462–464.]

17. Use (6) first for the largest, then for the smallest, noting that $\lfloor n/2 \rfloor$ of the comparisons are common to both.

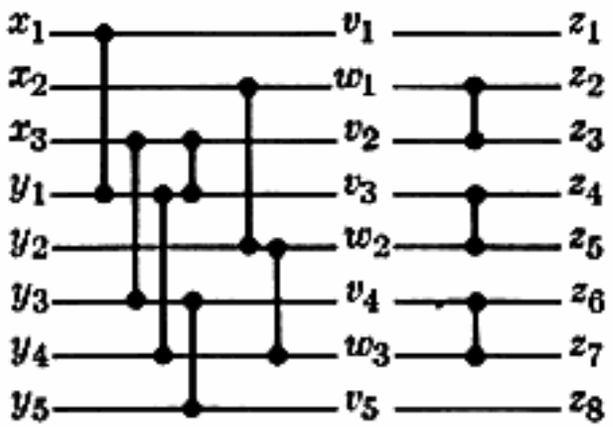
18. $V_t(n) \leq 18n - 151$, for all sufficiently large n.

21. Let the elements be $\{x_1, \ldots, x_n, y, z\}$, where $n = 2^k$. Find the two largest x's, say $b < a$, in $2^k + k - 2$ comparisons, using Kislitsyn's method; and compare $y:z$ (say $y < z$). Now if $z < b$, continue Kislitsyn's method, finding the third largest of the x's in $k-1$ comparisons, and compare it to z. On the other hand, if $z > b$, it is easy to finish in at most three more steps.

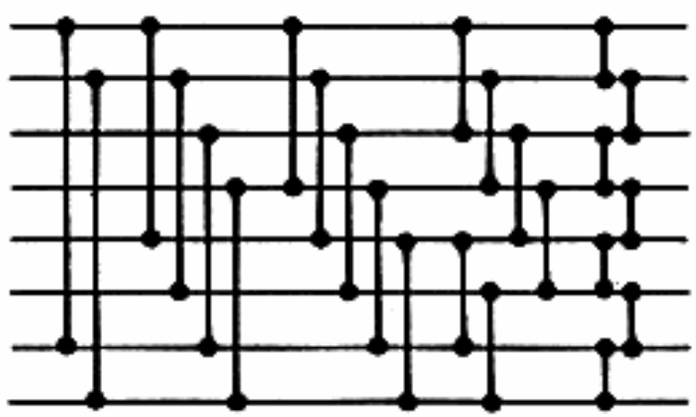
SECTION 5.3.4

1. (When m is odd it is best to have v_m followed by v_{m+1}, w_{m+1}, v_{m+2}, . . . instead of by w_{m+1}, v_{m+1}, w_{m+2}, . . . in the diagram.)

(3, 5) odd-even merge



Pratt eight-sort

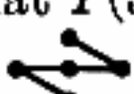


2. See the above diagram for $n = 8$.

3. $C(m, m-1) = C(m, m) - 1$, for $m \geq 1$.

4. If $\hat{T}(6) = 4$, there would be three comparators acting at each time, since $\hat{S}(6) = 12$. But then removing the bottom line and its four comparators would give $\hat{S}(5) \leq 8$, a contradiction. [The same argument yields $\hat{T}(7) = \hat{T}(8) = 6$.]

5. Let $f(n) = f(\lceil n/2 \rceil) + 1 + \lceil \log_2 \lceil n/2 \rceil \rceil$, if $n \geq 2$; by exercises 1.2.4–34, 35, $f(n) = (1 + \lceil \log_2 \frac{1}{2}n \rceil) + (1 + \lceil \log_2 \frac{1}{4}n \rceil) + \cdots$.

6. We may assume that each stage makes $\lfloor n/2 \rfloor$ comparisons (extra comparisons can't hurt). Since $\hat{T}(6) = 5$, it suffices to show that $T(5) = 5$. After two stages when $n = 5$, we cannot avoid the partial orderings  or  which cannot be sorted in two more stages.

7. Assume that the input keys are $\{1, 2, \ldots, 10\}$. The key fact is that after the

first 16 comparators, lines 2–5 cannot contain 8 or 9, nor can they contain both 6 and 7.

8. Straightforward generalization of Theorem F.

9. $\hat{M}(3, 3) \geq \hat{S}(6) - 2\hat{S}(3)$; $\hat{M}(4, 4) \geq \hat{S}(8) - 2\hat{S}(4)$; $\hat{M}(5, 5) \geq 2\hat{M}(2, 3) + 3$ by exercise 8; and $\hat{M}(2, 3) \geq \hat{S}(5) - \hat{S}(2) - \hat{S}(3)$.

10. The hint follows by the method of proof in Theorem Z. Then show that the number of 0's in the even subsequence minus the number of 0's in the odd subsequence is ± 1.

11. (Solution by M. W. Green.) The network is symmetric in the sense that, whenever z_i is compared to z_j, there is a corresponding comparison of z_{2^p-1-j}:z_{2^p-1-i}. Any symmetric network capable of sorting a sequence $\langle z_0, \ldots, z_{2^p-1}\rangle$ will also sort the sequence $\langle -z_{2^p-1}, \ldots, -z_0\rangle$.

Batcher has observed that the network will actually sort any cyclic shift $\langle z_j, z_{j+1}, \ldots, z_{2^p-1}, z_0, \ldots, z_{j-1}\rangle$ of a bitonic sequence. This is a consequence of the 0-1 principle.

12. $x \vee y$ is (consider 0-1 sequences), but not $x \vee y$ (consider $\langle 3, 1, 4, 5\rangle \wedge \langle 6, 7, 8, 2\rangle$).

13. A perfect shuffle has the effect of replacing z_i by z_j, where the binary representation of j is that of i rotated cyclically to the right one place (cf. exercise 3.4.2–13). Consider shuffling the comparators instead of the lines; then the first column of comparators acts on the pairs $z[i]$ and $z[i \oplus 2^{r-1}]$, the next column on $z[i]$ and $z[i \oplus 2^{r-2}]$, ..., the tth column on $z[i]$ and $z[i \oplus 1]$, the $(t+1)$st column on $z[i]$ and $z[i \oplus 2^{r-1}]$ again, etc. Here "$\oplus$" denotes exclusive-or on the binary representation. This shows that Fig. 57 is equivalent to Fig. 56; after s stages we have groups of 2^s elements which are alternately sorted and reverse-sorted.

14. After l levels the input x_1 can be in at most 2^l different places. After merging is complete, x_1 can be in $n+1$ different places.

15. [1:4][3:2][1:3][2:4][2:3].

16. The process clearly terminates. Each execution of step T2 has the effect of interchanging the i_qth and j_qth outputs, so the result of the algorithm is to permute the output lines in some way. Since the resulting (standard) network makes no change to the input $\langle 1, 2, \ldots, n\rangle$, the output lines must have been returned to their original position.

17. Make the network standard by the algorithm of exercise 16; then by considering the input sequence $\langle 1, 2, \ldots, n\rangle$, we see that standard selection networks must take the t largest elements into the t highest-numbered lines; and a $\hat{V}_t(n)$ network must take the tth largest into line $n+1-t$. Apply the zero-one principle.

18. The proof in Theorem A shows that $\hat{V}_t(n) \geq (n-t)\lceil\log_2(t+1)\rceil + \lceil\log_2 t\rceil$.

19. The network $[1:n][2:n]\ldots[1:3][2:3]$ selects the smallest two elements with $2n-4$ comparators; add [1:2] for $\hat{V}_2(n)$. The lower bound comes from Theorem A (cf. exercise 18).

20. First note that $\hat{V}_3(n) \geq \hat{V}_3(n-1) + 2$ when $n \geq 4$; by symmetry the first comparator may be assumed to be $[1:n]$; after this must come a network to select the third largest of $\langle x_2, x_3, \ldots, x_n\rangle$, and another comparator touching line 1. On the other hand, $\hat{V}_3(5) \leq 7$, since four comparators find the min and max of $\{x_1, x_2, x_3, x_4\}$ and it remains to sort three elements.

21. By induction on the length of α, since $x_i \le y_i$ and $x_j \le y_j$ implies that $x_i \wedge x_j \le y_i \wedge y_j$ and $x_i \vee x_j \le y_i \vee y_j$.

22. By induction on the length of α, since $(x_i \wedge x_j)(y_i \wedge y_j) + (x_i \vee x_j)(y_i \vee y_j) \ge x_i y_i + x_j y_j$. [Consequently $\nu(x \wedge y) \le \nu(x\alpha \wedge y\alpha)$, an observation due to W. Shockley.]

23. Let $x_k = 1$ iff $p_k \ge j$, $y_k = 1$ iff $p_k > j$; then $(x\alpha)_k = 1$ iff $(p\alpha)_k \ge j$, etc.

24. The formula for l'_i is obvious and for l'_j take $z = x \wedge y$ as in the hint and observe that $(z\alpha)_i = (z\alpha)_j = 0$ by exercise 21. Adding additional 1's to z shows the existence of a permutation p with $(p\alpha')_j \le \zeta(z)$, by exercise 23. The relations for u'_i, u'_j follow by reversing the order.

25. Let $f(x) = (x\alpha)_k$; let z be a vector of 0's and 1's with $f(z) = 1$, having minimum $\nu(z)$ over all such vectors. (It follows that $\nu(z) = n + 1 - u_k$.) Let p in P_n be a permutation such that $f(p) = l_k$, and let p correspond to vectors x and y as in exercise 23. The following algorithm transforms x and y in such a way that the corresponding vectors p have $f(p)$ taking all the values from l_k to u_k, inclusive; each step of the algorithm makes x have at least one more component in common with z, while changing $\nu(x)$ by 0 or ± 1. (Note that if $z \not\le x$, there is some r with $z_r > x_r$ and some s with $x_s < z_s$ because of the minimality of $\nu(z)$.)

If the following hold:	Replace (x, y) by:
$z \not\le x$, $z_r > x_r$, $z_s < x_s$, and	
$f(y \vee e^{(r)}) = 0$	$(x \vee e^{(r)}, y \vee e^{(r)})$
$f(y \vee e^{(r)}) = 1$, $f(x \wedge \bar{e}^{(s)}) = 1$	$(x \wedge \bar{e}^{(s)}, y \wedge \bar{e}^{(s)})$
$f(y \vee e^{(r)}) = 1$, $f\big((x \vee e^{(r)}) \wedge \bar{e}^{(s)}\big) = 0$	$(x \vee e^{(r)}, (x \vee e^{(r)}) \wedge \bar{e}^{(s)})$
$f\big((x \vee e^{(r)}) \wedge \bar{e}^{(s)}\big) = 1$, $f\big((y \vee e^{(r)}) \wedge \bar{e}^{(s)}\big) = 0$	$\big((x \vee e^{(r)}) \wedge \bar{e}^{(s)}, (y \vee e^{(r)}) \wedge \bar{e}^{(s)}\big)$
$f\big((y \vee e^{(r)}) \wedge \bar{e}^{(s)}\big) = 1$	$\big((y \vee e^{(r)}) \wedge \bar{e}^{(s)}, y \wedge \bar{e}^{(s)}\big)$
$z \le x$, $z_s < x_s$	$(x \wedge \bar{e}^{(s)}, y \wedge \bar{e}^{(s)})$.

[Is there a simpler solution?]

26. There is a one-to-one correspondence which takes the element $\langle p_1, \ldots, p_n \rangle$ of $P_n\alpha$ into the "covering sequence" $x^{(0)}$ covers $x^{(1)}$ covers $\cdots$ covers $x^{(n)}$, where the $x^{(i)}$ are in $D_n\alpha$; in this correspondence, $x^{(i-1)} = x^{(i)} \vee e^{(j)}$ iff $p_j = i$. For example, $\langle 3, 1, 4, 2 \rangle$ corresponds to the sequence $\langle 1, 1, 1, 1 \rangle$ covers $\langle 1, 0, 1, 1 \rangle$ covers $\langle 1, 0, 1, 0 \rangle$ covers $\langle 0, 0, 1, 0 \rangle$ covers $\langle 0, 0, 0, 0 \rangle$.

27. If x and y denote different columns of a matrix whose rows are sorted, so that $x_i \le y_i$ for all i, and if $x\alpha$ and $y\alpha$ denote the result of sorting the columns, the sta td principle shows that $(x\alpha)_i \le (y\alpha)_i$ for all i, since we can choose i elements of x intehe same rows as any i given elements of y. [This principle was used in the text, to prove the invariance property of Shell's sort, Theorem 5.2.1K. Further exploitation of the idea appears in an interesting paper by David Gale and R. M. Karp, *J. Computer and System Sciences* **6** (1972), 103–115. The fact that column sorting does not mess up sorted rows was apparently first observed in connection with the manipulation of tableaux; cf. Hermann Boerner, *Darstellung von Gruppen* (Springer, 1955), 137.]

28. If $\{x_{i_1}, \ldots, x_{i_t}\}$ are the t largest elements, then $x_{i_1} \wedge \cdots \wedge x_{i_t}$ is the tth largest. If $\{x_{i_1}, \ldots, x_{i_t}\}$ are *not* the t largest, then $x_{i_1} \wedge \cdots \wedge x_{i_t}$ is *less than* the tth largest.

29. $\langle x_1 \wedge y_1, (x_2 \wedge y_1) \vee (x_1 \wedge y_2), (x_3 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_1 \wedge y_3), y_1 \vee (x_3 \wedge y_2) \vee$

$(x_2 \wedge y_3) \vee (x_1 \wedge y_4), y_2 \vee (x_3 \wedge y_3) \vee (x_2 \wedge y_4) \vee (x_1 \wedge y_5), y_3 \vee (x_3 \wedge y_4) \vee (x_2 \wedge y_5) \vee x_1, y_4 \vee (x_3 \wedge y_5) \vee x_2, y_5 \vee x_3\rangle$.

30. Applying the distributive and associative laws reduces any formula to $\vee$'s of $\wedge$'s; then the commutative, idempotent, and absorption laws lead to canonical form. The S_i are precisely those sets S such that the formula is 1 when ($x_j = 1$ iff $j \in S$) while the formula is 0 when ($x_j = 1$ iff $j \in S'$) for any proper subset S' of S.

31. $\delta_4 = 166$. R. Church [*Duke Math. J.* **6** (1940), 732–734] found $\delta_5 = 7579$, M. Ward [*Bull. Amer. Math. Soc.* **52** (1946), 423] found $\delta_6 = 7828352$, and W. F. Lunnon [unpublished, 1969] found $\delta_7 = 2208061288136$; the latter value has not yet been checked independently. No simple formula for δ_n is apparent; D. Kleitman [*Proc. Amer. Math. Soc.* **21** (1969), 677–682] proved that

$$\log_2 \delta_n \sim \binom{n}{\lfloor n/2 \rfloor},$$

using an extremely complicated argument.

32. G_{n+1} is also the set of all strings $\theta\psi$ where θ and ψ are in G_n and $\theta \le \psi$ as vectors of 0's and 1's. It follows that G_n is the set of all strings $z_0 \ldots z_{2^n-1}$ of 0's and 1's, where $z_i \le z_j$ whenever the binary representation of i is "$\le$" the binary representation of j in the 0-1 vector sense. Each element $z_0 \ldots z_{2^n-1}$ of G_n, except $00\ldots0$ and $11\ldots1$, represents a $\wedge$-$\vee$ function $f(x_1, \ldots, x_n)$ from D_n into $\{0, 1\}$, under the correspondence $f(x_1, \ldots, x_n) = z[(x_1 \ldots x_n)_2]$.

33. If such a network existed we would have $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4) = f(x_1 \wedge x_2, x_1 \vee x_2, x_3, x_4)$ or $f(x_1 \wedge x_3, x_2, x_1 \vee x_3, x_4)$ or $\ldots$ or $f(x_1, x_2, x_3 \wedge x_4, x_3 \vee x_4)$ for some function f. The choices $\langle x_1, x_2, x_3, x_4\rangle = \langle x, \bar{x}, 1, 0\rangle$, $\langle x, 0, \bar{x}, 1\rangle$, $\langle x, 1, 0, \bar{x}\rangle$, $\langle 1, x, \bar{x}, 0\rangle$, $\langle 1, x, 0, \bar{x}\rangle$, $\langle 0, 1, x, \bar{x}\rangle$ show that no such f exists.

34. Yes; after proving this, you are ready to tackle the $n = 16$ network in Fig. 49.

35. Otherwise the permutation in which only i and $i+1$ are misplaced would never be sorted.

36. (a) Each adjacent comparator reduces the number of inversions by 0 or 1, and $\langle n, n-1, \ldots, 1\rangle$ has $\binom{n}{2}$ inversions. (b) Let $\alpha = \beta[p{:}p+1]$, and argue by induction on the length of α. If $p = i$, then $j > p+1$, and $(x\beta)_p > (x\beta)_j$, $(x\beta)_{p+1} > (x\beta)_j$; hence $(y\beta)_p > (y\beta)_j$ and $(y\beta)_{p+1} > (y\beta)_j$. If $p = i-1$, then either $(x\beta)_p$ or $(x\beta)_{p+1}$ is $> (x\beta)_j$; hence either $(y\beta)_p$ or $(y\beta)_{p+1}$ is $> (y\beta)_j$. If $p = j-1$ or j, the arguments are similar. For other p the argument is trivial. [An amusing consequence of this exercise is that if α is sorting network consisting of $\binom{n}{2}$ adjacent comparators, so is α^R (the comparators in reverse order).]

37. It suffices to show that if each comparator is replaced by an *interchange* operation we obtain a "reflection network" which transforms $\langle x_1, \ldots, x_n\rangle$ into $\langle x_n, \ldots, x_1\rangle$. But in this interpretation it is not difficult to trace the route of x_k. (Note that the permutation $\pi = (1, 2)(3, 4)\ldots(2n-1, 2n)(2, 3)(4, 5)\ldots(2n-2, 2n-1) = (1, 3, 5, \ldots, 2n-1, 2n, 2n-2, \ldots, 2)$ satisfies $\pi^{n/2} = (1, 2n)(2, 2n-1)\ldots(n-1, n)$.) The odd-even transposition sort was mentioned briefly by H. Seward in 1954; it has been discussed by A. Grasselli [*IRE Trans.* **EC-11** (1962), 483] and by Kautz et al. [*IEEE Trans.* **C-17** (1968), 443–451].

38. Let u be the smallest element of $(x\alpha)_j$, and let $y^{(0)}$ be any vector in D_n such that

$(y^{(0)})_k = 0$ implies $(x\alpha)_k$ contains an element $\leq u$, $(y^{(0)})_k = 1$ implies $(x\alpha)_k$ contains an element $> u$. If $\alpha = \beta[p:q]$, it is possible to find a vector $y^{(1)}$ satisfying the same conditions but with α replaced by β, and such that $y^{(1)}[p:q] = y^{(0)}$. Starting with $(y^{(0)})_i = 1$, $(y^{(0)})_j = 0$, we eventually have a vector $y = y^{(r)}$ satisfying the desired condition.

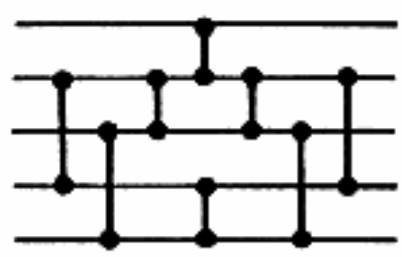
39. Both $(x \veebar y) \veebar z$ and $x \veebar (y \veebar z)$ represent the largest m elements of the multiset $x \uplus y \uplus z$; $(x \wedge y) \wedge z$ and $x \wedge (y \wedge z)$ represent the smallest m. If $x = y = z = \{0, 1\}$, $(x \wedge z) \veebar (y \wedge z) = (x \wedge y) \veebar (x \wedge z) \veebar (y \wedge z) = \{0, 0\}$ while (middle elements of $\{0, 0, 0, 1, 1, 1\}$) $= (x \veebar y) \wedge z = \{0, 1\}$. Sorting networks for three elements and the result of exercise 38 imply that the middle elements of $x \uplus y \uplus z$ may be expressed either as $((x \veebar y) \wedge z) \veebar (x \wedge y)$ or $((x \wedge y) \veebar z) \wedge (x \veebar y)$ or any other formula obtained by permuting x, y, z in these expressions. (There seems to be no "symmetrical" formula for the middle elements.)

40. Let $\alpha' = \alpha[i:j]$, and let k be an index $\neq i, j$. If $(x\alpha)_i \leq (x\alpha)_k$ for all x, then $(x\alpha')_i \leq (x\alpha')_k$; if $(x\alpha)_k \leq (x\alpha)_i$ and $(x\alpha)_k \leq (x\alpha)_j$ for all x, the same holds when α is replaced by α'; if $(x\alpha)_k \leq (x\alpha)_i$ for all x, then $(x\alpha')_k \leq (x\alpha')_j$. In this way we see that α' has at least as many known relations as α, plus one more if $[i:j]$ isn't redundant. [*Bell System Tech. J.* **49** (1970), 1627–1644.]

41. (a) Let $y_{i_s} = x_{j_s}$, $y_{j_s} = x_{i_s}$, $y_k = x_k$ for $i_s \neq k \neq j_s$; then $y\alpha^s = x\alpha$. (b) This is obvious unless the set $\{i_s, j_s, i_t, j_t\}$ has only three distinct elements; suppose that $i_s = i_t$. Then if $s < t$ the first $s - 1$ comparators have (i_s, j_s, j_t) replaced, respectively, by (j_s, j_t, i_s) in both $(\alpha^s)^t$ and $(\alpha^t)^s$. (c) $(\alpha^s)^s = \alpha$, and $\alpha^1 = \alpha$, so we can assume that $s_1 > s_2 > \cdots > s_k > 1$. (d) Let $\beta = \alpha[i:j]$; then $g_\beta(x_1, \ldots, x_n) = (\bar{x}_i \vee x_j) \wedge (g_\alpha(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) \vee g_\alpha(x_1, \ldots, x_j, \ldots, x_i, \ldots, x_n))$. Iterating this identity yields the result. (e) $f_\alpha(x) = 1$ iff no path in G_α goes from i to j where $x_i > x_j$. If α is a sorting network, the conjugates of α are also; and $f_\alpha(x) = 0$ for all x with $x_i > x_{i+1}$. Take $x = e^{(i)}$; this shows that G has an arc from i to k_1 for some $k_1 \neq i$. If $k_1 \neq i + 1$, $x = e^{(i)} \vee e^{(k_1)}$ shows that G has an arc from i or k_1 to k_2 for some $k_2 \notin \{i, k_1\}$. If $k_2 \neq i + 1$, continue in the same way until finding a path in G from i to $i + 1$. Conversely if α is not a sorting network, let x be a vector with $x_i > x_{i+1}$ and $g_\alpha(x) = 1$. Some conjugate α' has $f_{\alpha'}(x) = 1$, so $G_{\alpha'}$ can have no path from i to $i + 1$. [In general, $(x\alpha)_i \leq (x\alpha)_j$ for all x iff $G_{\alpha'}$ has an oriented path from i to j for all α' conjugate to α.]

42. There must exist a path of length $\lceil \log_2 n \rceil$ or more, from some input to the largest output (consider m_n in Theorem A); when that input is set to ∞, the comparators on this path have a predetermined behavior, and the remaining network must be an $(n - 1)$-sorter. [*IEEE Trans. on Computers* **C–21** (1972), 612–613.]

43.



[If $P(n)$ denotes the minimum number of switches needed in a permutation network, it is clear that $P(n) \geq \lceil \log_2 n! \rceil$. A. Waksman and M. Green have proved that $P(n) \leq B(n)$ for all n, where $B(n)$ is the binary insertion function of Eq. 5.3.1–3; cf. *IEEE Trans.* **C-17** (1968), 447; Green also has proved that $P(5) = 8$.]

52. If $h[k + 1] = h[k] + 1$ and the file is not in order, something must happen to it

on the next pass; this decreases the number of inversions, by exercise 5.2.2–1, hence the file will eventually become sorted. But if $h[k+1] \geq h[k]+2$ for $1 \leq k < m$, the smallest key will never move into its proper place if it is initially in R_2.

53. We use the hint, and also regard $K_{N+1} = K_{N+2} = \cdots = 1$. If $K_{h[1]+j} = \cdots = K_{h[m]+j} = 1$ at step j, and if $K_i = 0$ for some $i > h[1]+j$, we must have $i < h[m]+j$ since there are less than n 1's. Suppose k and i are minimal such that $h[k]+j < i < h[k+1]+j$ and $K_i = 0$. Let $s = h[k+1]+j-i$; we have $s < h[k+1]-h[k] \leq k$. At step $j-s$, at least $k+1$ 0's must have been under the heads, since $K_i = K_{h[k+1]+j-s}$ was set to zero at that step; s steps later, there are at least $k+1-s \geq 2$ 0's remaining between $K_{h[1]+j}$ and K_i, inclusive, contradicting the minimality of i.

The second pass gets the next $n-1$ elements into place, etc. If we start with the permutation $N\ (N-1)\ \ldots\ 2\ 1$, the first pass changes it to $(N+1-n)\ (N-n)\ \ldots\ 1\ (N+2-n)\ \ldots\ (N-1)\ N$, since $K_{h[1]+j} \leftarrow K_{h[m]+j}$ whenever $1 \leq h[1]+j$ and $h[m]+j \leq N$; therefore the bound is best possible.

54. Suppose that $h[k+1]-s > h[k]$ and $h[k] \leq s$; the smallest key ends in position R_i for $i > 1$ if it starts in R_{n-s}. Therefore $h[k+1] \leq 2h[k]$ is necessary; it is also sufficient, by the special case $t = 0$ of the

Theorem. *If $n = N$ and if $K_1 \ldots K_N$ is a permutation of $\{1, 2, \ldots, n\}$, a single sorting pass will set $K_i = i$ for $1 \leq i \leq t+1$, if $h[k+1] \leq h[k]+h[k-i]+i$ for $1 \leq k < m$ and $0 \leq i \leq t$.* (By convention, let $h[k] = k$ when $k \leq 0$.)

Proof. By induction on t; if step t does not find the key $t+1$ under the heads, we may assume that it appears in position $R_{h[k+1]+t-s}$ for some $s > 0$, where $h[k+1]-s < h[k]$; hence $h[k-t]+t-s > 0$. But this is impossible if we consider step $t-s$, which presumably placed the element $t+1$ into position $R_{h[k+1]+t-s}$ although there were at least $t+1$ lower heads active. ∎

(The condition is necessary for $t = 0, 1$, but not for $t = 2$.)

55. If the numbers $\{1, \ldots, 23\}$ are being sorted, the theorem in the previous exercise shows that $\{1, 2, 3, 4\}$ find their true destination. When 0's and 1's are being sorted it is possible to verify that it is impossible to have all heads reading 0 while all positions not under the heads contain 1's, at steps -2, -1, and 0; hence the proof in the previous exercise can be extended to show that $\{5, 6, 7\}$ find their true destination. Finally $\{8, \ldots, 23\}$ must be sorted, by the argument in exercise 53.

57. When $r \leq m-2$, the heads take the string $0^p1^101^301^70 \ldots 01^{2^r-1}01^q$ into $0^{p+1}1^101^301^70 \ldots 01^{2^{r-1}-1}01^{2^{r-1}+q}$; hence $m-2$ passes are necessary. [When the heads are at positions $1, 2, 3, 5, \ldots, 1+2^{m-2}$, Pratt has discovered a similar result: the string $0^p1^a01^{2^b-1}01^{2^{b+1}-1}0 \ldots 01^{2^r-1}01^q$, $1 \leq a \leq 2^{b-1}$, goes into $0^{p+1}1^{a-1}01^{2^b-1}01^{2^{b+1}-1}0 \ldots 01^{2^{r-1}-1}01^{2^r+q}$, hence at least $m - \lceil \log_2 m \rceil - 1$ passes are necessary in the worst case for this sequence of heads. The latter head sequence is of special interest since it has been used as the basis of a very ingenious sorting device invented by P. N. Armstrong [cf. *U.S. Patent 3399383*]. Pratt conjectures that these input sequences provide the true worst case, over all inputs.]

58. During quicksort, each key $K_2, \ldots, K_N$ is compared with K_1; let $A = \{i | K_i < K_1\}$, $B = \{j | K_j > K_1\}$. Subsequent operations quicksort A and B independently; all comparisons $K_i:K_j$ for i in A and j in B are suppressed, by both quicksort and the

restricted uniform algorithm, and no other comparisons are suppressed by the unrestricted uniform algorithm.

In this case we could restrict the algorithm even further, omitting Cases 1 and 2 so that arcs are added to G only when comparisons are explicitly made, yet considering only paths of length 2 when testing for redundancy. Another way to solve this problem is to consider the equivalent tree insertion sorting algorithm of Section 6.2.2, which makes precisely the same comparisons as the uniform algorithm in the same order.

59. (a) The probability that K_{a_i} is compared with K_{b_i} is the probability that c_i other specified keys do not lie between K_{a_i} and K_{b_i}; this is the probability that two numbers chosen at random from $\{1, 2, \ldots, c_i + 2\}$ are consecutive, namely

$$(c_i + 1) \Big/ \binom{c_i + 2}{2}.$$

(b) The first $n - 1$ values of c_i are zero, then come $(n - 2)$ 1's, $(n - 3)$ 2's, etc.; hence the average is $2\sum_{1\le k\le n} (n-k)/(k+1) = 2\sum_{1\le k\le n} \big((n+1)/(k+1) - 1\big) = 2(n+1)(H_{n+1} - 1) - 2n$. (c) The "bipartite" nature of merging shows that the restricted uniform algorithm is the same as the uniform algorithm for this sequence. The pairs involving vertex N have c's equal to $0, 1, \ldots, N - 2$, respectively; so the average number of comparisons is exactly the same as quicksort.

60. No; when $N = 5$ no pair sequence ending in (1, 5)(1, 2)(2, 3)(3, 4)(4, 5) will require 10 comparisons. [An interesting research problem: For all N, find a (restricted) uniform sorting method whose worst case is as good as possible.]

62. An item can lose at most one inversion per pass, so the minimum number of passes is at least the maximum number of inversions of any item in the input permutation. The bubble sort strategy achieves this bound, since each pass decreases the inversion count of every inverted item by one (cf. 5.2.2–1). An additional pass may be needed to determine whether or not sorting is complete, but the wording of this exercise allows us to overlook such considerations.

It is perhaps unfortunate that the first result in the study of computational complexity via automata established the "optimality" of a sorting method which is so poor from a programming standpoint! The situation is analogous to the history of random number generation, which took several backward steps when generators that are "optimum" from one particular point of view were recommended for general use. The moral is that optimality results are often heavily dependent on the abstract model; although the results are very interesting, they must be applied wisely in practice.

[Demuth went on to consider a generalization to an r-register machine (saving a factor of r), and to a Turing-like machine in which the direction of scan could oscillate between left-right and right-left at will. He observed that the latter type of machine can do the straight insertion and the cocktail shaker sorts; but any such 1-register machine must go through at least $\frac{1}{4}N^2$ steps on the average, since each step reduces the total number of inversions by at most one. Finally he considered r-register random-access machines and the question of minimum-comparison sorting.]

SECTION 5.4

1. We could omit the internal sorting phase, but that would generally be much

slower since it would increase the number of times each piece of data is read and written on the external memory.

2. The runs are distributed as in (1), then Tape 3 is set to $R_1 \ldots R_{2000}$; $R_{2001} \ldots R_{4000}$; $R_{4001} \ldots R_{5000}$. After all tapes are rewound, a "one-way merge" sets T_1 and T_2 to the respective contents of T_3 and T_4 in (2). Then T_1 and T_2 are merged to T_3, and the information is copied back and merged once again, for a total of five passes. In general, the procedure is like the four-tape balanced merge, but with copy passes between each of the merge passes, so one less than twice as many passes are performed.

3. (a) $\lceil \log_P S \rceil$. (b) $\log_B S$, where $B = \sqrt{P(T-P)}$ is called the "effective power of the merge." When $T = 2P$ the effective power is P; when $T = 2P - 1$ the effective power is $\sqrt{P(P-1)} = P - \frac{1}{2} - \frac{1}{8}P^{-1} + O(P^{-2})$, slightly less than $\frac{1}{2}T$.

4. $\frac{1}{2}T$. If T is odd and P must be an integer, both $\lceil T/2 \rceil$ and $\lfloor T/2 \rfloor$ give the same maximum value. It is best to have $P \geq T - P$, according to exercise 3, so we should choose $P = \lceil T/2 \rceil$ for balanced merging.

SECTION 5.4.1

1.

$$087\ 154\ 170\ 426\begin{cases}503\begin{cases}503\ \infty\\908\ \infty\end{cases}\\426\begin{cases}426\ 653\ \infty\\612\ \infty\end{cases}\end{cases}$$

2. The path (061) — (154) — (087) — (512) — [061] would be changed to (087) — (154) — (512) — (612) — [612]. (We are essentially doing a "bubble sort" from bottom to top along the path!)

3. `and fourscore our seven years/ ago brought fathers forth on this/ a conceived continent in liberty nation new the to/ and dedicated men proposition that/ all are created equal.`

4. (The problem is slightly ambiguous; in this interpretation we do not clear the internal memory until the reservoir is about to overflow.)

`and fourscore on our seven this years/ ago brought continent fathers forth in liberty nation new to/ a and conceived dedicated men proposition that the/ all are created equal.`

5. False, the complete binary tree with P external nodes is defined for all $P \geq 1$.

6. Insert "If T = LOC(X[0]) then go to R2, otherwise" at the beginning of step R6, and delete the similar clause from step R7.

7. There is no output, and RMAX stays equal to 0.

8. If the first actual key were ∞ the record would be lost. To avoid ∞, we could set a switch, initially avoiding the test involving LASTKEY in step R4. Then when RQ $\neq$ 0 in step R3 for the first time, the switch is changed so that R4 tests LASTKEY and R3 no longer tests RQ.

9. Assume, for example, that the current run is ascending, while the next should be

descending. Then the steps of Algorithm R will work properly except for one change: In step R6 if RN(T) = RQ > RC, reverse the test on KEY(LOSER(T)) vs. KEY(Q).

When RC changes, the key tests of steps R4 and R6 should change appropriately.

10. Let $\cdot j \equiv \mathtt{LOC}(\mathtt{X}[j])$. The mechanism of Algorithm R ensures that the following conditions are true whenever we reach step R3, if we set first $\mathtt{LOSER}(\cdot 0) \leftarrow \mathtt{Q}$ and $\mathtt{RN}(\cdot 0) \leftarrow \mathtt{RQ}$: The values of $\mathtt{LOSER}(\cdot 0), \ldots, \mathtt{LOSER}\big(\cdot(P-1)\big)$ are a permutation of $\{\cdot 0, \cdot 1, \ldots, \cdot(P-1)\}$; and there exists a permutation of the pointers $\{\mathtt{LOSER}(\cdot j) \mid \mathtt{RN}(\cdot j) = 0\}$ which corresponds to an actual tournament. In other words when $\mathtt{RN}(\cdot j) = 0$, the value of $\mathtt{KEY}\big(\mathtt{LOSER}(\cdot j)\big)$, and therefore also of $\mathtt{LOSER}(\cdot j)$, is irrelevant; we may permute such "Losers" among themselves. After P steps all $\mathtt{RN}(\cdot j)$ will be nonzero, so the entire tree will be consistent. (The answer to the hint is "yes.")

11. True. (Both keys are from the same subsequence in the proof of Theorem K.)

13. The keys left in memory when the first run has ended tend to be smaller than average, since they didn't make it into the first run. Thus the second run can output more of the smaller keys.

14. Assume that the snow suddenly stops when the snowplow is at a random point u, $0 \le u < 1$, after it has reached its "steady state." Then the second-last run contains $(1 + 2u - u^2)P$ records, and the last run contains u^2P. Integrating this times du yields an average time of $(2 - \frac{1}{3})P$ records in the penultimate run, $\frac{1}{3}P$ in the last.

15. False, the last run can be arbitrarily long; but only in the comparatively rare circumstance that all records in memory belong to the same run when the input is exhausted.

16. Iff each element has less than P inversions. (Cf. Sections 5.1.1, 5.4.8.) The probability is 1 when $N \le P$, $P^{N-P}P!/N!$ when $N \ge P$, by considering inversion tables. (In actual practice, however, a one-pass sort is not too uncommon, since people tend to sort a file even when they suspect it might be in order, as a precautionary measure.)

17. Exactly $\lceil N/P \rceil$ runs, all but the last of length P. (The "worst case.")

18. Nothing changes on the second pass, since it is possible to show that the kth record of a run is less than at least $P + 1 - k$ records of the preceding run, for $1 \le k \le P$. (However, there seems to be no simple way to characterize the result of P-way replacement selection followed by P'-way replacement selection when $P' > P$.)

19. Argue as in the derivation of (2) that $h(x, t)\,dx = KL\,dt$, where this time $h(x, t) = I + Kt$ for all x, and $P = IL$. This implies $x(t) = L \ln\big((I + Kt)/I\big)$, so that when $x(T) = L$ we have $KT = (e - 1)I$. The amount of snowfall since $t = 0$ is therefore $(e - 1)IL = (e - 1)P$.

20. As in exercise 19, we have $(I + Kt)\,dx = K(L - x)\,dt$; hence $x(t) = LKt/(I + Kt)$. The reservoir contents is $IL = P = P' = \int_0^T x(t)K\,dt = L\big(KT - I \ln\big((I + KT)/I\big)\big)$, hence $KT = \alpha I$, where $\alpha \approx 2.1461$ is the root of $1 + \alpha = e^{\alpha - 1}$. The run length is the total amount of snowfall during $0 \le t \le T$, namely $KLT = \alpha P$.

21. Proceed as in the text, but after each run wait for $P - P'$ snowflakes to fall before the plow starts out again. This means that $h\big(x(t), t\big)$ is now KT_1, instead of KT, where $T_1 - T$ is the amount of time taken by the extra snowfall. The run length is LKT_1, $x(t) = L(1 - e^{-t/T_1})$, $P = LKT_1e^{-T/T_1}$, and $P' = \int_0^T x(t)K\,dt = P + LK(T - T_1)$. In other words, a run length of $e^\theta P$ is obtained when $P' = \big(1 - (1 - \theta)e^\theta\big)P$, for $0 \le \theta \le 1$.

22. For $0 \le t \le (\kappa - 1)T$, $dx \cdot h = K\,dt\big(x(t+T) - x(t)\big)$, and for $(\kappa - 1)T \le t \le T$, $dx \cdot h = K\,dt\big(L - x(t)\big)$, where h is seen to be constantly equal to KT at the position of the plows. It follows that for $0 \le j \le k$, $0 \le u \le 1$, and $t = (\kappa - j - u)T$, we have $x(t) = L\big(1 - e^{u-\theta}F_j(u)/F(\kappa)\big)$. The run length is KTL, the amount of snowfall between the times that consecutive snowplows leave point 0 in the steady state; P is the amount cleared during each snowplow's last burst of speed, namely $KT\big(L - x(\kappa T)\big) = KTLe^{-\theta}/F(\kappa)$; and $P' = \int_0^{\kappa T} x(t)K\,dt$ can be shown to have the stated form.

[*Notes:* It turns out that the stated formulas are valid also for $k = 0$. When $k \ge 1$ the number of elements per run which go into the reservoir *twice* is $P'' = \int_0^{(\kappa-1)T} x(t)K\,dt$, and it is easy to show that (run length) $- P' + P'' = (e-1)P$, a phenomenon noticed by Frazer and Wong. Is it a coincidence that the generating function for $F_k(\theta)$ is so similar to that in exercise 5.1.3–11?]

23. Let $P = pP'$ and $q = 1 - p$. For the first T_1 units of time the snowfall comes from the qP' elements remaining in the reservoir after the first pP' have been initially removed in random order; and when the old reservoir is empty, uniform snow begins to fall again. We choose T_1 so that $KLT_1 = qP'$. For $0 \le t \le T_1$, $h(x, t) = (p + qt/T_1)g(x)$, where $g(x)$ is the height of snow put into the reservoir from position x; for $T_1 \le t \le T$, $h(x, t) = g(x) + (t - T_1)K$. For $0 \le t \le T_1$, $g\big(x(t)\big)$ is $\big(q(T_1 - t)/T_1\big)g\big(x(t)\big) + (T - T_1)K$; and for $T_1 \le t \le T$, $g\big(x(t)\big) = (T - t)K$. Hence $h\big(x(t), t\big) = (T - T_1)K$ for $0 \le t \le T$, and $x(t) = L\big(1 - \exp\big(-t/(T - T_1)\big)\big)$. The total run length is $(T - T_1)KL$; the total amount "recycled" from the reservoir back again (cf. exercise 22) is T_1KL; and the total amount cleared after time T is $P = KT\big(L - x(T)\big)$.

So the assumptions of this exercise give runs of length $(e^s/s)P$ when the reservoir size is $\big(1 + (s-1)e^s/s\big)P$. This is considerably worse than the results of exercise 22, since the reservoir contents are being used in a more advantageous order in that case.

(The fact that $h\big(x(t), t\big)$ is constant in so many of these problems is not surprising, since it is equivalent to saying that the elements of each run obtained during a steady state of the system are uniformly distributed.)

24. (a) Essentially the same proof works, each of the subsequences has runs in the same direction as the output runs. (b) The stated probability is the probability that the run has length $n + 1$ and is followed by y; it equals $(1 - x)^n/n!$ when $x > y$, and it is $(1 - x)^n/n! - (y - x)^n/n!$ when $x \le y$. (c) Induction. For example if the nth run is ascending, the $(n - 1)$st was descending with probability p, so the first integral applies. (d) We find that $f'(x) = f(x) - c - pf(1 - x) - qf(x)$, then $f''(x) = -2pc$, which ultimately leads to $f(x) = c(1 - qx - px^2)$, $c = 6/(3 + p)$. (e) If $p > eq$ then $pe^x + qe^{1-x}$ is monotone increasing for $0 \le x \le 1$, and $\int_0^1 |pe^x + qe^{1-x} - e^{1/2}|\,dx = (p - q)(e^{1/2} - 1)^2 < 0.43$. If $q \le p < eq$ then $pe^x + qe^{1-x}$ lies between $2\sqrt{pqe}$ and $p + qe$, so $\int_0^1 |pe^x + qe^{1-x} - \frac{1}{2}(p + qe + 2\sqrt{pqe})|\,dx \le \frac{1}{2}(\sqrt{p} - \sqrt{qe})^2 < 0.4$; and if $p < q$ we may use a symmetrical argument. Thus for all p and q there is a constant C such that $\int_0^1 |pe^x + qe^{1-x} - C|\,dx < 0.43$. Let $\delta_n(x) = f_n(x) - f(x)$. Then $\delta_{n+1}(y) = (1 - e^{y-1})\int_0^1 (pe^x + qe^{1-x} - C)\delta_n(x)\,dx + p\int_0^{1-y} e^{y-1+x}\delta_n(x)\,dx + q\int_y^1 e^{y-x}\delta_n(x)\,dx$; hence if $\delta_n(y) \le \alpha_n$, $|\delta_{n+1}(y)| \le (1 - e^{y-1})(1.43)\alpha_n < 0.91\alpha_n$. (f) For all $n \ge 0$, $(1 - x)^n/n!$ is the probability that the run length is $> n$. (g) $\int_0^1 (pe^x + qe^{1-x})f(x)\,dx = 6/(3 + p)$.

26. (a) Consider the number of permutations with $n + r + 1$ elements and n left-to-right minima, where the rightmost element is not the smallest. (b) Use the fact that

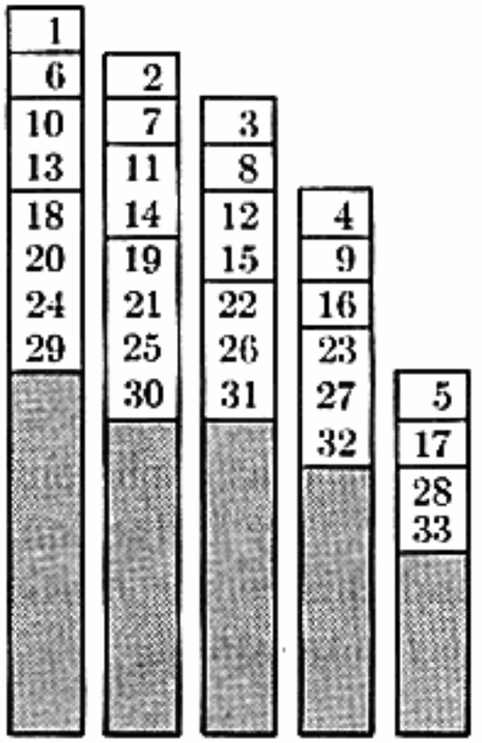
$$\sum_{1\le k<n} \begin{bmatrix} k \\ k-r \end{bmatrix} k = \begin{bmatrix} n \\ n-r-1 \end{bmatrix},$$

by the definition of Stirling numbers in the Index to Notations. (c) Add $r+1$ to the mean, using the fact that $\sum_{n\ge 0} \left[{n+r \atop n}\right](n+r)/(n+r+1)! = 1$, to get $\sum_{n\ge 0} \left[{n+r \atop n}\right]/(n+r-1)!$.

27. For multiway merging there is comparatively little problem, since P stays constant and records are processed sequentially on each file; but when forming initial runs, we would like to vary the number of records in memory depending on their lengths. We could keep a heap of as many records as will fit in memory, using dynamic storage allocation as described in Section 2.5. M. A. Goetz [*Proc. AFIPS Joint Computer Conf.* **25** (1964), 602–604] has suggested another approach, breaking each record into fixed-size parts which are linked together; they occupy space at the leaves of the tree, but only the leading part participates in the tournament.

SECTION 5.4.2

1.



2. After the first merge phase, all remaining dummies are on tape T, and there are at most $a_n - a_{n-1} \le a_{n-1}$ of them. Therefore they all disappear during the second merge phase.

3. We have $(\mathtt{D}[1], \mathtt{D}[2], \ldots, \mathtt{D}[T]) = (a_n - a_{n-P}, a_n - a_{n-P+1}, \ldots, a_n - a_n)$, so the condition follows from the fact that the a's are nondecreasing. The condition is important to the validity of the algorithm, since steps D2 and D3 never decrease $\mathtt{D}[j+1]$ more often than $\mathtt{D}[j]$.

4. $(1 - z - \cdots - z^5)a(z) = 1$ because of (3). Also, $t(z) = \sum_{n\ge 1} (a_n + b_n + c_n + d_n + e_n)z^n = (z + \cdots + z^5)a(z) + (z + \cdots + z^4)a(z) + \cdots + za(z) = (5z + 4z^2 + 3z^3 + 2z^4 + z^5)a(z)$. [Cf. (16).]

5. Let $g_p(z) = (z-1)f_p(z) = z^{p+1} - 2z^p + 1$, and let $h_p(z) = z^{p+1} - 2z^p$. Let $\phi^{-1} > \epsilon > 0$; Rouche's theorem tells us that $h_p(z)$ and $g_p(z)$ have the same number of roots inside the circle $|z| = 1 + \epsilon$, provided $|h_p(z)| > |h_p(z) - g_p(z)| = 1$ on the circle. We have $|h_p(z)| \ge (1+\epsilon)^p(1-\epsilon) > (1+\phi^{-1})^2(1-\phi^{-1}) = 1$. Hence g_p has p roots of magnitude ≤ 1. They are distinct, since $\gcd\big(g_p(z), g_p'(z)\big) = \gcd\big(g_p(z), (p+1)z - 2p\big) = 1$. [*AMM* **67** (1960), 745–752.]

6. Let $c_0 = -\alpha p(\alpha^{-1})/q'(\alpha^{-1})$. Then $p(z)/q(z) - c_0/(1 - \alpha z)$ is analytic in $|z| \le R$ for some $R > |\alpha|^{-1}$; hence the coefficient of z^n in $p(z)/q(z)$ is $c_0\alpha^n + O(R^{-n})$. Thus, $\ln S = n \ln \alpha + \ln c_0 + O((\alpha R)^{-n})$; and $n = (\ln S/\ln \alpha) + O(1)$ implies that $O((\alpha R)^{-n}) = O(S^{-\epsilon})$. Similarly, let $c_1 = \alpha^2 p(\alpha^{-1})/q'(\alpha^{-1})^2$, $c_2 = -\alpha p'(\alpha^{-1})/q'(\alpha^{-1})^2 + \alpha p(\alpha^{-1})q''(\alpha^{-1})/q'(\alpha^{-1})^3$, and consider $p(z)/q(z)^2 - c_1/(1 - \alpha z)^2 - c_2/(1 - \alpha z)$.

7. Let $\alpha_p = 2(1 - t)$, and $z = 2^{-p-1}$. Then $z = t(1 - t)^p$, and reversion of this series (Algorithm 4.7*R*) gives a power series for t in terms of z, yielding the asymptotic expansion $\alpha_p = 2 - 2^{-p} - p2^{-2p-1} + O(p^2 2^{-3p})$.

Note: It follows that the quantity ρ in exercise 6 becomes approximately $\log_4 S$ as p increases. Similarly, for both Table 5 and Table 6, the coefficient c approaches $1/((\phi + 2) \ln \phi)$ on a large number of tapes.

8. Evidently $N_0^{(p)} = 1$, $N_m^{(p)} = 0$ for $m < 0$, and by considering the different possibilities for the first summand we have $N_m^{(p)} = N_{m-1}^{(p)} + \cdots + N_{m-p}^{(p)}$ when $m > 0$. Hence $N_m^{(p)} = F_{m+p-1}^{(p)}$. [*Lehrbuch der Combinatorik* (Leipzig: Teubner, 1901), 136–137.]

9. Consider the position of the leftmost 0, if there is one; we find $K_m^{(p)} = F_{m+p}^{(p)}$. *Note:* There is a simple one-to-one correspondence between such sequences of 0's and 1's and the representations of $m + 1$ considered in exercise 8: Place a 0 at the right end of the sequence, and look at the positions of all the 0's.

10. *Lemma:* If $n = F_{j_1}^{(p)} + \cdots + F_{j_m}^{(p)}$ is such a representation, with $j_1 > \cdots > j_m \ge p$, we have $n < F_{j_1+1}^{(p)}$. *Proof:* The result is obvious if $m < p$; otherwise let k be minimal with $j_k > j_{k+1} + 1$; we have $k < p$, and by induction $F_{j_{k+1}}^{(p)} + \cdots + F_{j_m}^{(p)} < F_{j_k-1}^{(p)}$, hence $n < F_{j_1}^{(p)} + \cdots + F_{j_1-k-1}^{(p)} \le F_{j_1+1}^{(p)}$.

The stated result can now be proved, by induction on n. If $n > 0$ let j be maximal such that $F_j^{(p)} \le n$. The lemma shows that each representation of n must consist of $F_j^{(p)}$ plus a representation of $n - F_j^{(p)}$. By induction, $n - F_j^{(p)}$ has a unique representation of the desired form, and this representation does not include all of the numbers $F_{j-1}^{(p)}, \ldots, F_{j-p+1}^{(p)}$ because j is maximal.

Notes: The case $p = 2$, which is due to E. Zeckendorf [cf. *Simon Stevin* **29** (1952), 190–195], has been considered in exercise 1.2.8–34. There is a simple algorithm to go from the representation of n to that of $n + 1$, working on the sequence $c_j \ldots c_1 c_0$ of 0's and 1's such that $n = \sum c_j F_{j+p}^{(p)}$: For example, if $p = 3$, we look at the rightmost digits, changing $\ldots 0$ to $\ldots 1$, $\ldots 01$ to $\ldots 10$, $\ldots 011$ to $\ldots 100$; then we "carry" to the left if necessary, replacing "$\ldots 0111 \ldots$" by "$\ldots 1000 \ldots$". See the sequences of 0's and 1's in exercise 9, in the order listed. A similar number system has been studied by W. C. Lynch [*Fibonacci Quarterly* **8** (1970), 6–22], who found a very interesting way to make it govern both the distribution and merge phases of a polyphase sort.

12. The kth power contains the perfect distributions for levels $k - 4$ through k, on successive rows, with the largest elements to the right.

13. By induction on the level.

14. (a) $n(1) = 1$, so assume that $k > 1$. The law $T_{nk} = T_{n-1,k-1} + \cdots + T_{n-P,k-1}$ shows that $T_{nk} \le T_{n+1,k}$ iff $T_{n-P,k-1} \le T_{n,k-1}$. Let r be any positive integer, and let n' be minimal such that $T_{n'-r,k-1} > T_{n',k-1}$; then $T_{n-r,k-1} \ge T_{n,k-1}$ for all $n \ge n'$, since it is trivial for $n \ge n(k - 1) + r$ and otherwise $T_{n-r,k-1} \ge T_{n'-r,k-1} \ge T_{n',k-1} \ge T_{n,k-1}$. (b) The same argument with $r = n - n'$ shows that $T_{n'k'} < T_{nk'}$

implies $T_{n'-j,k'} \le T_{n-j,k'}$ for all $j \ge 0$; hence the recurrence implies that $T_{n'-j,k} \le T_{n-j,k}$ for all $j \ge 0$ and $k \ge k'$. (c) Let $\ell(S)$ be the least n such that $\Sigma_n(S)$ assumes its minimum value. The sequence M_n exists as desired iff $\ell(S) \le \ell(S+1)$ for all S. Suppose $n = \ell(S) > \ell(S+1) = n'$, so that $\Sigma_n(S) < \Sigma_{n'}(S)$ and $\Sigma_n(S+1) \ge \Sigma_{n'}(S+1)$. There is some smallest S' such that $\Sigma_n(S') < \Sigma_{n'}(S')$, and we have $m = \Sigma_n(S') - \Sigma_n(S'-1) < \Sigma_{n'}(S') - \Sigma_{n'}(S'-1) = m'$. Then $\sum_{1\le k\le m} T_{n'k} < S' \le \sum_{1\le k\le m} T_{nk}$; hence there is some $k' \le m$ such that $T_{n'k'} < T_{nk'}$. Similarly we have $l = \Sigma_n(S+1) - \Sigma_n(S) > \Sigma_{n'}(S+1) - \Sigma_{n'}(S) = l'$; hence $\sum_{1\le k\le l'} T_{n'k} \ge S+1 > \sum_{1\le k\le l'} T_{nk}$. Since $l' \ge m' > m$, there is some $k > m$ such that $T_{n'k} > T_{nk}$. But this contradicts part (b).

15. This theorem has been proved by D. A. Zave, whose article was cited in the text.

16. D. A. Zave has shown that the number of records input (and output) is $S \log_{T-1} S + O(S\sqrt{\log S})$.

17. Let $T = 3$; $A_{11}(x) = 6x^6 + 35x^7 + 56x^8 + \cdots$, $B_{11}(x) = x^6 + 15x^7 + 35x^8 + \cdots$, $T_{11}(x) = 7x^6 + 50x^7 + 91x^8 + 64x^9 + 19x^{10} + 2x^{11}$. The optimum distribution for $S = 144$ requires 55 runs on T2, and this forces a nonoptimum distribution for $S = 145$. D. A. Zave has studied near-optimum procedures of this kind.

18. Let $S = 9$, $T = 3$, and consider the following two patterns.

Optimum Polyphase:

T1	T2	T3	Cost
$0^2 1^6$	$0^2 1^3$	—	
1^3	—	$0^2 2^3$	6
—	$1^2 3^1$	2^2	5
3^2	3^1	—	6
3^1	—	6^1	6
—	9^1	—	9
			32

Alternative:

T1	T2	T3	Cost
$0^1 1^6$	$0^1 1^3$	—	
1^3	—	$0^1 2^3$	6
—	$1^1 3^2$	2^1	7
3^1	3^2	—	3
—	3^1	6^1	6
9^1	—	—	9
			31

(Still another way to improve on "optimum" polyphase is to reconsider where dummy runs appear on the output tape of every merge phase. For example, the result of merging $0^2 1^3$ with $0^2 1^3$ might be regarded as $2^1 0^1 2^1 0^1 2^1$ instead of $0^2 2^3$. Thus, many unresolved questions of optimality remain.)

19.

Level	T1	T2	T3	T4	Total	Final output on
0	1	0	0	0	1	T1
1	0	1	1	1	3	T6
2	1	1	1	0	3	T5
3	1	2	1	1	5	T4
4	2	2	2	1	7	T3
5	2	4	3	2	11	T2
6	4	5	4	2	15	T1
7	5	8	6	4	25	T6
. . . . .						
n	a_n	b_n	c_n	d_n	t_n	T(k)
$n+1$	b_n	$c_n + a_n$	$d_n + a_n$	a_n	$t_n + 2a_n$	T($k-1$)

20. $a(z) = 1/(1 - z^2 - z^3 - z^4)$, $t(z) = (3z + 3z^2 + 2z^3 + z^4)/(1 - z^2 - z^3 - z^4)$, $\sum_{n\geq 1} T_n(x)z^n = x(3z + 3z^2 + 2z^3 + z^4)/\big(1 - x(z^2 + z^3 + z^4)\big)$. $D_n = A_{n-1} + 1$, $C_n = A_{n-1}A_{n-2} + 1$, $B_n = A_{n-1}A_{n-2}A_{n-3} + 1$, $A_n = A_{n-2}A_{n-3}A_{n-4} + 1$.

21. 333343333332322 33334333333323 33334333333 3333433 333323 T5

22. $t_n - t_{n-1} - t_{n-2} = 2$ when $n \bmod 3 = 1$, and it is -1 otherwise. (This Fibonacci-like relation follows from the fact that $1 - z^2 - 2z^3 - z^4 = (1 - \phi z)(1 - \hat{\phi} z)(1 - \omega z)(1 - \bar{\omega} z)$, where $\omega^3 = 1$.)

23. In place of (25), the run lengths during the first half of the nth merge phase are s_n, and on the second half they are t_n, where

$$s_n = t_{n-2} + t_{n-3} + s_{n-3} + s_{n-4}, \qquad t_n = t_{n-2} + s_{n-2} + s_{n-3} + s_{n-4}.$$

Here we regard $s_n = t_n = 1$ for $n \leq 0$. [In general, if v_{n+1} is the sum of the first $2r$ terms of $u_{n-1} + \cdots + v_{n-P}$, we have $s_n = t_n = t_{n-2} + \cdots + t_{n-r} + 2t_{n-r-1} + t_{n-r-2} + \cdots + t_{n-P}$; if v_{n+1} is the sum of the first $2r - 1$, we have $s_n = t_{n-2} + \cdots + t_{n-r-1} + s_{n-r-1} + \cdots + s_{n-P}$, $t_n = t_{n-2} + \cdots + t_{n-r} + s_{n-r} + \cdots + s_{n-P}$.]

In place of (27) and (28), $A_n = (U_{n-1}V_{n-1}U_{n-2}V_{n-2}U_{n-3}V_{n-3}U_{n-4}V_{n-4}) + 1$, $\ldots$, $D_n = (U_{n-1}V_{n-1}) + 1$, $E_n = (U_{n-2}V_{n-2}U_{n-3}) + 1$; $V_{n+1} = (U_{n-1}V_{n-1}U_{n-2}) + 1$, $U_n = (V_{n-2}U_{n-3}V_{n-3}U_{n-4}V_{n-4}) + 1$.

25.

1^{16}	1^8	—	1^8
1^{12}	1^4	R	$1^8 2^4$
1^8	—	2^4	R
. . . . .			
R	$8^1 16^1$	8^1	8^0
16^0	R	8^1	—
16^1	16^1	8^0	R
R	16^1	—	24^0
16^1	16^1	R	$24^0 32^0$
16^0	16^0	32^1	(R)

26. When 2^n are sorted, $n \cdot 2^n$ initial runs are processed while merging; each half phase (with few exceptions) merges 2^{n-2} and rewinds 2^{n-1}. When $2^n + 2^{n-1}$ are sorted, $n \cdot 2^n + (n - 1) \cdot 2^{n-1}$ initial runs are processed while merging; each half phase (with few exceptions) merges 2^{n-2} or 2^{n-1} and rewinds $2^{n-1} + 2^{n-2}$.

27. It works if and only if the gcd of the distribution numbers is 1. For example, let there be six tapes; if we distribute (a, b, c, d, e) to T1 through T5, where $a \geq b \geq c \geq d \geq e > 0$, the first phase leaves a distribution $(a - e, b - e, c - e, d - e, e)$, and $\gcd(a - e, b - e, c - e, d - e, e) = \gcd(a, b, c, d, e)$. (Any common divisor of one set of numbers divides the others too.) The process decreases the number of runs at each phase until $\gcd(a, b, c, d, e)$ runs are left on a single tape.

[Note that these nonpolyphase distributions sometimes turn out to be superior to polyphase under certain configurations of dummy runs, as shown in exercise 18. This property was first observed by B. Sackman about 1963.]

28. We get any such (a, b, c, d, e) by starting with $(1, 0, 0, 0, 0)$ and doing the following operation exactly n times: Choose x in $\{a, b, c, d, e\}$, and add x to each of the other four elements of (a, b, c, d, e).

To show that $a + b + c + d + e \leq t_n$, we shall prove by induction that if $a \geq b \geq c \geq d \geq e$ we always have $a \leq a_n$, $b \leq b_n$, $c \leq c_n$, $d \leq d_n$, $e \leq e_n$. Assuming that this holds for level n, it must hold also for level $n + 1$, since the level $n + 1$

distributions are $(b+a,\ c+a,\ d+a,\ e+a,\ a)$, $(a+b,\ c+b,\ d+b,\ e+b,\ b)$, $(a+c, b+c, d+c, e+c, c)$, $(a+d, b+d, c+d, e+d, d)$, $(a+e, b+e, c+e, d+e, e)$.

30. The following table has been computed by J. A. Mortenson.

Level	$T=5$	$T=6$	$T=7$	$T=8$	$T=9$	$T=10$	
1	2	2	2	2	2	2	M_1
2	4	5	6	7	8	9	M_2
3	4	5	6	7	8	9	M_3
4	8	8	10	12	14	16	M_4
5	10	14	18	17	20	23	M_5
6	18	20	26	27	32	31	M_6
7	26	32	46	47	56	42	M_7
8	44	53	74	82	92	92	M_8
9	68	83	122	111	138	139	M_9
10	112	134	206	140	177	196	M_{10}
11	178	197	317	324	208	241	M_{11}
12	290	350	401	488	595	288	M_{12}
13	466	566	933	640	838	860	M_{13}
14	756	917	1371	769	1064	1177	M_{14}

SECTION 5.4.3

1. Comparing the average number of times each record is processed, in Table 5.4.2–6, the tape-splitting polyphase is superior when there are 6, 7, or 8 tapes.

2. The methods are essentially identical when the number of initial runs is a Fibonacci number; but the manner of distributing dummy runs in other cases is better with polyphase. The cascade algorithm puts 1 on T1, then 1 on T2, 1 on T1, 2 on T2, 3 on T1, 5 on T2, etc. and step D8 never finds $D[p-1] = M[p-1]$ when $p = 2$. In effect, all dummies are on one tape, and this is less efficient than the method of Algorithm 5.4.2D.

3. (Distribution stops after putting 12 runs on T3 during Step (3, 3).)

T1	T2	T3	T4	T5	T6
1^{26}	1^{21}	1^{24}	1^{14}	1^{15}	—
1^{5}	—	1^{12}	$1^{2}2^{7}$	1^{15}	$2^{2}4^{12}$
8^{4}	$6^{2}9^{3}$	5^{2}	6^{3}	1^{1}	—
—	9^{1}	23^{1}	17^{1}	25^{1}	26^{1}
100^{1}	—	—	—	—	—

4. Induction. (Cf. exercise 5.4.2–28.)

5. When there are a_n initial runs, the kth pass outputs a_{n-k} runs of length a_k, then b_{n-k} of length b_k, etc.

6.
$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

7. $e_2e_{n-2} + e_3e_{n-3} + \cdots + e_ne_0$ initial run lengths (cf. exercise 5), which may also be written $a_1a_{n-3} + a_2a_{n-4} + \cdots + a_{n-2}a_0$; it is the coefficient of z^{n-2} in $A(z)^2 - A(z)$.

8. The denominator of $A(z)$ has distinct roots and greater degree than the numerator, hence $A(z) = \sum q_3(\rho)/(1 - \rho z)\rho(1 - q_4'(\rho))$ summed over all roots ρ of $q_4(\rho) = \rho$. The special form of ρ is helpful in evaluating $q_3(\rho)$ and $q_4'(\rho)$.

9. The formulas hold for all *large* n, by (8) and (12), in view of the value of $q_m(2 \sin \theta_k)$. To show that they hold for all n we need to know that $q_{m-1}(z)$ is the quotient when $q_{r-1}(z)q_m(z)$ is divided by $q_r(z) - z$, for $0 \leq m < r$. This can be proved either by (a) using (10) and noting that cancellations bring down the degree of $q_{r-1}(z)q_m(z) - q_r(z)q_{m-1}(z)$; or (b) the result of exercise 5 implies that $A(z)^2 + B(z)^2 + \cdots + E(z)^2 \to 0$ as $z \to \infty$; or (c) it is possible to find an explicit formula for the numerators of $B(z)$, $C(z)$, etc.

10. $E(z) = r_1(z)A(z)$; $D(z) = r_2(z)A(z) - r_1(z)$; $C(z) = r_3(z)A(z) - r_2(z)$; $B(z) = r_4(z)A(z) - r_3(z)$; $A(z) = r_5(z)A(z) + 1 - r_4(z)$. Thus $A(z) = \big(1 - r_4(z)\big)/\big(1 - r_5(z)\big)$. [Note that $r_m(2 \sin \theta) = \sin (2m\theta)/\cos \theta$; $r_m(z)$ is the Chebyshev polynomial $(-1)^{m+1}U_{2m-1}(z/2)$.]

11. Prove that $f_m(z) = q_{\lfloor m/2 \rfloor}(z) - r_{\lceil m/2 \rceil}(z)$ and that $f_m(z)f_{m-1}(z) = 1 - r_m(z)$. Then use the result of exercise 10.

SECTION 5.4.4

1. When writing an ascending run, *first* write a sentinel record containing $-\infty$ before outputting the run. (And a $+\infty$ sentinel should be written at the end of the run as well, if the output is ever going to be read forward, e.g., on the final pass.) For descending runs, interchange the roles of $-\infty$ and $+\infty$.

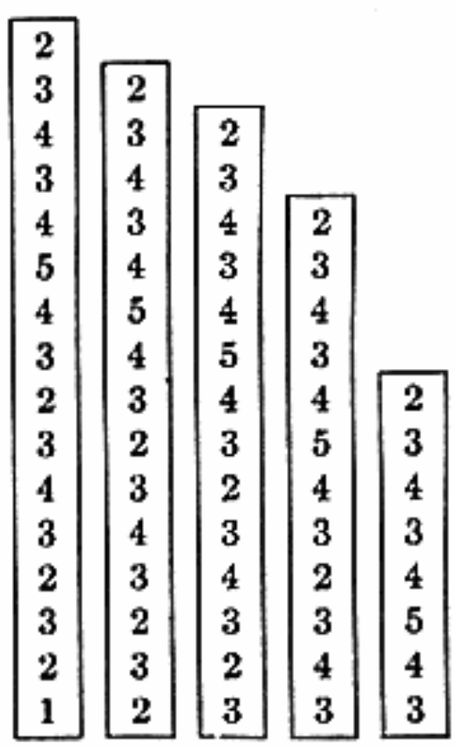
2. The smallest number on level $n+1$ is equal to the largest on level n; hence the columns are nondecreasing, regardless of the way we permute the numbers in any particular row.

3. In fact, during the merge process the first run on T2–T6 will always be descending, and the first on T1 will always be ascending. (By induction.)

4. It requires several "copy" operations on the second and third phases; the approximate extra cost is $(\log 2)/(\log \rho)$ passes, where ρ is the "growth ratio" in Table 5.4.2–1.

5. If α is a string, let α^R denote its left-right reversal.

Level	T1	T2	T3	T4	T5
0	0	—	—	—	—
1	1	1	1	1	1
2	12	12	12	12	2
3	1232	1232	1232	232	32
4	12323432	12323432	2323432	323432	3432
. . .	. . .	. . .	. . .	. . .	. . .
n	A_n	B_n	C_n	D_n	E_n
$n+1$	$B_n(A_n^R+1)$	$C_n(A_n^R+1)$	$D_n(A_n^R+1)$	$E_n(A_n^R+1)$	A_n^R+1



$$E_n = A^R_{n-1} + 1,$$
$$D_n = A^R_{n-2}A^R_{n-1} + 1,$$
$$C_n = A^R_{n-3}A^R_{n-2}A^R_{n-1} + 1,$$
$$B_n = A^R_{n-4}A^R_{n-3}A^R_{n-2}A^R_{n-1} + 1,$$
$$A_n = A^R_{n-5}A^R_{n-4}A^R_{n-3}A^R_{n-2}A^R_{n-1} + 1 = n - Q_n,$$

where

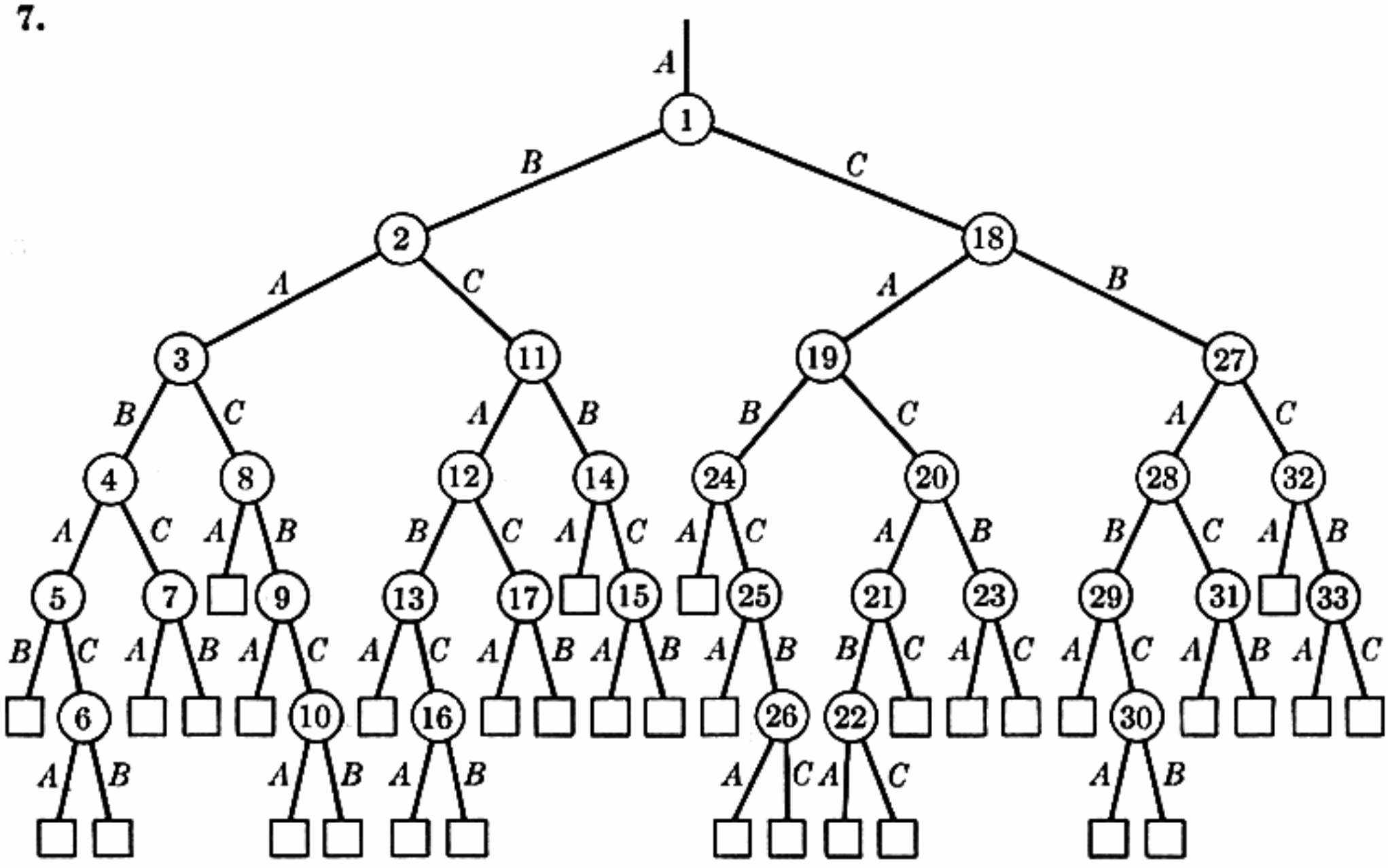
$$Q^R_n = Q_{n-1}(Q_{n-2}+1)(Q_{n-3}+2)(Q_{n-4}+3)(Q_{n-5}+4), \qquad n \geq 1.$$

Q_0 = '0', and Q_n is null for $n < 0$. In place of 5.4.2–12 we could consider a doubly infinite string which contains Q^R_1, Q_2, Q^R_3, Q_4, Q^R_5, etc. near its center.

These strings A_n, B_n, etc. contain the same entries as the corresponding strings in Section 5.4.2, but in another order. Note that adjacent merge numbers always differ by 1. An initial run must be A iff its merge number is even, D iff odd. Simple distribution schemes such as Algorithm 5.4.2D are not quite as effective at placing dummies into high-merge-number positions; therefore it is probably advantageous to compute Q_n between phases 1 and 2, in order to help control dummy run placement (assuming S is small enough that Q_n will fit comfortably in memory).

6. $y^{(4)} = (+1, +1, -1, +1)$
$y^{(3)} = (+1, \ 0, -1, \ 0)$
$y^{(2)} = (+1, -1, +1, +1)$
$y^{(1)} = (-1, +1, +1, +1)$
$y^{(0)} = (\ 1, \ 0, \ 0, \ 0)$

7.



Incidentally, 34 is apparently the smallest Fibonacci number F_n for which polyphase doesn't produce the optimum read-backwards merge for F_n initial runs on three tapes. This tree has external path length 178, which beats polyphase by two.

8. For $T = 4$, the tree with external path length 13 is not T-lifo, and every tree with external path length 14 includes a one-way merge.

9. We may consider a complete $(T - 1)$-ary tree, by the result of exercise 2.3.4.5–6; the degree of the "last" internal node is between 2 and $T - 1$. When there are $(T - 1)^q - m$ external nodes, $\lfloor m/(T - 2) \rfloor$ of them are on level $q - 1$, and the rest are on level q.

11. True by induction on the number of initial runs. If there is a valid distribution with S runs and two adjacent runs in the same direction, then there is one with $< S$ runs; but there is none when $S = 1$.

12. Conditions (a), (b) are obvious. If either configuration in (4) is present, for some tape name A and some $i < j < k$, node j must be in a subtree below node i and to the left of node k, by the definition of preorder. Hence the "$j - l$" case can't be present, and A must be the "special" name since it appears on an external branch. But this contradicts the fact that the special name is supposed to be on the leftmost branch below node i.

13. Nodes now numbered 4, 7, 11, 13 could be external instead of one-way merges. (This gives an external path length one higher than the polyphase tree.)

15. Let the tape names be A, B, and C. We shall construct several species of trees, botanically identified by their root and leaf (external node) structure:

Type $r(A)$. Root A.
Type $s(A, C)$. Root A, no C leaves.
Type $t(A)$. Root A, no A leaves.
Type $u(A, C)$. Root A, no C leaves, no compound B leaves.
Type $v(A, C)$. Root A, no C leaves, no compound A leaves.
Type $w(A, C)$. Root A, no A leaves, no compound C leaves.

A "compound leaf" is a leaf whose brother is not a leaf. We can grow a 3-lifo type $r(A)$ tree by first growing its left subtree as a type $s(B, C)$, then growing the right subtree as type $r(C)$. Similarly, type $s(A, C)$ comes from a type $s(B, C)$ then $t(C)$; type $u(A, C)$ from $v(B, C)$ and $w(C, B)$; type $v(A, C)$ from $u(B, C)$ and $w(C, A)$. We can grow a 3-lifo type $t(A)$ tree whose left subtree is type $u(B, A)$ and whose right subtree is type $s(C, A)$, by first letting the left subtree grow except for its (non-compound) C leaves and its right subtree; at this point the left subtree has only A and B leaves, so we can grow the right subtree of the whole tree, then grow off the A leaves of the left left subtree, and finally grow the left right subtree. Similarly, a type $w(A, C)$ tree can be fabricated from a $u(B, A)$ and a $v(C, A)$. [The tree of exercise 7 is an $r(A)$ tree constructed in this manner.]

Let $r(n), \ldots, w(n)$ denote the minimum external path length over all n-leaf trees of the relevant type, when they are constructed by such a procedure. We have $r(1) = s(1) = u(1) = 0$, $r(2) = t(2) = w(2) = 2$, $t(1) = v(1) = w(1) = s(2) = u(2) = v(2) = \infty$; and for $n \geq 3$,

$$r(n) = n + \min_k \big(s(k) + r(n - k)\big), \qquad u(n) = n + \min_k \big(v(k) + w(n - k)\big),$$
$$s(n) = n + \min_k \big(s(k) + t(n - k)\big), \qquad v(n) = n + \min_k \big(u(k) + w(n - k)\big),$$
$$t(n) = n + \min_k \big(u(k) + s(n - k)\big), \qquad w(n) = n + \min_k \big(u(k) + v(n - k)\big).$$

It follows that $r(n) \le s(n) \le u(n)$, $s(n) \le v(n)$, and $r(n) \le t(n) \le w(n)$ for all n; furthermore $s(2n) = t(2n+1) = \infty$. (The latter is evident *a priori*.)

Let $A(n)$ be the function defined by the laws $A(1) = 0$, $A(2n) = 2n + 2A(n)$, $A(2n+1) = 2n+1+A(n)+A(n+1)$; then $A(2n) = 2n + A(n-1) + A(n+1) -$ (0 or 1) for all $n \ge 2$. Let C be a constant such that, for $4 \le n \le 8$,

i) n even implies that $w(n) \le A(n) + Cn - 1$.
ii) n odd implies that $u(n)$ and $v(n)$ are $\le A(n) + Cn - 1$.

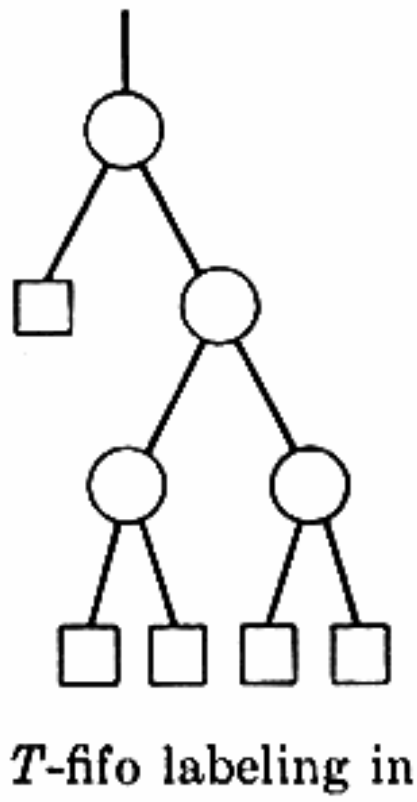
(This actually works for all $C \ge \frac{5}{6}$.) Then an inductive argument, choosing k to be $\lfloor n/2 \rfloor \pm 1$ as appropriate, shows that the relations are valid for *all* $n \ge 4$. But $A(n)$ is the lower bound in (9) when $T = 3$, and $r(n) \le \min\big(u(n), v(n), w(n)\big)$, hence we have proved that $A(n) \le K_3(n) \le r(n) \le A(n) + \frac{5}{6}n - 1$. [The constant $\frac{5}{6}$ can be lowered here.]

17.

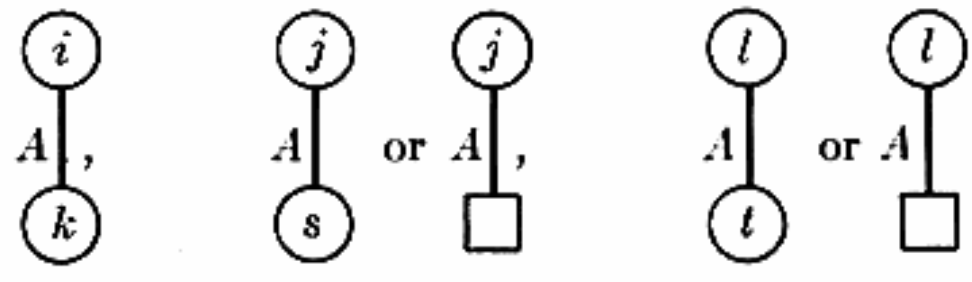
Level	T1	T2	T3	T4	T5
0	1	0	0	0	0
1	5	4	3	2	1
2	55	50	41	20	15
. . .	. . .	. . .	. . .	. . .	. . .
n	a_n	b_n	c_n	d_n	e_n
$n+1$	$5a_n + 4b_n + 3c_n + 2d_n + e_n$	$4a_n + 4b_n + 3c_n + 2d_n + e_n$	$3a_n + 3b_n + 3c_n + 2d_n + e_n$	$2a_n + 2b_n + 2c_n + 2d_n + e_n$	$a_n + b_n + c_n + d_n + e_n$

To get from level n to level $n+1$ during the initial distribution, insert k_1 "sublevels" with (4, 4, 3, 2, 1) runs added respectively to tapes (T1, T2, . . . , T5), k_2 "sublevels" with (4, 3, 3, 2, 1) runs added, k_3 with (3, 3, 2, 2, 1), k_4 with (2, 2, 2, 1, 1), k_5 with (1, 1, 1, 1, 0), where $k_1 \le a_n$, $k_2 \le b_n$, $k_3 \le c_n$, $k_4 \le d_n$, $k_5 \le e_n$. [If $(k_1, \ldots, k_5) = (a_n, \ldots, e_n)$ we have reached level $n+1$.] Add dummy runs if necessary to fill out a sublevel. Then merge $k_1 + k_2 + k_3 + k_4 + k_5$ runs from (T1, . . . , T5) to T6, merge $k_1 + \cdots + k_4$ from (T1, . . . , T4) to T5, . . . , merge k_1 from T1 to T2; and merge k_1 from (T2, . . . , T6) to T1, k_2 from (T3, . . . , T6) to T2, . . . , k_5 from T6 to T5. [This method was used in the UNIVAC III sort program, about 1962, and presented at the ACM Sort Symposium that year.]

19.

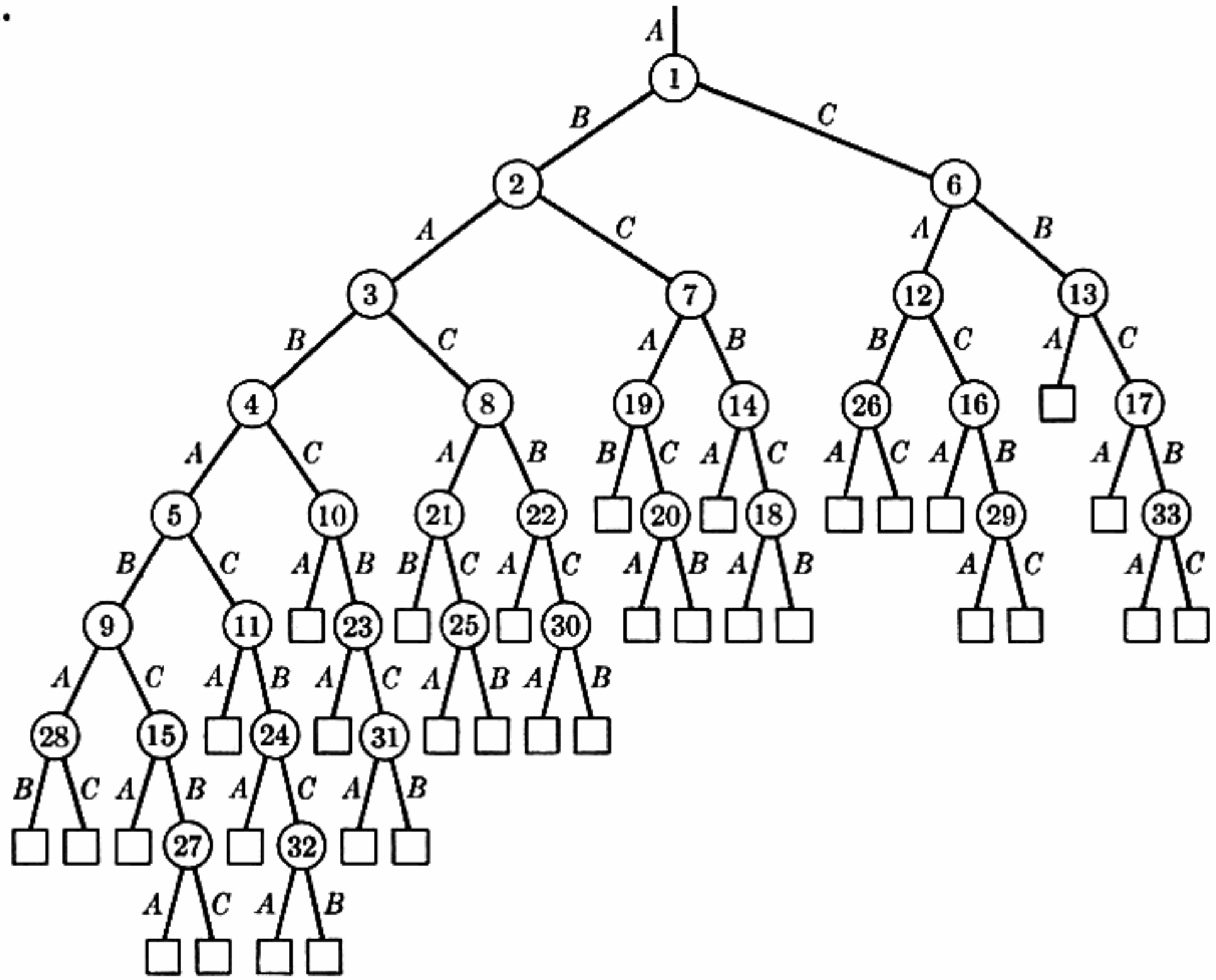


20. A strongly T-fifo tree has a T-fifo labeling in which there are no three branches having the respective forms



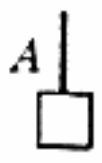
for some tape name A and some $i < j < k < l < s$. Informally, when we "grow on" an A, we must grow on all other A's before creating any new A's.

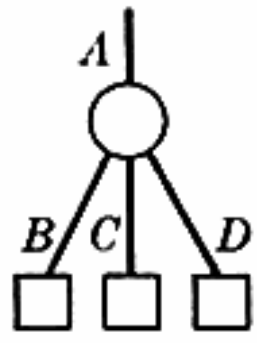
21.



which is very weakly fifo.

22. This occurs for any tree representations formed by successively replacing all occurrences of,

e.g., 

by, e.g., 

for some fixed tape names A, B, C, D.

Since all occurrences are replaced by the same pattern, the lifo or fifo order makes no difference in the structure of the tree.

Stating the condition in terms of the vector model: Whenever ($y^{(k+1)} \neq y^{(k)}$ or $k = m$) and $y_j^{(k)} = -1$, we have $y_j^{(k)} + \cdots + y_j^{(1)} + y_j^{(0)} = 0$.

23. (a) Assume that $v_1 \leq v_2 \leq \cdots \leq v_T$; the "cascade" stage $(1, \ldots, 1, -1)^{v_T}$ $(1, \ldots, 1, -1, 0)^{v_{T-1}} \ldots (1, -1, 0, \ldots, 0)^{v_2}$ takes $C(v)$ into v. (b) Immediate, since $C(v)_k \leq C(w)_k$ for all k. (c) If v is obtained in q stages, we have $u \to u^{(1)} \to \cdots \to u^{(q)} = v$ for some unit vector u, and some other vectors $u^{(1)}, \ldots$. Hence $u^{(1)} \preceq C(u)$, $u^{(2)} \preceq C(C(u))$, $\ldots$, $v \preceq C^q(u)$. Hence $v_1 + \cdots + v_T$ is less than or equal to the sum of the elements of $C^q(u)$; and the latter is obtained in cascade merge. [This theorem generalizes the result of exercise 5.4.3–4. Unfortunately the concept of "stage" as defined here doesn't seem to have any practical significance.]

24. Let $y^{(m)} \dots y^{(l+1)}$ be a stage which reduces w to v. If $y_j^{(0)} = -1$, $y_j^{(t-1)} = 0$, $\dots$, $y_j^{(k+1)} = 0$, and $y_j^{(k)} = -1$, for some $k < i - 1$, we can insert $y^{(k)}$ between $y^{(i)}$ and $y^{(i-1)}$. Repeat this operation until all (-1)'s in each column are adjacent. Then if $y_j^{(0)} = 0$ and $y_j^{(t-1)} \neq 0$ it is possible to set $y_j^{(t)} \leftarrow 1$; ultimately each column consists of $+1$'s followed by -1's followed by 0's, and we have constructed a stage which reduces w' to v for some $w' \geq w$. Permuting the columns, this stage takes the form $(1, \dots, 1, -1)^{a_T} \dots (1, -1, 0, \dots, 0)^{a_2}(-1, 0, \dots, 0)^{a_1}$. The sequence of $T - 1$ relations

$$\begin{aligned}(x_1, \dots, x_T) &\preceq (x_1 + x_T, \dots, x_{T-1} + x_T, 0)\\ &\preceq (x_1 + x_{T-1} + x_T, \dots, x_{T-2} + x_{T-1} + x_T, x_T, 0)\\ &\preceq (x_1 + x_{T-2} + x_{T-1} + x_T, \dots, x_{T-3} + x_{T-2} + x_{T-1} + x_T,\\ &\qquad x_{T-1} + x_T, x_T, 0) \preceq \cdots\\ &\preceq (x_1 + x_2 + x_3 + \cdots + x_T, x_3 + \cdots + x_T, \dots, x_{T-1} + x_T, x_T, 0)\end{aligned}$$

now shows that the best choice of the a's is $a_T = v_T$, $a_{T-1} = v_{T-1}$, $\dots$, $a_2 = v_2$, $a_1 = 0$. And the result is best if we permute columns so that $v_1 \leq \cdots \leq v_T$.

25. (a) Assume that $v_{T-k+1} \geq \cdots \geq v_T \geq v_1 \geq \cdots \geq v_{T-k}$ and use $(1, \dots, 1, -1, 0, \dots, 0)^{v_T - v_{T-k+1}} \dots (1, \dots, 1, 0, \dots, 0, -1)^{v_T}$. (b) The sum of the l largest elements of $D_k(v)$ is $(l-1)s_k + s_{k+l}$, for $1 \leq l \leq T - k$. (c) If $v \Rightarrow w$ in a phase that uses k output tapes, we may obviously assume that the phase has the form $(1, \dots, 1, -1, 0, \dots, 0)^{a_1} \dots (1, \dots, 1, 0, \dots, 0, -1)^{a_k}$, with each of the other $T - k$ tapes used as input in each operation. Choosing $a_1 = v_{T-k+1}, \dots, a_k = v_T$ is best. (d) See exercise 22(c). We always have $k_1 = 1$; and $k = T - 2$ always beats $k = T - 1$ since we assume that at least one component of v is zero. Hence for $T = 3$ we have $k_1 \dots k_q = 1^q$ and the initial distribution $(F_{q+1}, F_q, 0)$. For $T = 4$ the undominated strategies and their corresponding distributions are found to be

$q = 2$	12 (3, 2, 0, 0)
$q = 3$	121 (5, 3, 3, 0); 122 (5, 5, 0, 0)
$q = 4$	1211 (8, 8, 5, 0); 1222 (10, 10, 0, 0); 1212 (11, 8, 0, 0)
$q = 5$	12121 (19, 11, 11, 0); 12222 (20, 20, 0, 0); 12112 (21, 16, 0, 0)
$q = 6$	122222 (40, 40, 0, 0); 121212 (41, 30, 0, 0)
$q \geq 7$	12^{q-1} $(5 \cdot 2^{q-3}, 5 \cdot 2^{q-3}, 0, 0)$

So for $T = 4$ and $q \geq 6$, the minimum-phase merge is like balanced merge, with a slight twist at the very end (going from (3, 2, 0, 0) to (1, 0, 1, 1) instead of to (0, 0, 2, 1)).

When $T = 5$ the undominated strategies are $1(32)^{n-1}2$, $1(32)^{n-1}3$ for $q = 2n \geq 2$; $1(32)^{n-1}32$, $1(32)^{n-1}22$, $1(32)^{n-1}23$ for $q = 2n + 1 \geq 3$. (The first strategy listed has most runs in its distribution.) On six tapes they are 13 or 14, 142 or 132 or 133, 1333 or 1423, then 13^{q-1} for $q \geq 5$.

SECTION 5.4.5

1. The following algorithm is controlled by a table A[L — 1] . . . A[1]A[0] that essentially represents a number in radix P notation. As we repeatedly add unity to this number, the "carries" tell us when to merge. Tapes are numbered from 0 to P.

O1. [Initialize.] Set $(\mathtt{A}[L-1], \ldots, \mathtt{A}[0]) \leftarrow (0, \ldots, 0)$ and $q \leftarrow 0$. (During this algorithm, q will equal $(\mathtt{A}[L-1] + \cdots + \mathtt{A}[0]) \bmod T$.)

O2. [Distribute.] Write an initial run on tape q, in ascending order. Set $l \leftarrow 0$.

O3. [Add one.] If $l = L$, stop; the output is on tape $(-L) \bmod T$, in ascending order iff L is even. Otherwise set $\mathtt{A}[l] \leftarrow \mathtt{A}[l] + 1$, $q \leftarrow (q+1) \bmod T$.

O4. [Carry?] If $\mathtt{A}[l] < P$, return to O2. Otherwise merge to tape $(q-l) \bmod T$, increase l by 1, set $\mathtt{A}[l] \leftarrow 0$ and $q \leftarrow (q+1) \bmod T$, and return to O3. ∎

2. Keep track of how many runs are on each tape. When the input is exhausted, add dummy runs if necessary until reaching a situation with at most one run on each tape and at least one tape empty. Then finish the sort in one more merge, rewinding some tapes first if necessary. (It is possible to deduce the orientation of the runs from the A table.)

3.

Op	T0	T1	T2
Dist	—	A_1	A_1A_1
Merge	D_2	—	A_1
Dist	D_2A_1	—	A_1
Merge	D_2	D_2	—
Dist	D_2	D_2A_1	A_1
Merge	D_2D_2	D_2	—
Merge	D_2	—	A_4

Op	T0	T1	T2
Dist	D_2A_1	A_1	A_4
Merge	D_2	—	A_4D_2
Merge	—	A_4	A_4
Dist	—	A_4	A_4A_1
Copy	—	A_4D_1	A_4
Copy	—	A_4	A_4A_1
Merge	D_5	—	A_4

At this point T2 would be rewound and a final merge would complete the sort. To avoid the useless copy operations, replace step B4 by:

B4′. [Ready to merge?] If $\mathtt{A}[l-1, q] = s$, go to step B5. Otherwise if the input is exhausted and $\mathtt{A}[l-1, q] - s$ is a positive even number *and* $\mathtt{A}[l-1, (2r-q) \bmod T] \neq \mathtt{A}[l-1, q] - 1$, set $s \leftarrow \mathtt{A}[l-1, q]$ and go to B5. Otherwise go back to step B3.

[This change avoids situations in which the runs are simply shifted back and forth. The first parenthesized comment in step B5 is no longer strictly correct, but the algorithm still works properly because the A table entries on level l will not be looked at. The smallest S for which this change makes any difference is $P^3 + 1$. When P is large, the change hardly ever makes much difference, but it does keep the computer from looking too foolish in some circumstances. The algorithm should also be changed to handle the case $S = 1$ more efficiently.]

4. We can, in fact, omit setting $A[0, 0]$ in step B1, $A[l, q]$ in steps B3 and B5. [But $A[l, r]$ *must* be set in step B3.]

5. $P^{2k} - (P-1)P^{2k-2} < S \leq P^{2k}$ for some $k > 0$.

SECTION 5.4.6

1. $\lfloor 23000480/(n+480) \rfloor n$.

2. At the instant shown, all the records in that buffer have been moved to the output. Step F2 insists that the test "Is output buffer full?" precede the test "Is input buffer empty?" while merging, otherwise we would have trouble (unless the changes of exercise 4 were made).

3. No, e.g., we might reach a state with P buffers $1/P$ full and $P - 1$ buffers full, if file i contains the keys i, $i + P$, $i + 2P$, ... for $1 \le i \le P$. This example shows that $2P$ input buffers are necessary for continuous output even if we allowed simultaneous reading, unless we reallocated memory for partial buffers or used a "scatter read" in some way. [Actually, if blocks contain less than $P - 1$ records, we need less than $2P$ buffers, but this is unlikely. Older computers would read a block of data into a special hardware buffer separate from the computer memory, and computation would stop while the contents of this buffer were transferred to memory. In such a situation, $2P - 1$ "software" buffers in memory would be sufficient.]

4. Set up **S** sooner (in steps **F1** and **F4** instead of **F3**).

5. If, for example, all keys of all files were equal, we couldn't simply make arbitrary decisions while forecasting; the forecast must be compatible with decisions made by the merging process. One safe way is to find the smallest possible m in steps F1 and F4, i.e. to consider a record from file $C[i]$ to be less than all records having the same key on file $C[j]$ whenever $i < j$. (In essence, the file number is appended to the key.)

6. In step C1 also set $\texttt{TAPE}[T + 1] \leftarrow T + 1$. In step C8 the merge should be to $\texttt{TAPE}[p + 2]$ instead of $\texttt{TAPE}[p + 1]$. In step C9, set $(\texttt{TAPE}[1], \ldots, \texttt{TAPE}[T + 1]) \leftarrow (\texttt{TAPE}[T + 1], \ldots, \texttt{TAPE}[1])$.

7. The method used in Chart A is $(A_1D_1)^4A_0D_0(A_1D_1)^2A_0D_0(A_1D_1)^3A_0$, $D_1(A_1D_1)^4A_0D_0(A_1D_1)^3A_0D_0\alpha A_0D_0A_0$, $D_1A_0D_0(A_1D_1)^3A_0D_0\alpha A_1D_1A_0$, $D_1A_1D_1\alpha A_1D_1A_0$, where $\alpha = (A_0D_0)^2A_1D_1A_0D_0(A_1D_1)^2(A_0D_0)^7A_1D_1(A_0D_0)^3A_1D_1A_0D_0$. The first merge phase writes $D_0A_3D_3A_1D_1A_4D_4A_0D_0A_1D_1A_1D_1A_4D_4A_0D_0A_1D_1A_0D_0(A_1D_1)^4$ on tape 5; the next writes $A_4D_4A_4D_4A_1D_1A_4D_4A_0D_0A_1D_1A_1D_1A_7$ on tape 1; the next, $D_{13}A_4D_4A_0D_0A_{10}$ on tape 4. The final phases are

$A_4D_4A_4$	—	$D_{19}A_3D_3A_{12}$	$D_{13}A_4D_4A_4$	D_0A_3
A_4	$D_{23}A_{11}$	$D_{19}A_3$	$D_{13}A_4$	—
—	D_{23}	D_{19}	D_{13}	D_{22}
A_{77}	—	—	—	—

8. No, since at most S stop/starts are saved, and since the speed of the input tape (not the output tapes) tends to govern the initial distribution time anyway. The other advantages of the distribution schemes used in Chart A outweigh this miniscule disadvantage.

9. $P = 5$, $B = 8300$, $B' = 734$, $S = \lceil (3 + 1/P)N/6P' \rceil + 1 = 74$, $\omega = 1.094$, $\alpha = 0.795$, $\beta = -1.136$, $\alpha' = \beta' = 0$; (9) = 855 seconds, to which we add the time for initial rewind, for a total of 958 seconds. The savings of about one minute in the merging does not compensate for the loss of time due to the initial rewinding and tape changing (unless perhaps we are in a multiprogramming environment).

10. The rewinds during standard polyphase merge are over about 54 percent of the file (the "pass/phase" column in Table 5.4.2–1), and the longest rewinds during standard cascade merge cover approximately $a_k a_{n-k}/a_n \approx (4/(2T - 1)) \cos^2 (\pi/(4T - 2)) < \frac{4}{11}$ of the file, by exercise 5.4.3–5 and Eq. 5.4.3–13.

11. Only initial and final rewinds get to make use of the "high-speed" feature, since the reel is only a little more than 10/23 full when it contains the whole example file.

Using $\pi = \lceil .946 \ln S - 1.204 \rceil$, $\pi' = 1/8$ in example 8, we get the following estimated totals for examples 1–9, respectively:

$$1115,\ 1296,\ 1241,\ 1008,\ 1014,\ 967,\ 891,\ 969,\ 856.$$

12. (a) An obvious solution with $4P + 4$ buffers simply reads and writes simultaneously from paired tapes. But note that three output buffers are sufficient (at a given moment we are performing the second half of a write from one, the first half of a write from another, and outputting into the third), and this suggests a corresponding improvement in the input buffer situation. It turns out that $3P$ input buffers and 3 output buffers are necessary and sufficient, using a slightly weakened "forecasting" technique. A simpler and superior approach, suggested by J. Sue, adds a "lookahead key" to each block, specifying the final key of the subsequent block. Sue's method requires $2P + 1$ input buffers and 4 output buffers, and it is a straightforward modification of Algorithm F.

(b) In this case the high value of α means that we must do between five and six passes over the data, which wipes out the advantage of double-quick merging. The idea works out much better on eight or nine tapes.

13. No, consider for example the situation just before $A_{16}\ A_{16}\ A_{16}\ A_{16}$. But two reelfulls *can* be handled.

14.
$$\det\begin{pmatrix} 0 & -p_0z & 0 & z-1 \\ 0 & 1-p_1z & -p_0z & z-1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \Big/ \det\begin{pmatrix} 1-p_{\geq 1}z & -p_0z & 0 & z-1 \\ -p_{\geq 2}z & 1-p_1z & -p_0z & z-1 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

15. The A matrix has the form

$$A = \begin{pmatrix} B_{10}z & B_{11}z & \dots & B_{1n}z & 1-z \\ \vdots & & & & \vdots \\ B_{n0}z & B_{n1}z & \dots & B_{nn}z & 1-z \\ 0 \dots 0 & & 1 & 0 & 0 \\ 0 \dots 0 & & 0 & 0 & 0 \end{pmatrix}, \qquad \begin{aligned} B_{10} + B_{11} + \cdots + B_{1n} &= 1 \\ &\vdots \\ B_{n0} + B_{n1} + \cdots + B_{nn} &= 1 \end{aligned} \quad . \tag{11}$$

Therefore

$$\det(I - A) = \det\begin{pmatrix} 1-B_{10}z & -B_{11}z & \dots & -B_{1,n-1}z & -B_{1n}z \\ \vdots & & & & \vdots \\ -B_{n0}z & -B_{n1}z & \dots & 1-B_{n,n-1}z & -B_{nn}z \\ 0 & 0 & & -1 & 1 \end{pmatrix}$$

and we can add all columns to the first column, then factor out $(1 - z)$. Consequently $g_Q(z)$ has the form $h_Q(z)/(1-z)$, and $\alpha^{(Q)} = h_Q(1)$ because $h_Q(1) \neq 0$ and $\det(I - A) \neq 0$ for $|z| < 1$.

SECTION 5.4.7

1. Sort from least significant digit to most significant digit in the number system whose radices are alternately P and $T - P$. (If pairs of digits are grouped, we have

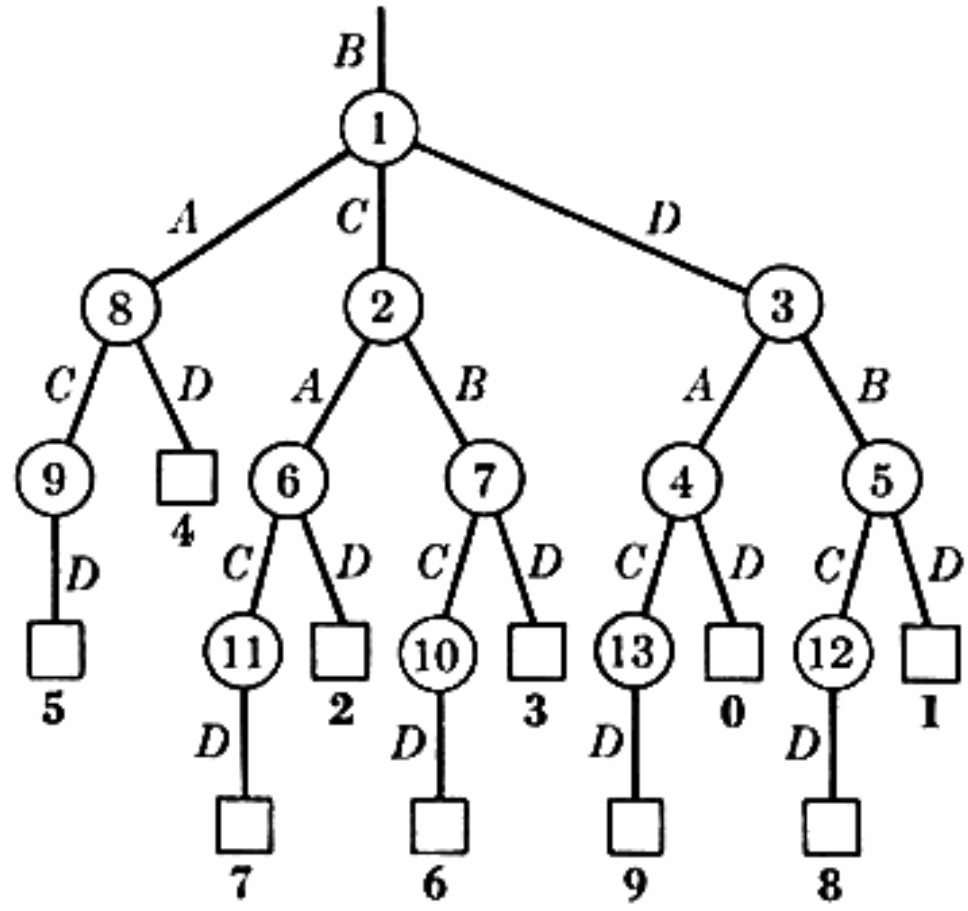
essentially a pure radix of $P \cdot (T - P)$. Thus, if $P = 2$ and $T = 7$, the number system is "biquinary," related to decimal notation in a simple way.)

2. If K is a key between 0 and $F_n - 1$, let the Fibonacci representation of $F_n - 1 - K$ be $a_{n-2}F_{n-1} + \cdots + a_1F_2$, where the a_j are 0 or 1, and no two consecutive 1's appear. After phase j, tape $j + 1$ (modulo 3) contains those keys with $a_j = 0$ and tape $j - 1$ (modulo 3) contains those with $a_j = 1$, in decreasing order of $a_{j-1} \ldots a_1$.

[Imagine a card sorter with two pockets "0" and "1," and consider the procedure of sorting F_n cards that have been punched with the keys $a_{n-2} \ldots a_1$ in $n - 2$ columns. The conventional procedure for sorting these into decreasing order, starting at the least significant digit, can be simplified since we know that everything in the "1" pocket at the end of one pass will go into the "0" pocket on the following pass.]

4. If there were an external node on level 2 we could not construct such a good tree. Otherwise there are at most three external nodes on level 3, six on level 4, since each external node is supposed to appear on the same tape.

5.



6. 09, 08, . . . , 00, 19, . . . , 10, 29, . . . , 20, 39, . . . , 30, 40, 41, . . . , 49, 59, . . . , 50, 60, 61, . . . , 99.

7. Yes; first distribute the records into smaller and smaller subfiles until obtaining one-reel files which can be individually sorted. This is dual to the process of sorting one-reel files and then merging them into larger and larger multireel files.

SECTION 5.4.8

1. Yes. The ordering of the file and the selection tree alternates between ascending and descending, and we have in effect an order-P cocktail shaker sort; cf. exercise 9.

2. Let $Z_N = Y_N - X_N$, and solve the recurrence for Z_N by noting that $(N+1)NZ_{N+1} = N(N-1)Z_N + N^2 + N$; hence

$$Z_N = \tfrac{1}{3}(N+1) + \binom{M+2}{3} \Big/ N(N-1), \qquad \text{for} \qquad N > M.$$

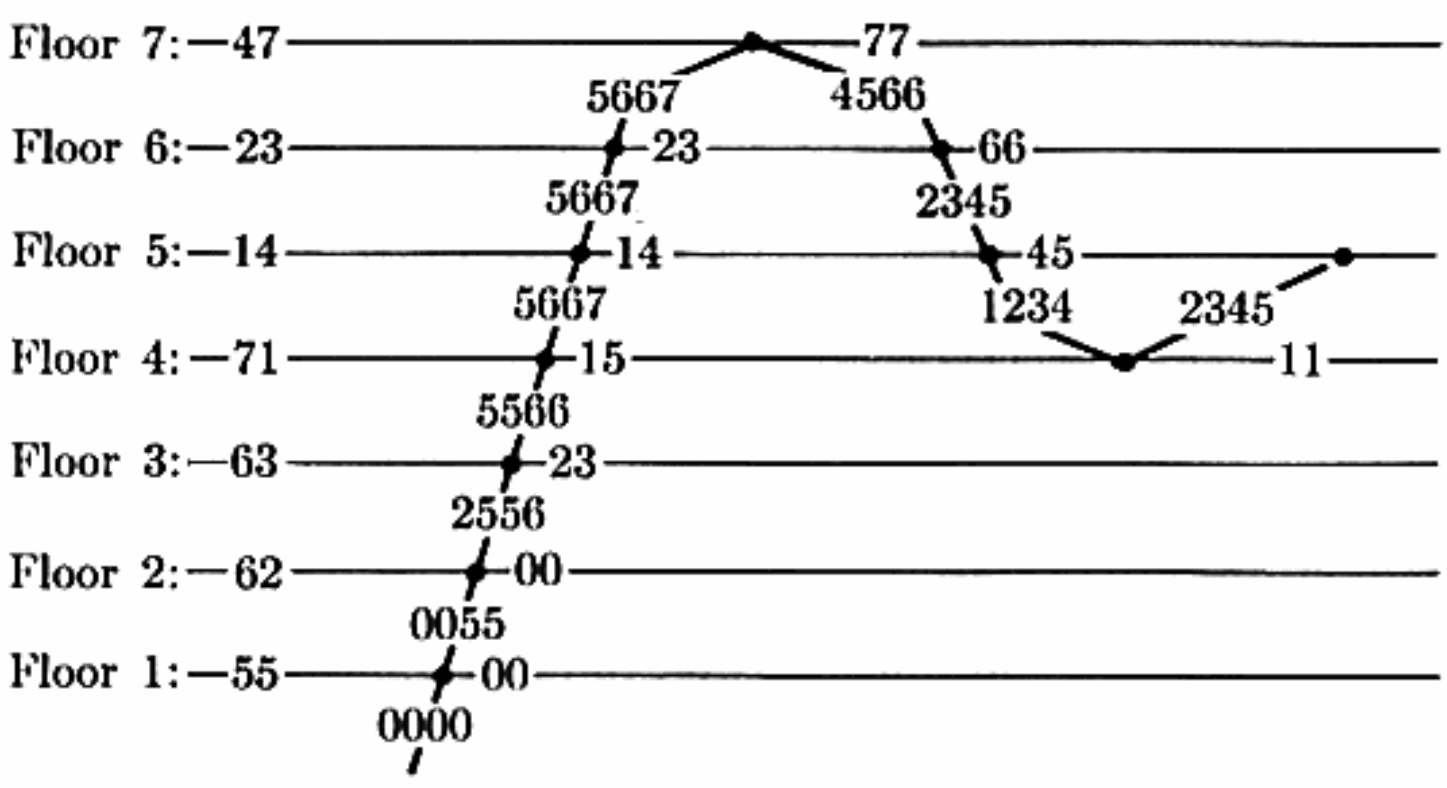
Now eliminate Y_N and obtain

$$\frac{X_N}{N+1} = 6\tfrac{2}{3}(H_{N+1} - H_{M+2}) + 2\left(\frac{1}{N+1} - \frac{1}{M+2}\right)$$
$$-\frac{2}{3}\binom{M+2}{3}\left(\frac{1}{(N+1)N(N-1)} - \frac{1}{(M+2)(M+1)M}\right) + \frac{3M+4}{M+2}, \qquad N > M.$$

3. Yes; find a median element in $O(N)$ steps, using a construction like that of Theorem 5.3.3, and use it to partition the file. Another interesting approach, due to R. W. Floyd and A. J. Smith, is to merge two runs of N items in $O(N)$ units of time as follows: Spread the items out on the tapes, with spaces between them, then successively fill each space with a number specifying the final position of the item just preceding that space.

4. It is possible to piece together a schedule for floors $1, \ldots, p+1$ with a schedule for floors $q, \ldots, n$: When the former schedule first reaches floor $p+1$, go up to floor q and carry out the latter schedule (using the current elevator contents as if they were the extra "0" men in the algorithm of Theorem K). After finishing that schedule, go back to floor $p+1$ and resume the previous schedule.

5. Consider $b = 4$, $c = 2$ and the following behavior of the algorithm:



Now 2 (in the elevator) is less than 3 (on floor 3).

[After constructing an example such as this, the reader should be able to see how to demonstrate the weaker property required in the proof of Theorem K.]

6. Let i, j be minimal with $c_i < c_i'$ and $c_j > c_j'$. Introduce a new man who wants to go from i to j. This doesn't increase $\max(u_k, d_{k+1}, 1)$ or $\max(c_k, c_k')$ for any k. Continue this until $c_j = c_j'$ for all j. Now observe that the algorithm in the text works also with c replaced by c_k in steps K1 and K3.

8. Let the number be P_n, and let Q_n be the number of permutations such that $u_k = 1$ for $1 \le k < n$. Then $P_n = Q_1P_{n-1} + Q_2P_{n-2} + \cdots + Q_nP_0$, $P_0 = 1$. It can be shown that $Q_n = 3^{n-2}$ for $n \ge 2$ (see below), hence a generating function argument yields

$$\sum P_n z^n = (1 - 3z)/(1 - 4z + 2z^2) = 1 + z + 2z^2 + 6z^3 + 20z^4 + \cdots;$$
$$2P_n = (2+\sqrt{2})^{n-1} + (2-\sqrt{2})^{n-1}.$$

To prove that $Q_n = 3^{n-2}$, consider a ternary sequence $x_1x_2 \ldots x_n$ such that $x_1 = 2, x_n = 0$, and $0 \leq x_k \leq 2$ for $1 < k < n$. The following rule defines a one-to-one correspondence between such sequences and the desired permutations $a_1a_2 \ldots a_n$:

$$a_k = \begin{cases} \max \{j \mid (j < k \quad \text{and} \quad x_j = 0) \quad \text{or} \quad j = 1\}, & \text{if} \quad x_k = 0; \\ k \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , & \text{if} \quad x_k = 1; \\ \min \{j \mid (j > k \quad \text{and} \quad x_j = 2) \quad \text{or} \quad j = n\}, & \text{if} \quad x_k = 2. \end{cases}$$

(This correspondence was obtained by the author jointly with E. A. Bender.)

9. The number of passes of the cocktail shaker sort is $2 \max (u_1, \ldots, u_n)$ — (0 or 1), since each pair of passes (left-right-left) reduces each of the nonzero u's by 1.

10. Begin with a distribution method (quicksort or radix exchange) until one-reel files are obtained. And be patient.

SECTION 5.4.9

1. $\frac{1}{4} - (x \bmod \frac{1}{2})^2$ revolutions.

2. The probability that $k = a_{iq}$ and $k+1 = a_{i'r}$ for fixed k, q, r, and $i \neq i'$ is $f(q, r, k)L!L!(PL - 2L)!/(PL)!$, where

$$f(q,r,k) = \binom{k-1}{q-1}\binom{k-q}{r-1}\binom{PL-k-1}{L-q}\binom{PL-k-1-L+q}{L-r}$$
$$= \binom{k-1}{q+r-2}\binom{q+r-2}{q-1}\binom{PL-k-1}{2L-q-r}\binom{2L-q-r}{L-q};$$

and

$$\sum_{\substack{1\le k<PL\\ 1\le q,r\le L}} |q-r|\, f(q,r,k) = \sum_{q,r} |q-r| \binom{PL-1}{2L-1}\binom{q+r-2}{q-1}\binom{2L-q-r}{L-q}$$
$$= \binom{PL-1}{2L-1} A_{2L-1}.$$

The probability that $k = a_{iq}$ and $k+1 = a_{i,q+1}$ for fixed k, q, and i is

$$g(k,q) \Big/ \binom{PL}{L}, \quad \text{where} \quad g(k,q) = \binom{k-1}{q-1}\binom{PL-k-1}{L-q-1};$$

and

$$\sum_{\substack{1\le k<PL\\ 1\le q<L}} g(k,q) = \sum_{1\le q<L} \binom{PL-1}{L-1} = (L-1)\binom{PL-1}{L-1}.$$

[*SIAM J. Computing* **1** (1972), 161–166.]

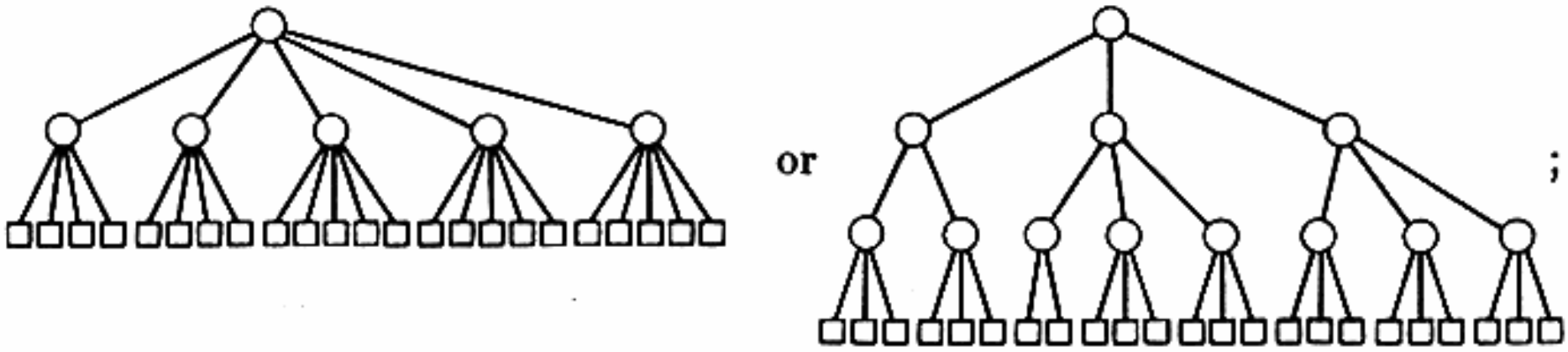
3. Take the minimum in (5) over the range $2 \le m \le \min(9, n)$.

4. (a) $(0.000725(\sqrt{P}+1)^2 + 0.014)L$. (b) Change "$\alpha mn + \beta n$" in formula (5) to "$(0.000725(\sqrt{m}+1)^2 + 0.014)n$." [Computer experiments show that the nature of the optimal trees defined by this new recurrence is very much the same as those

defined by Theorem K with $\alpha = 0.00145$, $\beta = 0.01545$; in fact, trees exist which are optimal for both recurrences, when $30 \le n \le 100$. The change suggested in this exercise saves about 10 percent of the merging time, when $n = 64$ or 100 as in the text's example. This style of buffer allocation was considered already in 1954 by H. Seward, who found that four-way merging minimizes the seek time.]

5. Let $A_m(n)$, $B_m(n)$ represent sets of m trees all of whose leaves are at (even, odd) levels, respectively. Then $A_1(1) = 0$, $B_1(1) = \alpha + \beta$; $A_m(n)$ and $B_m(n)$ are defined as in (4) when $m \ge 2$; $A_1(n) = \min_{1 \le m \le n} (\alpha mn + \beta n + B_m(n))$, $B_1(n) = \min_{1 \le m \le n} (\alpha mn + \beta n + A_m(n))$. The latter equations are well-defined in spite of the fact that $A_1(n)$ and $B_1(n)$ are defined in terms of each other!

6.



$A_1(23) = B_1(23) = 268$. [Curiously, $n = 23$ is the only value ≤ 50 for which no equal-parity tree with n leaves is optimal in the unrestricted-parity case. Perhaps it is the *only* such value, when $\alpha = \beta$.]

7. Consider the quantities $\alpha d_1 + \beta e_1, \ldots, \alpha d_n + \beta e_n$ in any tree, where (d_j, e_j) are the (degree sum, path length) for the jth leaf. An optimum tree for weights $w_1 \le \cdots \le w_n$ will have $\alpha d_1 + \beta e_1 \ge \cdots \ge \alpha d_n + \beta e_n$. It is always possible to reorder these so that $\alpha d_1 + \beta e_1 = \cdots = \alpha d_1 + \beta e_k$ where the first k leaves are merged together.

11. Using exercise 1.2.4–38, $f_m(n) = 3qn + 2(n - 3^q m)$, when $2 \cdot 3^{q-1} \le n/m \le 3^q$; $3qn + 4(n - 3^q m)$, when $3^q \le n/m \le 2 \cdot 3^q$. Thus $f_2(n) + 2n \ge f(n)$, with equality iff $4 \cdot 3^{q-1} \le n \le 2 \cdot 3^q$; $f_3(n) + 3n = f(n)$; $f_4(n) + 4n \ge f(n)$, with equality iff $n = 4 \cdot 3^q$; and $f_m(n) + mn > f(n)$ for all $m \ge 5$.

12. Use the specifications —, 1:1, 1:1:1, 1:1:1:1 or 2:2, 2:3, 2:2:2, . . . ,$\lfloor n/3 \rfloor$: $\lfloor (n+1)/3 \rfloor$:$\lfloor (n+2)/3 \rfloor$, . . . ; this gives trees with all leaves at level $q + 2$, for $4 \cdot 3^q \le n \le 4 \cdot 3^{q+1}$. (When $n = 4 \cdot 3^q$, two such trees are formed.)

14. The following tree specifications were found for $n = 1, 2, 3, \ldots$, by exhaustively examining all partitions of n. —, 1:1, 1:1:1, 1:1:1:1, 1:1:1:1:1, 1:1:1:1:1:1, 1:1:1:1:3, 1:1:3:3, 3:3:3, 1:3:3:3, 3:4:4, 3:3:3:3, 3:3:3:4, 3:3:4:4, 3:4:4:4, 4:4:4:4, . . . , 5:6:6:6:12, 6:6:6:6:12, 6:6:6:6:13, ... (The degrees seem to be always ≤ 6, but such a result appears to be quite difficult to prove.)

15. If a people initially got on the elevator, the togetherness rating increases by at most $a + c$ at the first stop. When it next stops at the initial floor, the rating increases by at most $b + c - a$. Hence the rating increases at most $(k - 1)b + kc$ after k stops.

16. 123456 → floor 2; 112334 → floor 3; 224456 → floor 4; 135566 → floor 5; 112334 → floor 6; 224456 → floor 2; 222444 → floor 4; 222222 → floor 2; 555666 → floor 5; 666666 → floor 6; 111333 → floor 3; 111111 → floor 1. [Is there a shorter solution? Exercise 15 implies that at least nine stops are needed.]

17. Consider doing step (g) backwards, distributing the records into bin 1, then bin 2, bin 3. This operation is precisely what step (d) is simulating on the key file.

18. The internal sort must be carefully chosen, with paging in mind; methods such as

Shell's diminishing increment sort, address calculation, heapsort, and list sorting would be disastrous, since they require a large "working set" of pages. Quicksort, radix exchange, and sequentially-allocated merge or radix sorting are much better suited to a paging environment.

Some things the designer of an external sort can do, which are "virtually impossible" to include in an automatically paged method, are: (a) Forecasting the input file which should be read next, so that the data is available when it is required; (b) choosing the buffer sizes and the order of merge according to hardware and data characteristics.

On the other hand a virtual machine is considerably easier to program, and it can give results that aren't bad, if the programmer is careful and knows the properties of the underlying actual machine. See the study made by Brawn, Gustavson, and Mankin [*CACM* **13** (1970), 483–494].

SECTION 5.5

1. *It is difficult to decide which sorting algorithm is best in a given situation.* ▮

2. For small N, list insertion; for medium N, e.g. $N = 64$, list merge; for large N, radix list sort.

SECTION 6.1

1. $\sqrt{(N^2 - 1)/12}$.

2. **S1′.** [Initialize.] Set P ← FIRST.

   S2′. [Compare.] If K = KEY(P), the algorithm terminates successfully.

   S3′. [Advance.] Set P ← LINK(P).

   S4′. [End of file?] If P ≠ Λ, go back to S2′. Otherwise the algorithm terminates unsuccessfully. ▮

3.

```
KEY       EQU   3:5
LINK      EQU   1:2
START     LDA   K              1
          LD1   FIRST          1
2H        CMPA  0,1(KEY)       C
          JE    SUCCESS        C
          LD1   0,1(LINK)      C - S
          J1NZ  2B             C - S
FAILURE   EQU   *              1 - S   ▮  (6C - 3S + 4)u.
```

4. Yes, if we have a way to set "KEY(Λ)" equal to K. [But the technique of loop duplication used in Program Q′ has no effect in this case.]

5. No, Program Q always does at least as many operations as Program Q′.

6. Replace line 08 by

```
          JE    *+4
          CMPA  KEY+N+2,1
          JNE   3B
          INC1  1
```

and change lines 03–04 to `ENT1 -2-N; 3H INC1 3`.

7. Note that $\overline{C}_N = \frac{1}{2}\overline{C}_{N-1} + 1$.

8. Euler's summation formula gives

$$H_n^{(x)} \sim \text{const} + n^{1-x}/(1-x) + \tfrac{1}{2}n^{-x} - B_2 x n^{-1-x}/2! + B_3 x(x+1)n^{-2-x}/3! - \cdots.$$

[The constant is $\zeta(x)$; complex variable theory tells us that

$$\zeta(x) = 2^x \pi^{x-1} \sin(\tfrac{1}{2}\pi x)\Gamma(1-x)\zeta(1-x),$$

a formula which is particularly useful when $x < 0$.]

9. Yes; in fact, $\overline{C}_N = N - N^{-\theta}H_{N-1}^{(-\theta)} = \theta N/(1+\theta) + \frac{1}{2} + O(N^{-\theta})$.

10. $p_1 \le \cdots \le p_N$; maximum $\overline{C}_N = (N+1) - (\text{minimum } \overline{C}_N)$. [Similarly in the unequal-length case, the maximum average search time is $L_1(1+p_1) + \cdots + L_N(1+p_N)$ minus the minimum average search time.]

11. (a) The terms of $f_{m-1}(x_{i_1}, \ldots, x_{i_{m-1}})p_i$ are just the probabilities of the possible sequences of requests that could have preceded, leaving R_i in position m. (b) The second identity comes from summing $\binom{n}{m}$ cases of the first, noting the number of times each P_{nk} occurs. The third identity is a consequence of the second, by inversion. [Alternatively, the principle of inclusion and exclusion could be used.] (c) $\sum_{m\ge 0} mP_{nm} = nQ_{nn} - Q_{n,n-1}$; hence

$$d_i = 1 + (N-1) - p_i \sum_{j\ne i} 1/(p_i + p_j);$$

$$\sum p_i d_i = N - \sum_{i<j} (p_i^2 + p_j^2)/(p_i + p_j)$$

$$= N - \sum_{i<j} (p_i + p_j - 2p_ip_j/(p_i + p_j)) = (17).$$

12. $\overline{C}_N = 2^{1-N} + 2\sum_{0\le n\le N-2} 1/(2^n+1)$, which converges rapidly to $2\alpha' \approx 2.5290$; exercise 5.2.4–13 gives α' to 40 decimal places.

13. After evaluating the rather tedious sum

$$\sum_{1\le i\le n} i^2 H_{n+i} = \tfrac{1}{6}n(n+1)(2n+1)(2H_{2n} - H_n) - \tfrac{1}{36}n(10n^2 + 9n - 1),$$

we obtain the answer

$$\overline{C}_N = \tfrac{4}{3}N - \tfrac{2}{3}(2N+1)(H_{2N} - H_N) + \tfrac{5}{6} - \tfrac{1}{3}(N+1)^{-1} \approx .409N.$$

[It would be very interesting to know the maximum factor by which (17) can exceed the optimum result (3).]

14. We may assume that $x_1 \le x_2 \le \cdots \le x_n$; then the maximum value occurs when $y_{a_1} \le y_{a_2} \le \cdots \le y_{a_n}$, and the minimum when $y_{a_1} \ge \cdots \ge y_{a_n}$, by an argument like that of Theorem S.

15. Arguing as in Theorem S, the arrangement $R_1R_2 \ldots R_N$ is optimum iff

$$P_1/L_1(1-P_1) \ge \cdots \ge P_N/L_N(1-P_N).$$

16. The expected time $T_1 + p_1T_2 + p_1p_2T_3 + \cdots + p_1p_2 \ldots p_{N-1}T_N$ is minimized iff $T_1/(1-p_1) \le \cdots \le T_N/(1-p_N)$. [*BIT* **3** (1963), 255–256.]

17. Do the jobs in order of increasing deadlines, independent of the respective times T_i! [Of course in practice some jobs are more important than others, and we may want to minimize the maximum *weighted* tardiness. Or we may wish to minimize the sum $\sum_{1\le i\le n} \max(T_{a_1} + \cdots + T_{a_i} - D_{a_i}, 0)$. Neither of these problems appears to have a simple solution.]

18. Let $h = 1$ if s is present, 0 if s is absent. Let $A = \{i \mid q_i < r_i\}$, $B = \{i \mid q_i = r_i\}$, $C = \{i \mid q_i > r_i\}$, $D = \{j \mid t_j > 0\}$; then the sum $\sum_{1\le i,j\le N} p_i p_j d_{|i-j|}$ for the (q, r) arrangement minus the corresponding sum for the (q', r') arrangement is equal to

$$2 \sum_{i\in A, j\in C} (q_i - r_i)(q_j - r_j)(d_{|i-j|} - d_{h+1+2k-i-j}) + 2 \sum_{i\in C, j\in D} (q_i - r_i)t_j(d_{h+2k-i+j} - d_{i-1+j}).$$

This is positive unless $C = \emptyset$ or $A \cup D = \emptyset$. The desired result now follows because the organ pipe arrangements are the only permutations which are not improved by this construction and its left-right dual when $m = 0, 1$.

[This result is essentially due to G. H. Hardy, J. E. Littlewood, and G. Pólya, *Proc. London Math. Soc.* (2), **25** (1926), 265–282, who showed, in fact, that the minimum of $\sum p_i q_j d_{|i-j|}$ is achieved, under all independent rearrangements of the p's and q's, when both p's and q's are in a consistent organ-pipe order. For further commentary and generalizations, see their book *Inequalities* (Cambridge University Press, 1934), Chapter 10).]

19. All arrangements are equally good. We have

$$\sum p_i p_j\, d(i, j) = \tfrac{1}{2} \sum p_i p_j (d(i, j) + d(j, i)) = \tfrac{1}{2}c.$$

[The special case $d(i, j) = 1 + (j - i) \bmod N$ is due to K. E. Iverson, *A Programming Language* (New York: Wiley, 1962), 138. R. L. Baber, *JACM* **10** (1963), 478–486, has studied some other problems associated with tape searching when a tape can read forward, rewind, or backspace k blocks without reading. W. D. Frazer observes that it would be possible to make significant reductions in the search time if we were allowed to *replicate* some of the information in the file; cf. E. B. Eichelberger et. al., *IBM J. Research & Development* **12** (1968), 130–139, for an empirical solution to a similar problem.]

20. Going from (q, r) to (q', r') as in exercise 18, with $m = 0$ or $m = h = 1$, gives a net change of

$$\sum_{i\in A, j\in C} (q_i - r_i)(q_j - r_j)\big(d_{|i-j|} - \min(d_{h+1+2k-i-j}, d_{i+j-1})\big),$$

which is positive unless A or C is $\emptyset$. By circular symmetry it follows that the only optimal arrangements are cyclic shifts of the organ pipe configurations. [For a different problem with the same answer, see Alfred Lehman, *Israel J. Math* **1** (1963), 22–28.]

21. This problem was essentially first solved by L. H. Harper, *J. SIAM* **12** (1964), 131–135. For generalizations and references to other work, see *J. Applied Probability* **4** (1967), 397–401.

22. A priority queue of size 1000 (represented as, say, a heap, see Section 5.2.3).

Enter the first 1000 records into this queue, with the element of *greatest* $d(K_i, K)$ at the front. For each subsequent K_i with $d(K_i, K) < d$(front of queue, K), replace the front element by R_i.

SECTION 6.2.1

1. Prove inductively that $K_{l-1} < K < K_{u+1}$ whenever we reach step B2; and that $l \le i \le u$ whenever we reach B3.

2. (a, c) No, it loops if $l = u$ and $K > K_u$. (b) Yes, it would!

3. This is Algorithm 6.1T with $N = 3$. In a successful search, that algorithm makes $(N+1)/2$ comparisons, on the average; in a unsuccessful search it makes $N/2 + 1 - 1/(N+1)$.

4. It must be an unsuccessful search with $N = 127$; hence by Theorem B the answer is $138u$.

5. Program 6.1Q′ has an average running time of $1.75N + 8.5 - (N \bmod 2)/4N$; this beats Program B iff $N \le 44$. [It beats Program C only for $N \le 11$.]

7. (a) Certainly not. (b) The parenthesized remarks in Algorithm U will hold true, so it will work, but only if $K_0 = -\infty$ and $K_{N+1} = +\infty$ are both present when N is odd.

8. (a) N. It is interesting to prove this by induction, observing that exactly one of the DELTA's increases if we replace N by $N+1$. [See *AMM* 77 (1970), 884 for a generalization.] (b) Maximum $= \sum$DELTA$[j] = N$; minimum $=$ 2DELTA[1] $- \sum$DELTA$[j] = N \bmod 2$.

9. Consider the corresponding tree (e.g. Fig. 6): When N is odd, the left subtree of the root is a mirror image of the right subtree, so $K < K_i$ occurs just as often as $K > K_i$; on the average, $C1 = \frac{1}{2}(C+S)$ and $C2 = \frac{1}{2}(C-S)$, $A = \frac{1}{2}(1-S)$. When N is even, the tree is the same as the tree for $N+1$ with all labels decreased by 1, except that (0) becomes redundant; on the average,

$$C1 = \tfrac{1}{2}(C+1) - \tfrac{1}{2}\lfloor \log_2 N \rfloor / N,$$
$$C2 = \tfrac{1}{2}(C-1) + \tfrac{1}{2}\lfloor \log_2 N \rfloor / N, \qquad A = 0 \quad \text{for} \quad S = 1;$$

$$C1 = \frac{1}{2}\,\frac{N}{N+1}\,(\lfloor \log_2 N \rfloor + 1),$$

$$C2 = \frac{1}{2}\,\frac{N+2}{N+1}\,(\lfloor \log_2 N \rfloor + 1), \qquad A = \frac{1}{2}\,\frac{N}{N+1} \quad \text{for} \quad S = 0.$$

(The average value of C is stated in the text.)

10. Iff $N = 2^k - 1$.

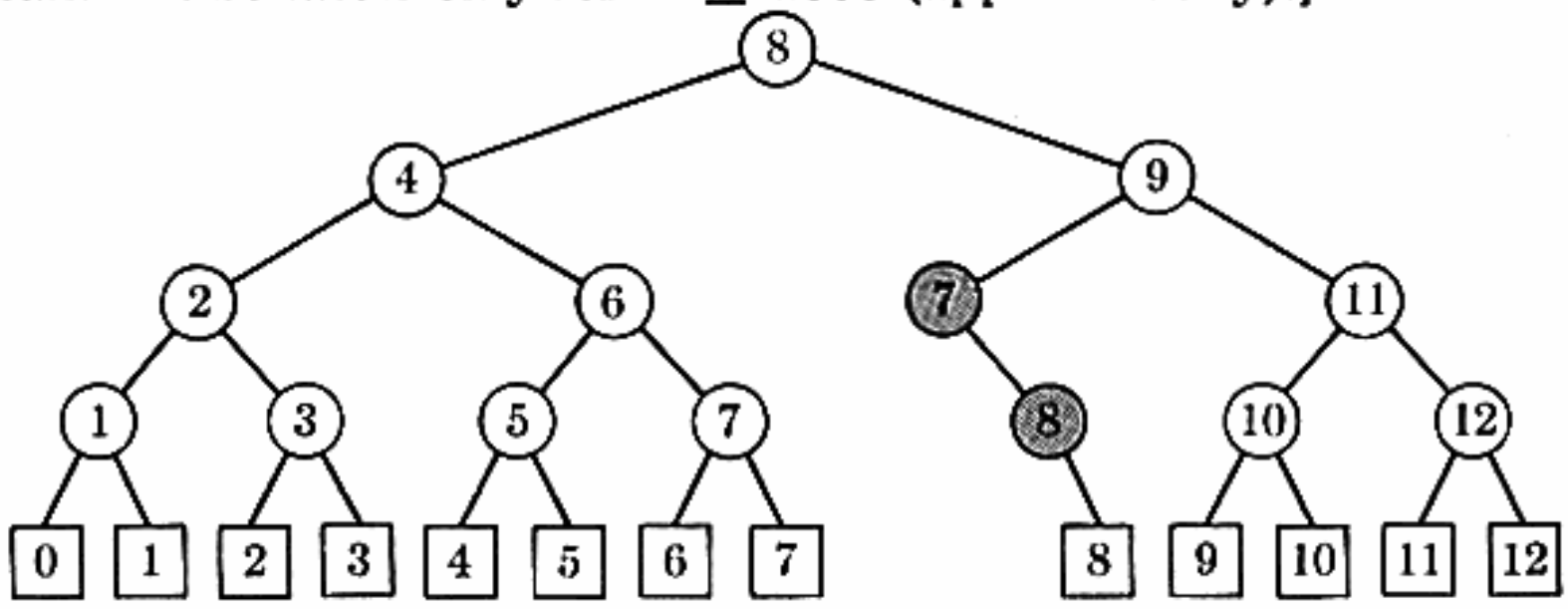
11. Use a "macro-expanded" program with the DELTA's included; e.g., for $N = 10$:

```
START   ENT1   5
        LDA    K
        CMPA   KEY,1
        JL     C3A
```

```
C4A     JE      SUCCESS         C3A     EQU     *
        INC1    3                       DEC1    3
        CMPA    KEY,1                   CMPA    KEY,1
        JL      C3B                     JGE     C4B
C4B     JE      SUCCESS         C3B     EQU     *
        INC1    1                       DEC1    1
        CMPA    KEY,1                   CMPA    KEY,1
        JL      C3C                     JGE     C4C
C4C     JE      SUCCESS         C3C     EQU     *
        INC1    1                       DEC1    1
        CMPA    KEY,1                   CMPA    KEY,1
        JE      SUCCESS                 JE      SUCCESS
        JMP     FAILURE                 JMP     FAILURE
```

[Exercise 23 shows that most of the "JE" instructions may be eliminated, yielding a program about $6 \log_2 N$ lines long which takes only about $4 \log_2 N$ units of time; but that program will be faster only for $N \geq 1000$ (approximately).]

12.



13.

$N =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$C_N =$	1	$1\frac{1}{2}$	$1\frac{2}{3}$	$2\frac{1}{4}$	$2\frac{1}{5}$	$2\frac{2}{6}$	$2\frac{3}{7}$	$3\frac{1}{8}$	3	3	3	$3\frac{2}{12}$	$3\frac{3}{13}$	$3\frac{3}{14}$	$3\frac{4}{15}$	$4\frac{1}{16}$
$C'_N =$	1	$1\frac{2}{3}$	2	$2\frac{3}{5}$	$2\frac{4}{6}$	3	3	$3\frac{6}{9}$	$3\frac{6}{10}$	$3\frac{8}{11}$	$3\frac{8}{12}$	4	4	4	4	$4\frac{13}{17}$

14. One idea is to find the least $M \geq 0$ such that $N + M$ has the form $F_{k+1} - 1$, then to start with $i \leftarrow F_k - M$ in step F1, and to insert "If $i \leq 0$, go to F4" at the beginning of F2. A better solution would be to adapt Shar's idea to the Fibonaccian case: If the result of the very first comparison is $K > K_{F_k}$, set $i \leftarrow i - M$ and go to F4 (proceeding normally from then on). This avoids extra time in the inner loop.

15. The Fibonacci tree of order k, with left and right reversed, is the binary tree corresponding to the lineal chart up to the kth month, under the "natural correspondence" of Section 2.3.2, if we remove the topmost node of the lineal chart.

16. The external nodes appear on levels $\lfloor k/2 \rfloor$ through $k - 1$; the difference between these levels is greater than unity except when $k = 0, 1, 2, 3, 4$.

17. Let the path length be $k - A(n)$; then $A(F_j) = j$ and $A(F_j + m) = 1 + A(m)$ when $0 < m < F_{j-1}$.

18. Successful search: $A_k = 0$, $C_k = (3kF_{k+1} + (k-4)F_k)/5(F_{k+1} - 1) - 1$, $C1_k = C_{k-1}$. Unsuccessful search: $A'_k = F_k/F_{k+1}$, $C'_k = (3kF_{k+1} + (k-4)F_k)/5F_{k+1}$, $C1'_k = C'_{k-1} + F_{k-1}/F_{k+1}$. $C2 = C - C1$. (Cf. exercise 1.2.8–12.)

20. (a) $p^{-p}q^{-q}$. (b) There are at least *two* errors. The first blunder is that division

is not a linear function, so it can't be simply "averaged over." Actually with probability p we get pN elements remaining, and with probability q we get qN, so we can expect to get $(p^2+q^2)N$; thus the average "reduction factor" is really $1/(p^2+q^2)$. Now the reduction factor after k iterations is $1/(p^2+q^2)^k$, but we cannot conclude that $b = 1/(p^2+q^2)$ since the number of iterations needed to locate some of the items is much more than to locate others. This is a second fallacy. [It is very easy to make plausible but fallacious probability arguments, and we must always be on our guard against such pitfalls!]

21. It's impossible, since the method depends on the key values.

23. Go to the right on $\geq$, to the left on $<$; when reaching node $\boxed{i}$ it follows from (1) that $K_i \leq K < K_{i+1}$, so a final test for equality will distinguish between success or failure. (The key $K_0 = -\infty$ should always be present.)

Algorithm C would be changed to go to C4 if $K = K_i$ in step C2. In C3 if DELTA[j] = 0, set $i \leftarrow i - 1$ and go to C5. In C4 if DELTA[j] = 0, go directly to C5. Add a new step C5: "If $K = K_i$, the algorithm terminates successfully, otherwise it terminates unsuccessfully." [This would not speed up Program C unless N were extremely large.]

24. The keys can be arranged so that we first set $i \leftarrow 1$, then $i \leftarrow 2i$ or $2i+1$ according as $K < K_i$ or $K > K_i$; the search is unsuccessful when $i > N$. For example when $N = 12$ the necessary key arrangement is

$$K_8 < K_4 < K_9 < K_2 < K_{10} < K_5 < K_{11} < K_1 < K_{12} < K_6 < K_3 < K_7.$$

When programmed for MIX this method will take only about $6 \log_2 N$ units of time, so it is faster than Program C. The only disadvantage is that it is a little tricky to set up the table in the first place.

25. (a) Since $a_0 = 1 - b_0$, $a_1 = 2a_0 - b_1$, $a_2 = 2a_1 - b_2$, etc., we have $A(z) + B(z) = 1 + 2zA(z)$. Several of the formulas derived in Section 2.3.4.5 follow immediately from this relation by considering $A(1)$, $B(1)$, $B(\frac{1}{2})$, $A'(1)$, $B'(1)$. If we use two variables to distinguish left and right steps of a path we obtain the more general result $A(x, y) + B(x, y) = 1 + (x+y)A(x, y)$, a special case of a formula which holds in t-ary trees [cf. R. M. Karp, *IRE Trans.* **IT–7** (1961), 27–38]. (b) var $(g) = ((N+1)/N)$ var $(h) - ((N+1)/N^2)$ mean $(h)^2 + 2$.

26. The merge tree for the three-tape polyphase merge with a perfect level k distribution is the Fibonacci tree of order $k+1$ if we permute left and right appropriately. (Redraw the Polyphase $T = 3$, $S = 13$ picture of Section 5.4.4, with the left and right subtrees of A, C reversed, obtaining Fig. 8.)

27. At most $k+1$ of the 2^k outcomes will ever occur, since we may order the indices such that $K_{i_1} < K_{i_2} < \cdots < K_{i_k}$. Thus the search can be described by a tree with at most $(k+1)$-way branching at each node. The number of items which can be found on the mth step is at most $k(k+1)^{m-1}$; hence the average number of comparisons is at least N^{-1} times the sum of the smallest N elements of the multiset $\{k \cdot 1, k(k+1) \cdot 2, k(k+1)^2 \cdot 3, \ldots\}$. When $N \geq (k+1)^n - 1$, the average number of comparisons is $\geq N^{-1} \sum_{1 \leq m \leq n} k(k+1)^{m-1} m > n - 1/k$.

1. Use a header node, with say ROOT ≡ RLINK(HEAD); start the algorithm at step T4 with P ← HEAD. Step T5 is to act as if $K >$ KEY(HEAD). [Thus change lines 04 and 05 of Program T to "ENT1 ROOT; CMPA K".]

2. In step T5, set RTAG ← —. Also, when inserting to the left, set RLINK(Q) ← P; when inserting to the right, set RLINK(Q) ← RLINK(P). [If nodes are inserted into successively increasing locations Q, and if all deletions are last-in-first-out, the RTAG fields can be eliminated since RTAG(P) will be — if and only if RLINK(P) $<$ P.]

3. We could replace Λ by a valid address, and set KEY(Λ) ← K at the beginning of the algorithm; then the tests for LLINK or RLINK = Λ could be removed from the inner loop. However, in order to do a proper insertion we need to introduce another pointer variable which follows P; this can be done without losing the stated speed advantage, by duplicating the code as in Program 6.2.1F. Thus the MIX time would be reduced to about $5.5C$.

4. $C_N = 1 + (0\cdot 1 + 1\cdot 2 + \cdots + (n-1)2^{n-1} + C'_{2^n} + \cdots + C'_{N-1})/N = (1 + 1/N)C'_N - 1$, for $N \geq 2^n - 1$. The solution to these equations is $C'_N = 2(H_{N+1} - H_{2^n}) + n$ for $N \geq 2^n - 1$, a savings of $2H_{2^n} - n - 2 \approx n(\ln 4 - 1)$ comparisons. The actual improvement for $n = 1, 2, 3, 4$ is, respectively, 0, $\frac{1}{6}$, $\frac{61}{140}$, $\frac{274399}{360360}$; thus comparatively little is gained for small fixed n. [See Frazer and McKellar, *JACM* **17** (1970), 502, for a more detailed derivation related to an equivalent sorting problem.]

5. (a) The first element must be CAPRICORN; then we multiply the number of ways to produce the left subtree by the number of ways to produce the right subtree, times $\binom{10}{3}$, the number of ways to shuffle those two sequences together. Thus the answer comes to

$$\binom{10}{3}\binom{2}{0}\binom{1}{0}\binom{0}{0}\binom{6}{3}\binom{2}{0}\binom{1}{0}\binom{0}{0}\binom{2}{1}\binom{0}{0}\binom{0}{0} = 4800.$$

[In general, the answer is the product, over all nodes, of $\binom{l+r}{r}$, where l and r stand for the sizes of the left and right subtrees of the node. This is equal to $N!$ divided by the product of the subtree sizes. It is same formula as in exercise 5.1.4-20; indeed, there is an obvious one-to-one correspondence between the permutations which yield a particular search tree and the "topological" permutations counted in that exercise, if we replace a_k in the search tree by k (using the notation of exercise 6).] (b) $2^{N-1} = 1024$; at each step but the last, insert either the smallest or largest remaining key.

6. (a) For each of the P_{nk} permutations $a_1 \ldots a_{n-1}a_n$ whose "score" is k, construct $n+1$ permutations $a'_1 \ldots a'_{n-1} m\, a'_n$, where $a'_j = a_j$ or $a_j + 1$, according as $a_j < m$ or $a_j \geq m$. [Cf. Section 1.2.5, Method 2.] If $m = a_n$ or $a_n + 1$, this permutation has a "score" of $k+1$, otherwise it has a score of k. (b) $G_n(z) = (2z + n - 2)(2z + n - 3) \ldots (2z)$. Hence

$$P_{nk} = \begin{bmatrix} n-1 \\ k \end{bmatrix} 2^k.$$

This generating function was, in essence, obtained by W. C. Lynch, *Comp. J.* **7** (1965), 299–302. (c) The generating function for probabilities is $g_n(z) = G_n(z)/n!$.

This is a product of simple probability generating functions, so

$$\begin{aligned}\operatorname{var}(g_n) &= \sum_{0\le k\le n-2} \operatorname{var}\left((2z+k)/(2+k)\right)\\ &= \sum_{0\le k\le n-2} \left(2/(k+2) - 4/(k+2)^2\right) = 2H_n - 4H_n^{(2)} + 2.\end{aligned}$$

[From exercise 6.2.1–25b we can therefore compute the variance of C_n, which is $(2+10/n)H_n - 4(1+1/n)(H_n^{(2)}/n + H_n^2) + 2$; this formula is due to G. D. Knott (unpublished).]

7. Set up a recurrence like that in the answer to exercise 5.2.2–32, but with $C_{nm} = 1 + (A_{nm} + B_{nm})/n$. The solution is $C_{nm} = H_m + H_{n+1-m} - 1$. [*CACM* **12** (1969), 77–80].

8. (a) $g_n(z) = z^{n-1}\sum_{1\le i\le n} g_{i-1}(z)g_{n-i}(z)/n$, $g_0(z) = 1$. (b) $7n^2 - 4(n+1)^2H_n^{(2)} - 2(n+1)H_n + 13n$. [P. F. Windley, *Comp. J.* **3** (1960), 86, gave recurrence relations from which this variance could be computed numerically, but he did not obtain the solution. Note that this result is *not* simply related to the variance of C_n stated in the answer to exercise 6.]

10. For example, each word x of the key could be replaced by $(ax) \bmod m$, where m is the computer word size and a is a random multiplier relatively prime to m. A value near to $(\phi - 1)m$ can be recommended, see Section 6.4. The flexible storage allocation of tree methods may make them more attractive than other hash coding schemes.

11. $N - 2$, but this occurs with probability $1/(N \cdot N!)$, only in the deletion (1) N $(N-1)\ldots 2$.

12. $\frac{1}{2}(n+1)(n+2)$ of the deletions in the proof of Theorem H belong to Case 1, so the answer is $(N+1)/2N$.

13. Yes, in fact the proof of Theorem H shows that if we delete the ith element inserted, for any fixed i, the result is random. [Conjecture: After an arbitrary sequence of random insertions and *first-in-first-out* deletions, the result is random.]

14. Let `NODE(T)` be on level k, and let `LLINK(T)` $= \Lambda$, `RLINK(T)` $=$ `R`$_1$, `LLINK(R`$_1$`)` $=$ `R`$_2$, $\ldots$, `LLINK(R`$_d$`)` $= \Lambda$, where `R`$_d \neq \Lambda$ and $d \ge 1$. Let `NODE(R`$_i$`)` have n_i nodes in its right subtree, for $1 \le i \le d$. With step $D1\frac{1}{2}$ the internal path length decreases by $k + d + n_1 + \cdots + n_d$; without that step it decreases by $k + d + n_d$.

16. Yes; even the deletion operation on permutations, as defined in the proof of Theorem H, is commutative (if we omit the renumbering aspect). If there is an element between X and Y, deletion is obviously commutative since the operation is affected only by the relative positions of X, Y, and their successors and there is no interaction between the deletion of X and the deletion of Y. On the other hand, if Y is the successor of X, and Y is the largest element, both orders of deletion have the effect of simply removing X and Y. If Y is the successor of X and Z the successor of Y, both orders of deletion have the effect of replacing the *first* occurrence of X, Y, or Z by Z and deleting the second and third occurrences of these elements within the permutation.

18. Use exercise 1.2.7–14.

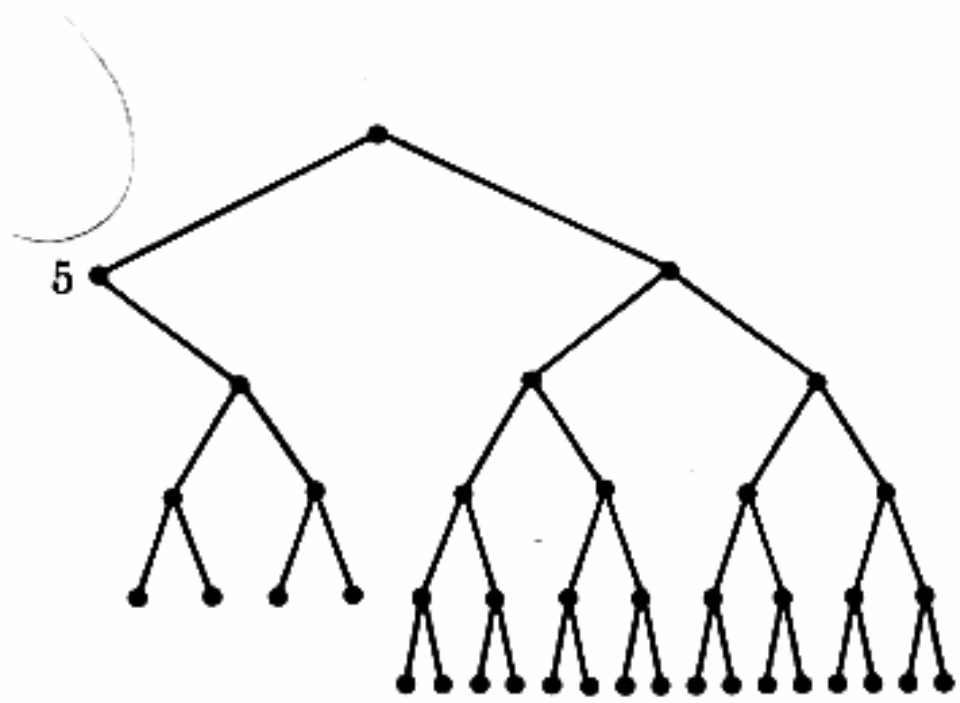
19. $2H_N - 1 - 2\sum_{1\le k\le N}(k-1)^\theta/kN^\theta = 2H_N - 1 - 2/\theta + O(N^{-\theta})$.

20. Yes indeed. Assume that $K_1 < \cdots < K_N$, so that the tree built by Algorithm T is degenerate; if, say, $p_k = (1 + ((N+1)/2 - k)\epsilon)/N$, the average number of comparisons is $(N+1)/2 - (N^2-1)\epsilon/12$, while the optimum tree requires less than $\lceil \log_2 N \rceil$ comparisons.

21. $\frac{1}{8}, \frac{3}{20}, \frac{9}{20}, \frac{3}{20}, \frac{1}{8}$. (Most of the angles are 30°, 60°, or 90°.)

22. This is obvious when $d = 2$, and for $d > 2$ we had $r[i, j-1] \le r[i+1, j-1] \le r[i+1, j]$.

23.



[Increasing the weight of the first node will eventually make it move to the root position; this suggests that dynamically maintaining a perfectly optimum tree is hard.]

24. Let c be the cost of a tree obtained by deleting the nth node of an optimum tree. Then $c(0, n-1) \le c \le c(0, n) - q_{n-1}$, since the deletion operation always moves $\boxed{n-1}$ up one level. Also $c(0, n) \le c(0, n-1) + q_{n-1}$, since the stated replacement yields a tree of the latter cost. It follows that $c(0, n-1) = c = c(0, n) - q_{n-1}$.

25. (a) Assume that $A \le B$ and $B \le C$, and let $a \in A$, $b \in B$, $c \in C$, $c < a$. If $c \le b$ then $c \in B$; hence $c \in A$ and $a \in B$; hence $a \in C$. If $c > b$, then $a \in B$; hence $a \in C$ and $c \in B$; hence $c \in A$. (b) Not hard to prove.

26. The cost of every tree has the form $y + lx$ for some real $y \ge 0$ and integer $l > 0$. The minimum of a finite number of such functions (taken over all trees) always has the form described.

27. (a) The answer to exercise 24 (especially the fact that $c = c(0, n-1)$) implies that $R(0, n-1) = R(0, n)\backslash\{n\}$. (b) If $l = l'$ the result in the hint is trivial. Otherwise let the respective paths to $\boxed{n}$ be

$$\textcircled{r_1}, \textcircled{r_2}, \ldots, \boxed{r_l} \qquad \text{and} \qquad \textcircled{s_1}, \textcircled{s_2}, \ldots, \boxed{s_{l'}}.$$

Since $r = r_1 > s_1 = s$ and $r_{l'} < s_{l'} = n$, we can find a level $k \ge 1$ such that $r_k > s_k$ and $r_{k+1} \le s_{k+1}$. We have $r_{k+1} \in R(r_k, n)$, $s_{k+1} \in R(s_k, n)$, and $R(s_k, n) \le R(r_k, n)$ by induction, hence $r_{k+1} \in R(s_k, n)$ and $s_{k+1} \in R(r_k, n)$; the right subtrees of $\textcircled{r_k}$ and $\textcircled{s_k}$ can be interchanged, and the result in the hint follows.

Now to show that $R'_h \le R_h$, let $r \in R'_h$, $s \in R_h$, $s < r$, and consider the optimum trees shown when $x = x_h$; we must have $l \ge l_h$ and we may assume that $l' = l_h$. To show that $R_h \le R'_{h+1}$, let $r \in R_h$, $s \in R'_{h+1}$, $s < r$, and consider the optimum trees shown when $x = x_{h+1}$; we must have $l' \le l_h$ and we may assume that $l = l_h$.

29. It is a degenerate tree with YOU at the top, THE at the bottom, needing 19.158 comparisons on the average.

Douglas A. Hamilton has proved that some degenerate tree is always worst. Therefore an $O(n^2)$ algorithm exists to find pessimal binary search trees.

31. (a) 30; (b) 31. Thus there are trees built up according to rule (i) which have less cost than trees for which Phase 3 will succeed.

32. Suppose that we form $q_i + q_j$ at one step, and $q_{i'} + q_{j'} < q_i + q_j$ at the next step. Then the combination $q_{i'} + q_{j'}$ must have been blocked at the first step by having external node $\boxed{q_i}$ or $\boxed{q_j}$ (or both) between $q_{i'}$ and $q_{j'}$. If both, we must have $q_{i'} > q_i$, $q_{j'} \geq q_j$; otherwise we must have $q_{i'} > q_j$, $q_{j'} > q_i$.

35. Let $i < j$; C_i is the leading $e_i + 1$ bits of x_i and C_j is the leading $e_j + 1$ bits of $x_j \geq x_i + 2^{-e_i-1} + 2^{-e_j-1}$. Therefore if $e_i \leq e_j$, C_i cannot be a prefix of C_j; if $e_i > e_j$, C_j cannot be a prefix of C_i.

36. In fact it is less than $2S + \sum_{0\leq i\leq n} (q_i + x_i + y_i) \log_2 \big(S/(q_i + x_i + y_i)\big)$, where the x_i and y_i are *any* nonnegative values such that $p_i = x_i + y_{i-1}$ and $x_0 = y_n = 0$. For if $\textcircled{i}$ appears at level l_i and $\boxed{j}$ at level l'_j, we have $l'_i > l_i$ and $l'_{i-1} > l_i$ in any tree. Hence, constructing the tree of Theorem G with weights $q_i + x_i + y_i$, we have $\sum p_i(l_i + 1) + \sum q_j l'_j \leq \sum (q_j + x_j + y_j) l'_j$. [The best bound is obtained when the $q_j + x_j + y_j$ are farthest from uniformity.]

38. Let $a_r = \sum_{i\in S(r)} c_i$; then $a_{r+1} - a_r \geq a_r - a_{r-1}$ for $1 \leq r < \lceil n/2 \rceil$. (Cf. exercise 32.) Hence $a_r \leq r/\lceil n/2 \rceil$, for all r; this follows since the relations $a_r > r/\lceil n/2 \rceil$ and $a_{r-1} \leq (r-1)/\lceil n/2 \rceil$ imply $1 \geq a_{\lceil n/2 \rceil} = a_r + \sum_{r\leq k<\lceil n/2 \rceil} (a_{k+1} - a_k) \geq a_r + (\lceil n/2 \rceil - r)(a_r - a_{r-1}) > r/\lceil n/2 \rceil + (\lceil n/2 \rceil - r)/\lceil n/2 \rceil = 1$. The proof is completed by using the Hu-Tucker algorithm on the given "worst" probabilities: the resulting tree has $n + 1 - 2^q$ pairs of external nodes on level $q + 1$, and all other external nodes on level q, so $f(n) = q(n + 1 - 2^q)/\lceil n/2 \rceil + (q-1)\big(1 - (n+1-2^q)/\lceil n/2 \rceil\big)$. This is at least as great as the cost of T, which has a weight of $a_{n+1-2^q} \leq (n + 1 - 2^q)/\lceil n/2 \rceil$ on level q. [*Acta Informatica* 2 (1972), to appear.]

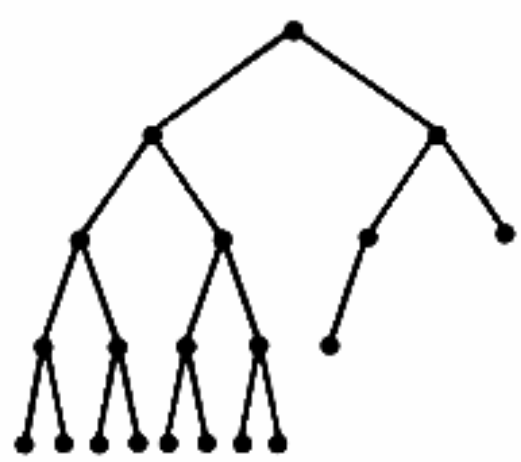
SECTION 6.2.3

1. The symmetric order of nodes must be preserved by the transformation, otherwise we wouldn't have a binary search tree.

2. B(S) = 0 can happen only when S points to the root of the tree (it has never been changed in steps A3 or A4), and all nodes from S to the point of insertion were balanced.

3. Let ρ_h be the largest possible ratio of unbalanced nodes in balanced trees of height h. Thus $\rho_1 = 0$, $\rho_2 = \frac{1}{2}$, $\rho_3 = \frac{1}{2}$. We will prove that $\rho_h = (F_{h+1} - 1)/(F_{h+2} - 1)$. Let T_h be a tree which maximizes ρ_h; then we may assume that its left subtree has height $h - 1$ and its right subtree has height $h - 2$, for if both subtrees had height $h - 1$ the ratio would be less than ρ_{h-1}. Thus the ratio for T_h is at most $(\rho_{h-1}N_l + \rho_{h-2}N_r + 1)/(N_l + N_r + 1)$, where there are (N_l, N_r) nodes in the (left, right) subtree. This formula takes its maximum value when (N_l, N_r) take their minimum values; hence we may assume that T_h is a Fibonacci tree.

4. When $n = 7$,



has greater path length. [Note: C. C. Foster, *Proc. ACM Nat. Conf.* **20** (1965), 197–198, gave an incorrect procedure for constructing N-node balanced trees of maximum path length; Edward Logg has observed that Foster's Fig. 3 gives a nonoptimal result after 24 steps (node number 22 can be removed in favor of number 25).]

5. This can be proved by induction; if T_N denotes the tree constructed, we have

$T_N =$  , if $2^n \le N < 2^n + 2^{n-1}$;

$T_N =$ $T_{2^{n-1}-1}$ $T_{N-2^{n-1}}$  T_{2^n-1} T_{N-2^n} , if $2^n + 2^{n-1} \le N < 2^{n+1}$.

6. The coefficient of z^n in $zB_i(z)B_j(z)$ is the number of n-node binary trees whose left subtree is a balanced binary tree of height i and whose right subtree is a balanced binary tree of height j.

7. $C_{n+1} = C_n^2 + 2B_{n-1}B_{n-2}$; hence if we let $\alpha_0 = \ln 2$, $\alpha_1 = 0$, and $\alpha_{n+2} = \ln(1 + 2B_{n+1}B_n/C_{n+2}^2) = O(1/B_nC_{n+2})$, and $\theta = \exp(\alpha_0/2 + \alpha_1/4 + \alpha_2/8 + \cdots)$, we find that $0 < \theta^{2^n} - C_n = C_n(\exp(\alpha_n/2 + \alpha_{n+1}/4 + \cdots) - 1) < 1$, i.e., $C_n = \lfloor \theta^{2^n} \rfloor$.

8. Let $b_h = B_h'(1)/B_h(1) + 1$, and let ϵ_h be the very small quantity $2B_hB_{h-1}(b_h - b_{h-1})/B_{h+1}$. Then $b_h = 2b_{h-1} - \epsilon_h$; hence $b_h = 2^h(1 - \epsilon_1/2 - \epsilon_2/4 - \cdots) + r_h$, where $r_h = \epsilon_{h+1}/2 + \epsilon_{h+2}/4 + \cdots$ is extremely small for large h. [*Zh. vychisl. matem. i matem. fiziki* **6**,2 (1966), 389–394.]

11. Good reason: A balanced tree with more than one internal node has precisely as many −A external nodes as it has −+B and ++B, and as many +A's as +−B's and −−B's.

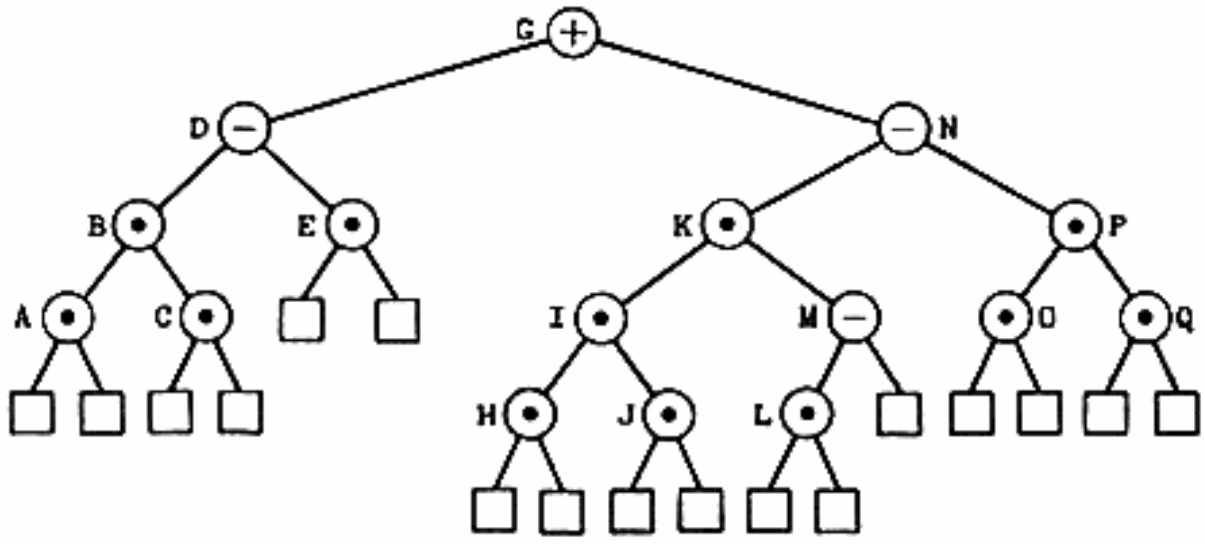
12. The maximum occurs when inserting into the second external node of (12); $C = 4$, $C1 = 3$, $D = 3$, $A = 1$, $F = G = H = 1$, $E = J = 0$, and the `JMP` in line 67 is taken, for a total time of $132u$. The minimum occurs when inserting into the third-last external node of (13); $C = 2$, $C1 = 1$, $D = 2$, $A = E = F = G = H = J = 0$, for a total time of $61u$. [The corresponding figures for Program 6.2.2T are $74u$ and $26u$.]

13. When the tree changes, only $O(\log N)$ `RANK` values need to be updated; the "simple" system might require very extensive changes.

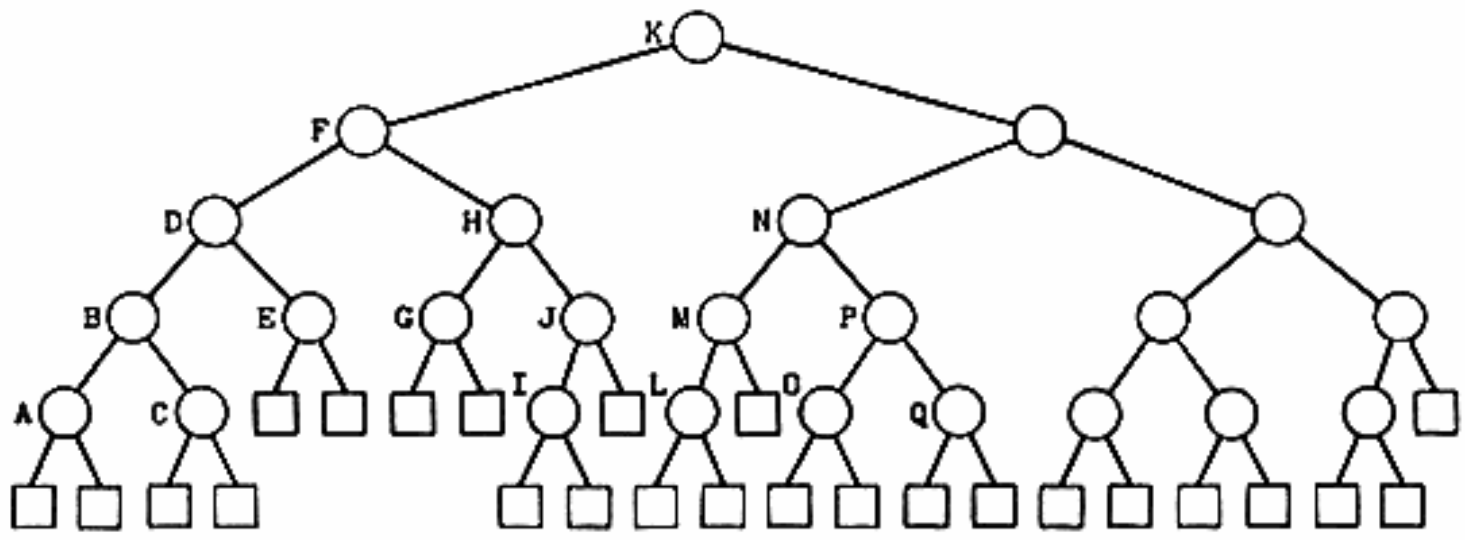
14. Yes. (But typical operations on lists are sufficiently nonrandom that degenerate trees would probably occur.)

15. Use Algorithm 6.2.2T with `M` set to zero in step T1, and `M` ← `M` + `RANK(P)` whenever $K \ge$ `KEY(P)` in step T2.

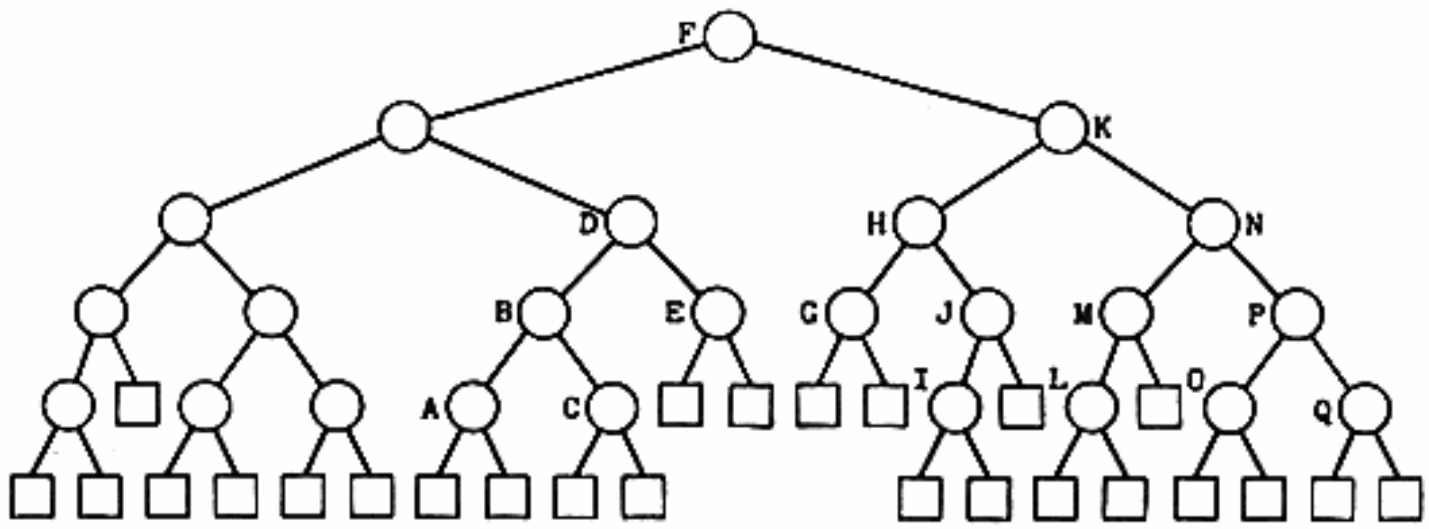
16. Delete G; replace F by G; Case 2 rebalancing at H; Case 3 rebalancing at K.



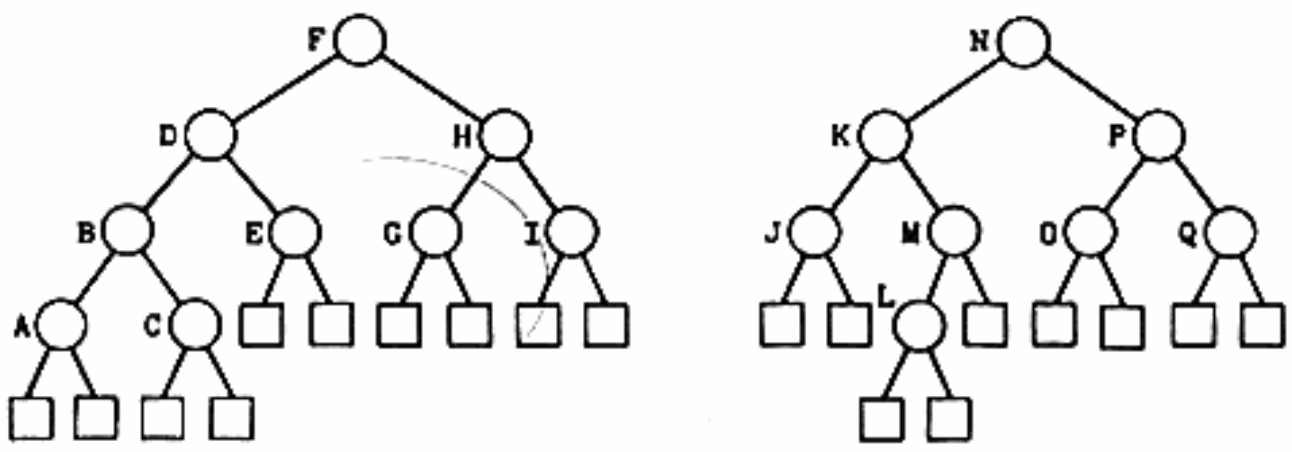
17. (a)



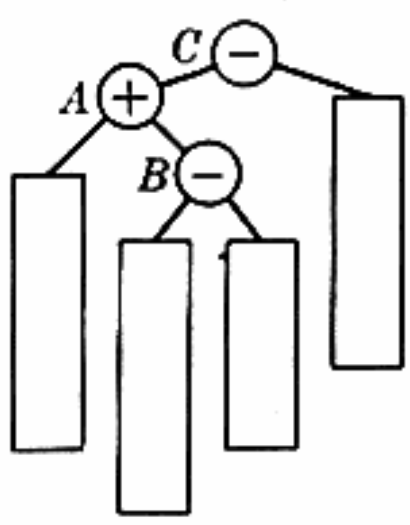
(b)



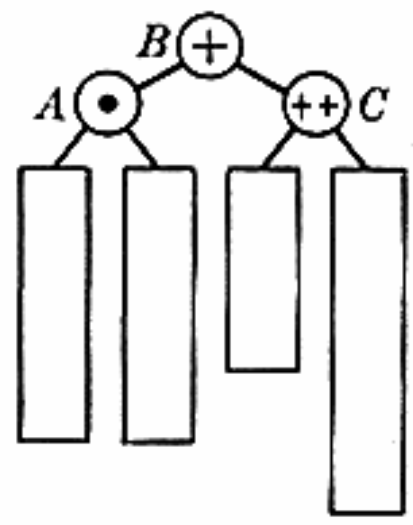
18.



19. (Solution by Clark Crane.) There is one case which can't be handled by a single or double rotation at the root, namely

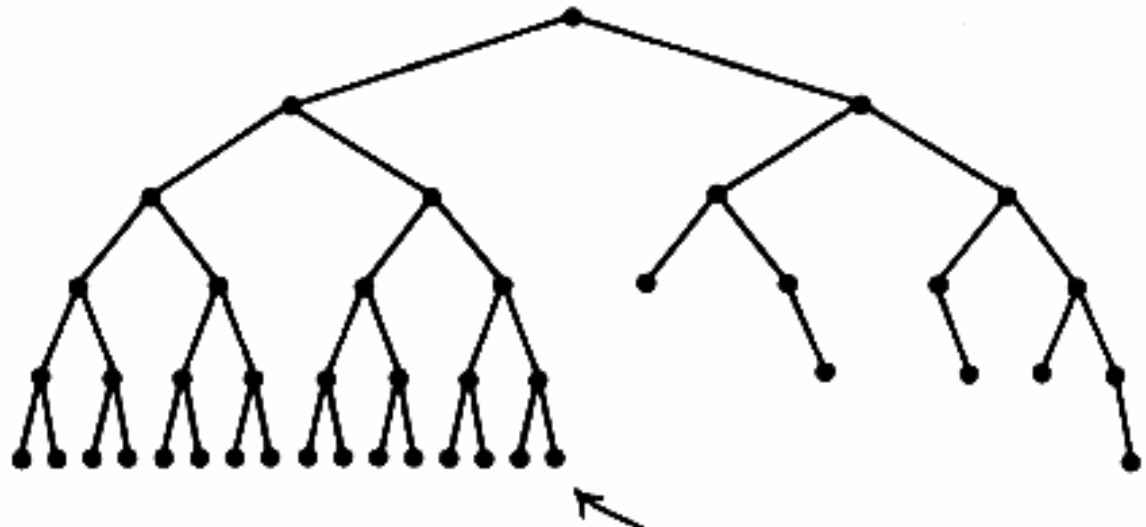


. Change it to



and then resolve the imbalance by applying a single or double rotation at *C*.

20.



It may be difficult to insert a new node here ↖ .

21. Algorithm A does the job in order $N \log N$ steps (see exercise 5); the following algorithm creates the same trees in $O(N)$ steps, using an interesting iterative rendition of a recursive method. We use three auxiliary lists:

$\mathtt{D}_1, \ldots, \mathtt{D}_l$ (a binary counter that essentially controls the recursion);

$\mathtt{J}_1, \ldots, \mathtt{J}_l$ (a list of pointers to "juncture nodes");

$\mathtt{T}_1, \ldots, \mathtt{T}_l$ (a list of pointers to trees).

Here $l = \lceil \log_2 (N+1) \rceil$. For convenience the algorithm also sets $\mathtt{D}_0 \leftarrow 1$, $\mathtt{J}_0 \leftarrow \mathtt{J}_{l+1} \leftarrow \Lambda$.

G1. [Initialize.] Set $l \leftarrow 0$, $\mathtt{J}_0 \leftarrow \mathtt{J}_1 \leftarrow \Lambda$, $\mathtt{D}_0 \leftarrow 1$.

G2. [Get next item.] Let P point to the next input node. (We may invoke another coroutine in order to obtain P.) If there is no more input, go to G5. Otherwise, set $k \leftarrow 1$, $\mathtt{Q} \leftarrow \Lambda$, and interchange $\mathtt{P} \leftrightarrow \mathtt{J}_1$.

G3. [Carry.] If $k > l$ (equivalently if $\mathtt{P} = \Lambda$), set $l \leftarrow l+1$, $\mathtt{D}_k \leftarrow 1$, $\mathtt{T}_k \leftarrow \mathtt{Q}$, $\mathtt{J}_{k+1} \leftarrow \Lambda$, and return to G2. Otherwise set $\mathtt{D}_k \leftarrow 1 - \mathtt{D}_k$, interchange $\mathtt{Q} \leftrightarrow \mathtt{T}_k$, $\mathtt{P} \leftrightarrow \mathtt{J}_{k+1}$, and increase k by 1. If now $\mathtt{D}_{k-1} = 0$, repeat this step.

G4. [Concatenate.] Set $\mathtt{LLINK(P)} \leftarrow \mathtt{T}_k$, $\mathtt{RLINK(P)} \leftarrow \mathtt{Q}$, $\mathtt{B(P)} \leftarrow 0$, $\mathtt{T}_k \leftarrow \mathtt{P}$, and return to G2.

G5. [Finish up.] Set $\mathtt{LLINK}(\mathtt{J}_k) \leftarrow \mathtt{T}_k$, $\mathtt{RLINK}(\mathtt{J}_k) \leftarrow \mathtt{J}_{k-1}$, $\mathtt{B}(\mathtt{J}_k) \leftarrow 1 - \mathtt{D}_{k-1}$, for $1 \le k \le l$. Then terminate the algorithm ($\mathtt{J}_l$ points to the root of the desired tree). ▮

Step G3 is executed $2N - \nu(N)$ times, where $\nu(N)$ is the number of 1's in the binary representation of N.

22. The height of a weight-balanced tree with N internal nodes always lies between $\log_2 (N+1)$ and $2 \log_2 (N+1)$. To get this upper bound, note that the heavier subtree of the root has at most $(N+1)/\sqrt{2}$ external nodes.

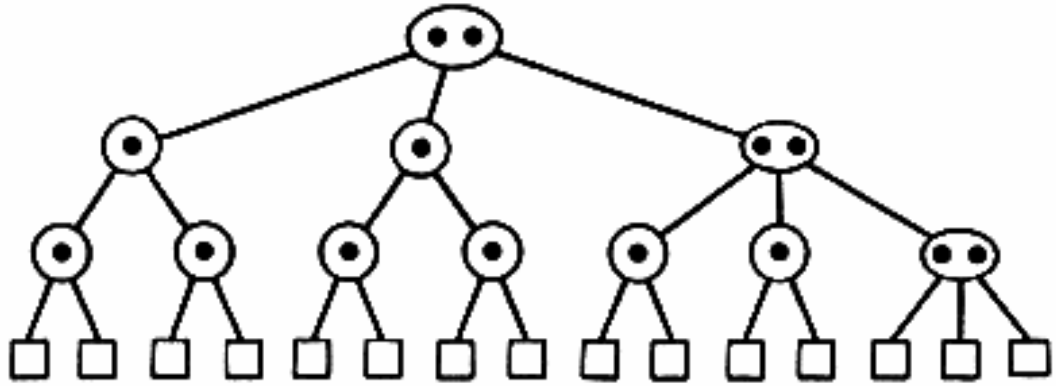
23. (a) Form a tree whose right subtree is a complete binary tree with $2^n - 1$ nodes, and whose left subtree is a Fibonacci tree with $F_{n+1} - 1$ nodes. (b) Form a weight balanced tree whose right subtree is about $2 \log_2 N$ levels high and whose left subtree is about $\log_2 N$ levels high (cf. exercise 22).

24. Consider a smallest tree which satisfies the condition but is not perfectly balanced. Then its left and right subtrees are perfectly balanced, so they have 2^l and 2^r external nodes, respectively, where $l \neq r$. But this contradicts the stated condition.

25. After inserting a node at the bottom of the tree, we work up from the bottom to check the weight balance at each node on the search path. Suppose imbalance occurs at node A in (1), after having inserted a new node in the right subtree, where B and its subtrees are weight-balanced. Then a single rotation will restore the balance unless

$(|\alpha| + |\beta|)/|\gamma| > \sqrt{2} + 1$, where $|x|$ denotes the number of external nodes in a tree x. But in this case it can be shown that a double rotation will suffice.

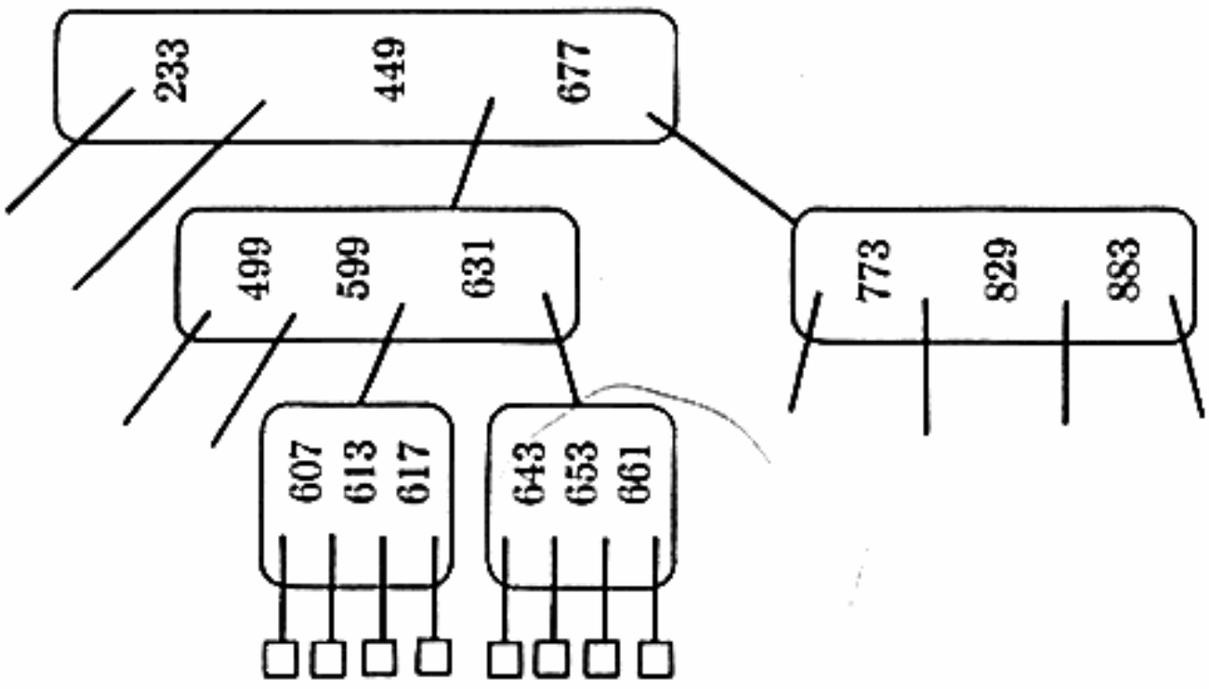
27. It is sometimes necessary to make two comparisons in nodes that contain two keys. The worst case occurs in a tree like the following, which sometimes needs $2 \log_2 (N + 2) - 2$ comparisons:



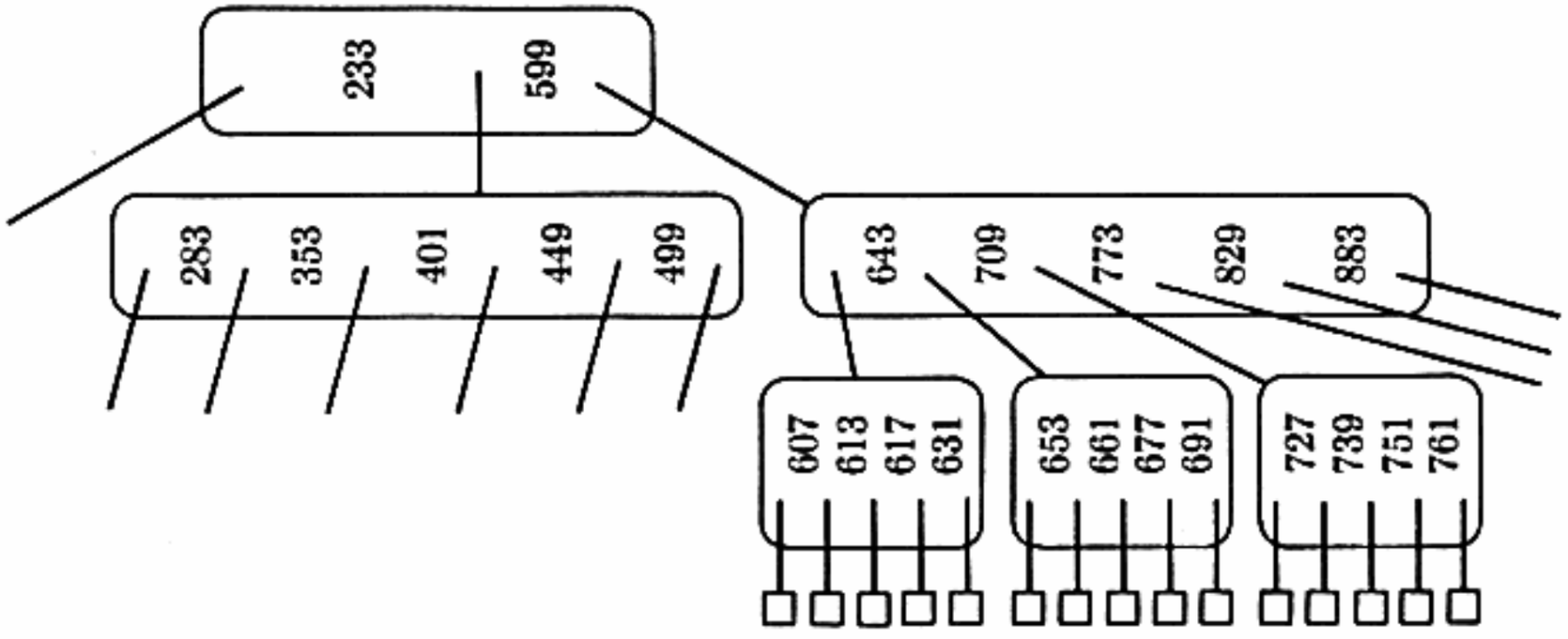
30. For best-fit, arrange the records in order of size, with an arbitrary rule to break ties in case of equality. (Cf. exercise 2.5–9.) For first fit, arrange the records in order of location, with an extra field in each node telling the size of the largest area in the left subtree of that node. This extra field can be maintained under insertions and deletions. (Although the running time is $O(\log n)$, it probably still doesn't beat the "ROVER" method of exercise 2.5–6 in practice.)

SECTION 6.2.4

1. Two nodes split:



2. Altered nodes:



(Of course a B^*-tree would have no 3-key nodes, although Fig. 30 does.)

3. (a) $1 + 2 \cdot 50 + 2 \cdot 51 \cdot 50 + 2 \cdot 51 \cdot 51 \cdot 50 = 2 \cdot 51^3 - 1 = 265301$. (b) $1 + 2 \cdot 50 + (2 \cdot 51 \cdot 100 - 100) + ((2 \cdot 51 \cdot 101 - 100) \cdot 100 - 100) = 101^3 = 1030301$. (c) $1 + 2 \cdot 66 + (2 \cdot 67 \cdot 66 + 2) + (2 \cdot 67 \cdot 67 \cdot 66 + 2 \cdot 67) = 601661$. (Less than (b)!)

4. Before splitting a non-root node, make sure that it has two full brothers, then split these three nodes into four. The root should split only when it has more than $3\lfloor(3m - 3)/4\rfloor$ keys.

5. 450. The worst case occurs if we have 1005 characters and the key to be passed to Father must be 50 characters long: 445 chars + ptr + 50-char key + ptr + 50-char key + ptr + 445 chars.

6. Yes, for example we could replace each K_i in (1) by i plus the number of keys in subtrees $P_0, \ldots, P_{i-1}$. The search, insertion, and deletion algorithms can be modified appropriately.

7. If the key to be deleted is not on level $l - 1$, replace it by its successor and delete the successor. To delete a key on level $l - 1$, we simply erase it; if this makes the node too empty, we look at its right (or left) brother, and "underflow," i.e., move keys in from the brother so that both nodes have approximately the same amount of data. This underflow operation will fail only if the brother was minimally full, but in that case the two nodes can be collapsed into one (together with one key from their father); such a collapsing may cause the father in turn to underflow, etc.

9. Brief sketch: Extend the paging scheme so that exclusive access to buffers is given to one user at a time; the search, insertion, and deletion algorithms must be carefully modified so that such exclusive access is granted only for a limited time when absolutely necessary, and such that no deadlocks can occur. When a particular user needs to update a father page, it will be necessary for him to see if it has changed or moved since he last observed it.

10. Given a tree $\mathfrak{I}$ with N internal nodes, let there be $a_k^{(j)}$ external nodes that require k accesses and whose father node belongs to a page containing j keys; and let $A^{(j)}(z)$ be the corresponding generating function. Thus $A^{(1)}(1) + \cdots + A^{(M)}(1) = N + 1$. (Note that $a_k^{(j)}$ is a multiple of $j + 1$, for $1 \le j < M$.) The next random insertion leads to $N + 1$ equally probable trees, whose generating functions are obtained by decreasing some coefficient $a_k^{(j)}$ by $j + 1$ and adding $j + 2$ to $a_k^{(j+1)}$; or (if $j = M$) by decreasing some $a_k^{(M)}$ by 1 and adding 2 to $a_{k+1}^{(1)}$. Now $B_N^{(j)}(z)$ is $(N + 1)^{-1}$ times the sum, over all $\mathfrak{I}$, of the generating function $A^{(j)}(z)$ for $\mathfrak{I}$ times the probability that $\mathfrak{I}$ occurs; the stated recurrence relations follow.

The recurrence has the form

$$\begin{aligned}(B_N^{(1)}(z), \ldots, B_N^{(M)}(z))^T &= (I + (N + 1)^{-1}\mathbf{W}(z))(B_{N-1}^{(1)}(z), \ldots, B_{N-1}^{(M)}(z))^T \\ &= \cdots = g_N(\mathbf{W}(z))(0, \ldots, 0, 1)^T,\end{aligned}$$

where

$$g_n(x) = (1 + x/(n + 1)) \cdots (1 + x/2) = \binom{x+n+1}{n+1}/(x + 1).$$

It follows that $C'_N = (1, \ldots, 1)(B_N^{(1)\prime}(1), \ldots, B_N^{(M)\prime}(1))^T = 2B_{N-1}^{(M)}(1)/(N + 1) + C'_{N-1} = 2f_N(\mathbf{W})_{MM}$, where $f_n(x) = g_{n-1}(x)/(n + 1) + \cdots + g_0(x)/2 = (g_n(x) - 1)/x$, and $\mathbf{W} = \mathbf{W}(1)$. (The subscript MM denotes the lower right corner element of the matrix.) Now $\mathbf{W} = \mathbf{S}^{-1}\,\mathbf{diag}\,(\lambda_1, \ldots, \lambda_M)\mathbf{S}$, for some matrix $\mathbf{S}$, where $\mathbf{diag}(\lambda_1, \ldots, \lambda_M)$ denotes the diagonal matrix whose entries are the roots of $\chi(\lambda) = (\lambda + 2) \ldots$

$(\lambda+M+1)-(M+1)!$. (These roots are distinct, since $\chi(\lambda)=\chi'(\lambda)=0$ implies $1/(\lambda+2)+\cdots+1/(\lambda+M+1)=0$; the latter can hold only when λ is real, and $-M-1<\lambda<-2$, which implies that $|\lambda+2|\ldots|\lambda+M+1|<(M+1)!$, a contradiction.) If $p(x)$ is any polynomial, $p(\mathbf{W})=p(\mathbf{S}^{-1}\,\mathbf{diag}\,(\lambda_1,\ldots,\lambda_M)\mathbf{S})=\mathbf{S}^{-1}\,\mathbf{diag}\,(p(\lambda_1),\ldots,p(\lambda_M))\mathbf{S}$; hence the lower right corner element of $p(\mathbf{W})$ has the form $c_1p(\lambda_1)+\cdots+c_Mp(\lambda_M)$ for some constants $c_1,\ldots,c_M$ independent of p. These constants may be evaluated by setting $p(\lambda)=\chi(\lambda)/(\lambda-\lambda_i)$; since $(\mathbf{W}^k)_{MM}=(-2)^k$ for $0\le k\le M-1$, we have $p(\mathbf{W})_{MM}=p(-2)=(M+1)!/(\lambda_i+2)=c_ip(\lambda_i)=c_i\chi'(\lambda_i)=c_i(M+1)!\big(1/(\lambda_i+2)+\cdots+1/(\lambda_i+M+1)\big)$; hence $c_i=(\lambda_i+2)^{-1}\big((1/(\lambda_i+2)+\cdots+1/(\lambda_i+M+1)\big)^{-1}$. This yields an "explicit" formula $C'_N=\sum_{1\le i\le M}2c_if_N(\lambda_i)$; and it remains to study the roots λ_i. Note that $|\lambda_i+M+1|\le M+1$ for all i, otherwise we would have $|\lambda_i+2|\ldots|\lambda_i+M+1|>(M+1)!$. Taking $\lambda_1=0$, this implies that $\Re(\lambda_i)<0$ for $2\le i\le M$. By Eq. 1.2.5–15, $g_n(x)\sim(n+1)^x/\Gamma(x+2)$ as $n\to\infty$; hence $g_n(\lambda_i)\to0$ for $2\le i\le M$. Consequently $C'_N=2c_1f_N(0)+O(1)=H_N/(H_{M+1}-1)+O(1)$.

Notes: The above analysis is relevant also to the "samplesort" algorithm discussed briefly in Section 5.2.2. The calculations may readily be extended to show that $B_N^{(j)}(1)\sim(H_{M+1}-1)^{-1}/(j+2)$ for $1\le j<M$, $B_N^{(M)}(1)\sim(H_{M+1}-1)^{-1}/2$. Hence the total number of interior nodes on unfilled pages is approximately $N\big(1/(3\times2)+2/(4\times3)+\cdots+(M-1)/((M+1)\times M)\big)/(H_{M+1}-1)=N\big(1-M/(M+1)(H_{M+1}-1)\big)$; and the total number of pages used is approximately

$$N\big(1/(3\times2)+1/(4\times3)+\cdots+1/((M+1)\times M)+1/(M+1)\big)/(H_{M+1}-1)$$
$$=N/2(H_{M+1}-1),$$

yielding an asymptotic storage utilization of $2(H_{M+1}-1)/M$.

SECTION 6.3

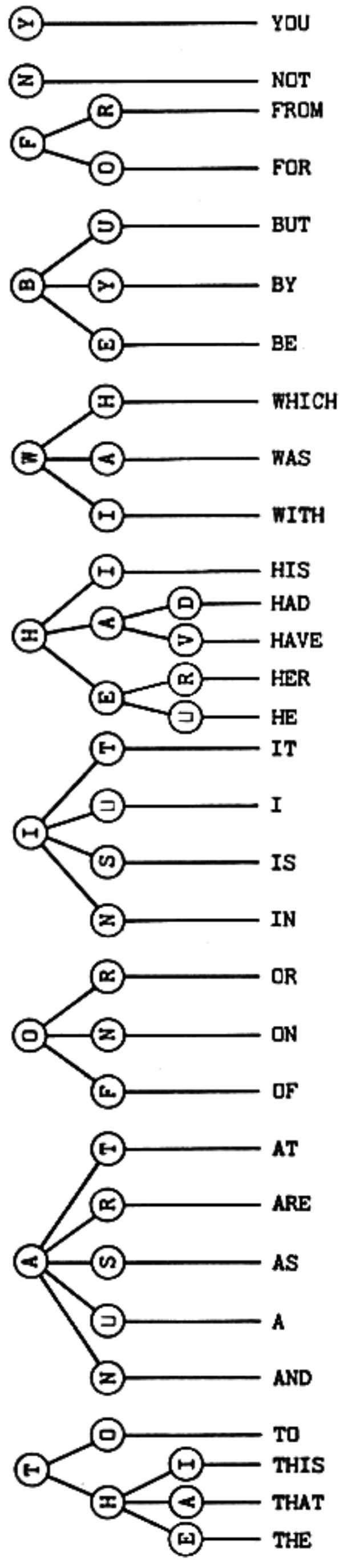
1. Lieves (pl. of "lief").

2. Perform Algorithm T using the new key as argument; it will terminate unsuccessfully in either step T3 or T4. If in T3, simply set table entry k of NODE(P) to K and terminate the insertion algorithm. Otherwise set this table entry to the address of a new node Q $\Leftarrow$ AVAIL, containing only null links, then set P $\leftarrow$ Q. Now set k and k' to the respective next characters of K and X; if $k \neq k'$, store K in position k of NODE(P) and store X in position k', but if $k = k'$ again make the k position point to a new node Q $\Leftarrow$ AVAIL, set P $\leftarrow$ Q, and repeat the process until eventually $k \neq k'$. (We must assume that no key is a prefix of another.)

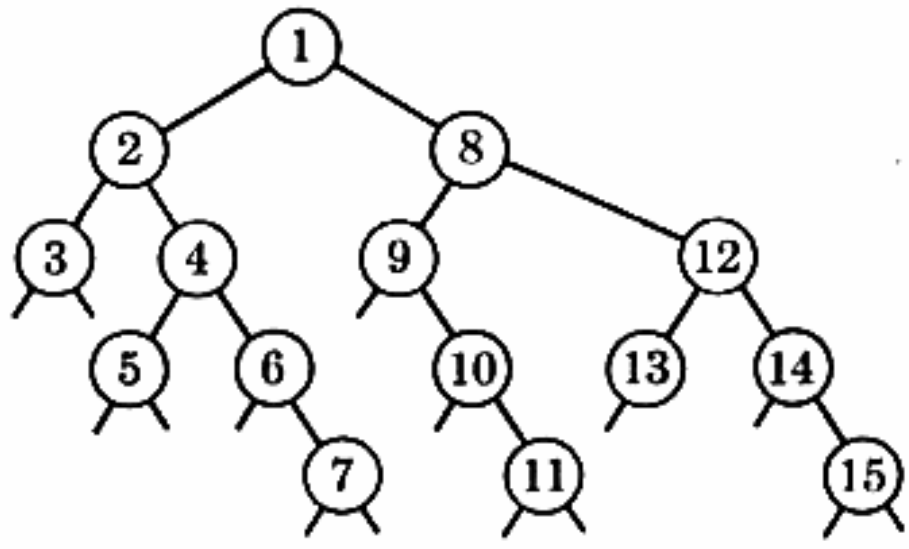
3. Replace the key by a null link, in the node where it appears. If this node is now "useless," i.e. if all its entries are null except one which is a key X, delete the node and replace the corresponding pointer in its father by X. If the father node is now useless, delete it in the same way.

4. Successful searches take place exactly as with the full table, but unsuccessful searches in the compressed table may go through several additional iterations. For example, an input argument such as IONIC will make Program T take *seven* iterations (more than five!); this is the worst case. It is necessary to verify that no infinite looping on blank sequences is possible.

5. In each family, test for the most probable outcome first, by arranging the letters from left to right in decreasing order of probability. The optimality of this arrangement can be proved as in Theorem 6.1S. [Cf. *CACM* **12** (1969), 72–76.]



6.



7. For example, 8, 4, 12, 1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14, 15. (No matter what sequence is used, the left subtree cannot contain more than two nodes on level 4, nor can the right subtree.) Even this "worst" tree is within 4 of the best possible tree, so we see that digital search trees aren't very sensitive to the order of insertion.

8. Yes. The KEY fields now contain only a truncated key; leading bits implied by the node position are chopped off. (A similar modification of Algorithm T is possible.)

9.

START	LDX	K	1	*D1. Initialize.* (rX ≡ K)
	LD1	ROOT	1	P ← ROOT. (rI1 ≡ P)
	JMP	2F	1	
4H	LD2	0,1(RLINK)	$C2$	*D4. Move right.* Q ← RLINK(P).
	J2Z	5F	$C2$	To D5 if Q = Λ.
1H	ENT1	0,2	$C - 1$	P ← Q.
2H	CMPX	1,1	C	*D2. Compare.*
	JE	SUCCESS	C	Exit if K = KEY(P).
	SLB	1	$C - S$	Shift K left one bit.
	JAO	4B	$C - S$	To D4 if the detached bit was 1.
	LD2	0,1(LLINK)	$C1$	*D3. Move left.* Q ← LLINK(P).
	J2NZ	1B	$C1$	To D2 with P ← Q if Q ≠ Λ.
5H	Continue as in Program 6.2.2T, interchanging the roles of rA, rX.			

The running time for the searching phase of this program is $(10C - 3S + 4)u$, where $C - S$ is the number of bit inspections. For random data, the approximate average running times are therefore:

	Successful	Unsuccessful
Program 6.2.2T	$15 \ln N - 12.34$	$15 \ln N - 2.34$
This program	$14.4 \ln N - 6.17$	$14.4 \ln N + 1.26$

(Consequently Program 6.2.2T is a shade faster unless N is very large.)

10. Let $\oplus$ denote the "exclusive or" operation on n-bit numbers, and let $f(x) = n - \lceil \log_2 (x + 1) \rceil$, the number of leading zero bits of x. One solution: (b) If a search via Algorithm T ends unsuccessfully in step T3, k is one less than the number of bit inspections made so far; otherwise if the search ends in step T4, $k = f(K \oplus X)$. (a, c) Do a regular search, but also keep track of the minimum value, x, of $K \oplus$ KEY(P) over all KEY(P) compared with K during the search. Then $k = f(x)$. (Prove that no other key can have more bits in common with K than those compared to K. In case (a), the maximum k occurs for either the largest key $\leq K$ or the smallest key $> K$.)

11. Yes, since the keys of all nodes in the subtree of NODE(Q) begin with the appropriate sequence of bits. But major changes to the tree would be necessary if keys are

truncated as in exercises 8 and 9, so truncation should not be used together with deletion.

12. Insert three random numbers α, β, γ between 0 and 1 into an initially empty tree; then delete α with probability p, β with probability q, γ with probability r, using Algorithm 6.2.2D. The tree

is obtained with probability $\frac{3}{8}p + \frac{1}{2}q + \frac{1}{2}r$, and this is $\frac{1}{2}$ only if $p = 0$.

13. Add a KEY field to each node, and compare K with this key before looking at the vector element in step T2. Table 1 would change as follows: Nodes (1), . . . , (12) would contain the respective keys THE, AND, BE, FOR, HIS, IN, OF, TO, WITH, HAVE, HE, THAT (if we inserted them in order of decreasing frequency), and these keys would be deleted from their previous positions. [The corresponding program would be slower and more complicated than Program T, in this case. A more direct M-ary generalization of Algorithm D would create a tree with N nodes, having one key and M links per node.]

14. If $j \leq n$, there is only one place, namely KEY(P). But if $j > n$, the set of all occurrences is found by traversing the subtree of node P: If there are r occurrences, this subtree contains $r - 1$ nodes (including node P), and so it has r link fields with TAG = 1; these link fields point to all the nodes that reference TEXT, positions matching K. (It isn't necessary to check the TEXT again at all.)

15. To begin forming the tree, set KEY(HEAD) to the first TEXT reference, and LLINK(HEAD) $\leftarrow$ HEAD, LTAG(HEAD) $\leftarrow 1$. Further TEXT references can be entered into the tree using the following insertion algorithm:

Set K to the new key which we wish to enter. (This is the first reference the insertion algorithm makes to the TEXT array.) Perform Algorithm P; it must terminate unsuccessfully, since no key is allowed to be a prefix of another. (Step P6 makes the second reference to the TEXT; no more references will be needed!) Now suppose that the key located in step P6 agrees with the argument K in the first l bits, but differs from it in position $l + 1$, where K has the digit b and the key has $1 - b$. (Even though the search in Algorithm P might have let j get much greater than l, it is possible to prove that the procedure specified here will find the longest match between K and *any* existing key. Thus, *all* keys of the text which start with the first l bits of K have $1 - b$ as their $(l + 1)$st bit.) Now repeat Algorithm P with K replaced by these leading l bits (i.e., $n \leftarrow l$). This time the search will be successful, so we needn't perform step P6. Now set R $\Leftarrow$ AVAIL, KEY(R) $\leftarrow$ position of the new key in TEXT. If LLINK(Q) = P, set LLINK(Q) $\leftarrow$ R, $l \leftarrow$ LTAG(Q), LTAG(Q) $\leftarrow 0$; otherwise set RLINK(Q) $\leftarrow$ R, $l \leftarrow$ RTAG(Q), RTAG(Q) $\leftarrow 0$. If $b = 0$, set LTAG(R) $\leftarrow 1$, LLINK(R) $\leftarrow$ R, RTAG(R) $\leftarrow l$, RLINK(R) $\leftarrow$ P; otherwise set RTAG(R) $\leftarrow 1$, RLINK(R) $\leftarrow$ R, LTAG(R) $\leftarrow l$, LLINK(R) $\leftarrow$ P. If $l = 1$, set SKIP(R) $\leftarrow 1 + l - j$; otherwise set SKIP(R) $\leftarrow 1 + l - j +$ SKIP(P) and SKIP(P) $\leftarrow j - l - 1$.

16. The tree setup requires precisely one dotted link coming from below a node to that node; it comes from that part of the tree where this key first differs from all others. If there is no such part of the tree, the algorithms break down. We could simply drop keys that are prefixes of others, but then the algorithm of exercise 14 wouldn't have enough data to find *all* occurrences of the argument.

17. If we define $a_0 = a_1 = 0$, then

$$x_n = a_n + \sum_{k\geq 2} \binom{n}{k} (-1)^k \hat{a}_k/(m^{k-1} - 1) = \sum_{k\geq 2} \binom{n}{k} (-1)^k \hat{a}_k m^{k-1}/(m^{k-1} - 1).$$

18. To solve (4) we need the transform of $a_n = 1 - \delta_{n0} - \delta_{n1}$, namely $\hat{a}_n = \delta_{n0} - 1 + n$; hence for $N \geq 2$ we obtain $A_N = 1 - U_N + V_N$, where $U_N = K(N, 0, M)$ and $V_N = K(N, 1, M)$ in the notation of exercise 19. Similarly to solve (5), take $a_n = n - \delta_{n1} = \hat{a}_n$ and obtain $C_N = N + V_N$ for $N \geq 2$.

19. For $s = 1$, we have $V_n = K(n, 1, m) = n\big((\ln n + \gamma)/(\ln m) - \frac{1}{2} - f_0(n-1)\big) + O(1)$, and for $s \geq 2$ we have $K(n, s, m) = (-1)^s n\big(1/(\ln m) + f_{s-1}(n-s)\big)/s(s-1) + O(1)$, where

$$f_s(n) = \frac{2}{\ln m} \sum_{k\geq 1} \Re\left(\Gamma(s - 2\pi i k/(\ln m)) \exp(2\pi i k \log_m n)\right)$$

is a periodic function of $\log n$. [In this derivation we have

$$K(n+s, s, m)/(-1)^s \binom{n+s}{s} = (n^{-s+1}/2\pi i) \int_{1/2-i\infty}^{1/2+i\infty} \Gamma(z) n^{s-1-z}\, dz/(m^{s-1-z} - 1) + O(n^{-s}).$$

For small m and s, the f's will be negligibly small; cf. exercise 5.2.2–46. Note that $f_s(n-a) = f_s(n) + O(n^{-1})$ for fixed a.]

20. For (a), let $a_n = 1 - \delta_{n0} - \delta_{n1} - \cdots - \delta_{ns}$; for (b), let $a_n = n - \delta_{n1} - 2\delta_{n2} - \cdots - s\delta_{ns}$; and for (c), we want to solve the recurrence

$$y_n = m^{1-n} \sum_k \binom{n}{k} (m-1)^{n-k} y_k \qquad \text{for} \qquad n > s,$$

$$y_n = \binom{n+1}{2} \qquad \text{for} \qquad n \leq s.$$

Setting $x_n = y_n - n$ yields a recurrence of the form of exercise 17, with

$$a_n = (1 - M^{-1}) \sum_{0\leq k\leq s} \binom{k}{2} \delta_{nk}.$$

Therefore, in the notation of previous exercises, the answers are (a) $1 - K(N, 0, M) + K(N, 1, M) - \cdots + (-1)^{s-1} K(N, s, M) = N/(s \ln M) - N\big(f_{-1}(N) + f_0(n-1) + f_1(N-2)/2\cdot 1 + \cdots + f_{s-1}(N-s)/s(s-1)\big) + O(1)$; (b) $N^{-1}\big(N + K(N, 1, M) - 2K(N, 2, M) + \cdots + (-1)^{s-1} sK(N, s, M)\big) = (\ln N + \gamma - H_{s-1})/(\ln M) + \frac{1}{2} - \big(f_0(N-1) + f_1(N-2)/1 + \cdots + f_{s-1}(N-s)/(s-1)\big) + O(N^{-1})$; (c) $N^{-1}\big(N + (1 - M^{-1})\sum_{2\leq k\leq s} (-1)^k \binom{k}{2} K(N, k, M)\big) = 1 + \frac{1}{2}(1 - M^{-1})\big((s-1)/(\ln M) + f_1(N-2) + \cdots + f_{s-1}(N-s)\big) + O(N^{-1})$.

21. Let there be A_N nodes in all. The number of nonnull pointers is $A_N - 1$, and the number of nonpointers is N, so the total number of null pointers is $MA_N - A_N +$

$1 - N$. To get the average number of these in any fixed position, divide by M. [The average value of A_N appears in exercise 20(a).]

22. There is a node for each of the M^l sequences of leading bits such that at least two keys have this bit pattern. The probability that exactly k keys have a particular bit pattern is

$$\binom{N}{k} M^{-lk}(1 - M^{-l})^{N-k},$$

so the average number of trie nodes on level l is $M^l(1 - (1 - M^{-l})^N) - N(1 - M^{-l})^{N-1}$.

23. More generally, consider the case of arbitrary s as in exercise 20. If there are a_l nodes on level l, they contain a_{l+1} links and $Ma_l - a_{l+1}$ places where the search might be unsuccessful. The average number of digit inspections will therefore be $\sum_{l\geq 0}(l+1)M^{-l-1}(Ma_l - a_{l+1}) = \sum_{l\geq 0} M^{-l}a_l$. Using the formula for a_l in a random trie, this equals $1 + (N+1)^{-1}\big(K(N+1, 1, M) - 2K(N+1, 2, M) + \cdots + (-1)^s (s+1)K(N+1, s+1, M)\big) = (\ln N + \gamma - H_s)/(\ln M) + \frac{1}{2} - f_0(N) - f_1(N-1)/1 - \cdots - f_s(N-s)/s + O(N^{-1})$.

24. We must solve the recurrences $x_0 = x_1 = y_0 = y_1 = 0$,

$$x_n = m^{-n} \sum_{n_1+\cdots+n_m=n} \binom{n}{n_1, \ldots, n_m}\Big(x_{n_1} + \cdots + x_{n_m} + \sum_{1\leq j\leq m}(1 - \delta_{n_j 0})\Big)$$

$$= a_n + m^{1-n}\sum_k \binom{n}{k} x_k,$$

$$y_n = m^{-n} \sum_{n_1+\cdots+n_m=n} \binom{n}{n_1, \ldots, n_m}\Big(y_{n_1} + \cdots + y_{n_m} + \sum_{1\leq i<j\leq m}(1 - \delta_{n_i 0})n_j\Big)$$

$$= b_n + m^{1-n}\sum_k \binom{n}{k} y_k,$$

for $n \geq 2$, where $a_n = m(1 - (1 - 1/m)^n)$ and $b_n = \frac{1}{2}(m-1)n(1 - (1 - 1/m)^{n-1})$. By exercises 17, 18 the answers are (a) $x_N = N + V_N - U_N - \delta_{N1} = A_N + N - 1$ [a result which could have been obtained directly, since the number of nodes in the forest is always $N - 1$ more than the number in the corresponding trie!]; and (b) $y_N/N = \frac{1}{2}(M-1)V_N/N = \frac{1}{2}(M-1)\big((\ln N + \gamma)/(\ln M) - \frac{1}{2} - f_0(N-1)\big) + O(N^{-1})$.

25. (a) Let $A_N = M(N-1)/(M-1) - E_N$; then for $N \geq 2$, we have $(1 - M^{1-N})E_N = M - 1 - M(1 - 1/M)^{N-1} + M^{1-N}\sum_{0<k<N}\binom{N}{k}(M-1)^{N-k}E_k$. Since $M - 1 - M(1 - 1/M)^{N-1} \geq 0$, we have $E_N \geq 0$ by induction. (b) By Theorem 1.2.7A with $x = 1/(M-1)$, $n = N - 1$, we find $D_N = a_N + M^{1-N}\sum_k \binom{N}{k}(M-1)^{N-k}D_k$, where $a_1 = 0$ and $0 < a_N < M(1 - 1/M)^N/\ln M \leq (M-1)^2/M \ln M$ for $N \geq 2$. Hence $0 \leq D_N \leq (M-1)^2 A_N/M \ln M \leq (M-1)(N-1)/\ln M$.

26. Taking $q = \frac{1}{2}$, $z = -\frac{1}{2}$ in the second identity of exercise 5.1.1–16, we get $1/3 - 1/(3\cdot 7) + 1/(3\cdot 7\cdot 15) - \cdots = 0.28879$; it's slightly faster to use $z = -\frac{1}{4}$ and take half of the result. Alternatively, Euler's formula from exercise 5.1.1–14 can be used,

involving only negative powers of 2. (John Wrench has computed the 40D value 0.28878 80950 86602 42127 88997 21929 23078 00889+.)

27. (For fun, the following derivation goes to $O(N^{-1})$.) In the notation of exercises 5.2.2–38, 48, we have $\overline{C}_N = U_N + N - 1 + V_{N+1}/(N+1) - \alpha N - \beta + \sum_{n\geq 2} (-1)^n 2^{-n(n+1)/2} \big(\prod_{1\leq r\leq n} (1-2^{-r})^{-1}\big) \big(\sum_{m\geq 0} (2^{1-n})^m (1-2^{-m})^N\big)$, where $\alpha = 2/(1\cdot 1) - 4/(3\cdot 3\cdot 1) + 8/(7\cdot 7\cdot 3\cdot 1) - \cdots \approx$ 1.60669 51524 15291 76378 33015 23190 92458 04806— and $\beta = 2/(1\cdot 3\cdot 1) - 4/(3\cdot 7\cdot 3\cdot 1) + 8/(7\cdot 15\cdot 7\cdot 3\cdot 1) - \cdots \approx 0.60670$. This numerical evaluation suggests that $\alpha = \beta + 1$, a fact which is not hard to prove. The value of $\sum_{m\geq 0} (2^{1-n})^m (1-2^{-m})^N$ is $O(N^{1-n})$, by exercise 5.2.2–46; and $V_{N+1}/(N+1) = U_{N+1} - U_N$. Hence $\overline{C}_N = U_{N+1} - (\alpha-1)N - \alpha + O(N^{-1}) = (N+1)\log_2(N+1) + N\big((\gamma-1)/\ln 2 + \frac{1}{2} - \alpha + f_{-1}(N)\big) + \frac{1}{2} - 1/\ln 4 - \alpha - \frac{1}{2}f_1(N) + O(N^{-1})$, by exercise 5.2.2–50.

28. The derivations in the text and exercise 27 apply to general $M \geq 2$, if we substitute M for 2 in the obvious places. Hence the average number of digit inspections in a random successful search is $\overline{C}_N/N = U_{N+1} - \alpha_M + 1 + O(N^{-1}) = \log_M N + (\gamma-1)/\ln M + \frac{1}{2} - \alpha_M + f_{-1}(N) + (\log_M N)/N + O(N^{-1})$; and for the unsuccessful case it is $\overline{C}_{N+1} - \overline{C}_N = V_{N+2}/(N+2) - \alpha_M + 1 + O(N^{-1}) = \log_M N + \gamma/\ln M + \frac{1}{2} - \alpha_M - f_0(N+1) + O(N^{-1})$. Here $f_s(n)$ is defined in exercise 19, and $\alpha_M = \sum_{j\geq 0} (-1)^j M^{j+1}/(M^{j+1}-1)^2(M^j-1)\cdots(M-1)$.

30. By iterating the recurrence, $h_n(z)$ is the sum of all possible terms of the form

$$\binom{n}{p_1}(z/(2^{p_1}-1))\binom{p_1}{p_2}(z/(2^{p_2}-1))\cdots(z/(2^{p_m}-1))\binom{p_m}{1},$$

for $n > p_1 > \cdots > p_m > 1$.

31. $h_n'(1) = V_n$ (cf. exercise 5.2.2–36b).

32. The sum of the SKIP fields is the number of nodes in the corresponding binary trie, so the answer is A_N (cf. exercise 20).

33. Here's how (18) was discovered: $A(2z) - 2A(z) = e^{2z} - 2e^z + 1 + A(z)(e^z - 1)$ can be transformed into $A(2z)/(e^{2z}-1) = (e^z-1)/(e^z+1) + A(z)/(e^z-1)$. Hence $A(z) = (e^z-1)\sum_{j\geq 1} (e^{z/2^j}-1)/(e^{z/2^j}+1)$. Now if $f(z) = \sum c_n z^n$, $\sum_{j\geq 1} f(z/2^j) = \sum c_n z^n/(2^n-1)$. In this case $f(z) = (e^z-1)/(e^z+1) = \tanh(z/2) = 1 - 2z^{-1}\times \big(z/(e^z-1) - 2z/(e^{2z}-1)\big) = \sum_{n\geq 1} B_{n+1}\, z^n(2^{n+1}-1)/(n+1)!$. From this formula the route is apparent.

34. (a) Consider $\sum_{j\geq 1}\sum_{2\leq k<n} \binom{n}{k} B_k 2^{j(k-1)}$; $1^{n-1} + \cdots + (m-1)^{n-1} = (B_n(m) - B_n)/n$ by exercise 1.2.11.2–4. (b) Consider $(B_n(m) - B_n)/nm^{n-1}$. (c) Argue as in Section 5.2.2 when $|x| < 2\pi$, then use analytic continuation. (d) $\frac{1}{2}\log_2(n/\pi) + \gamma/(2\ln 2) - \frac{3}{4} + f(n)$, where $f(n) = (2/\ln 2)\sum_{k\geq 1} \Re\big(\zeta(-2\pi ik/\ln 2)\, \Gamma(-2\pi ik/\ln 2)\, \exp(2\pi ik \log_2 n)\big) = (1/\ln 2)\sum_{k\geq 1} \Re\big(\zeta(1 + 2\pi ik/\ln 2)\times \exp(2\pi ik \log_2(n/\pi))/\cosh(\pi^2 k/\ln 2)\big)$.

35. The keys must be $\{\alpha 0\beta 0\omega_1, \alpha 0\beta 1\omega_2, \alpha 1\gamma 0\omega_3, \alpha 1\gamma 1\delta 0\omega_4, \alpha 1\gamma 1\delta 1\omega_5\}$, where $\alpha, \beta, \ldots$ are strings of 0's and 1's with $|\alpha| = a-1$, $|\beta| = b-1$, etc. The probability that five random keys have this form is $5!2^{a-1+b-1+c-1+d-1}/2^{a+b+a+b+a+c+a+c+d+a+c+d} = 5!/2^{4a+b+2c+d+4}$.

36. Let n be the number of internal nodes. (a) $(n!/2^I)\prod\big(1/s(x)\big) = n!\prod\big(1/2^{s(x)-1}s(x)\big)$, where I is the internal path length of the tree.

(b) $((n+1)!/2^n)\prod(1/(2^{s(x)}-1))$. (Consider summing the answer of exercise 35 over all $a, b, c, d \geq 1$.)

37. The smallest modified external path length is actually $2 - 1/2^{N-2}$, and it occurs only in a degenerate tree (whose external path length is *maximal*). [One can prove that the *largest* modified external path length occurs iff the external nodes appear on at most two adjacent levels! But it is not always true that a tree whose external path length is smaller than another has a larger modified external path length.]

38. Consider as subproblems the finding of k-node trees with parameters (α, β), $(\alpha, \frac{1}{2}\beta)$, . . . , $(\alpha, 2^{k-n}\beta)$.

40. Let N/r be the true period length of the sequence. Form a Patricia-like tree, with $a_0a_1\ldots$ as the TEXT and with N/r keys starting at positions 0, 1, . . . , $N/r - 1$. (No key is a prefix of another, because of our choice of r.) Also include in each node a SIZE field, containing the number of tagged link fields in the subtree below that node. To do the specified operation, use Algorithm P; if the search is unsuccessful, the answer is 0, but if it is successful and $j \leq n$ the answer is r. Finally if it is successful and $j > n$, the answer is $r \cdot$ SIZE(P).

SECTION 6.4

1. $-38 \leq$ rI1 ≤ 46. Therefore the locations preceding and following TABLE must be guaranteed to contain no data that matches any given argument, e.g. their first byte could be zero. It would certainly be bad to store K in this range! [In a sense we could say that the method in exercise 6.3–4 uses less space, since the boundaries of that table are never exceeded.]

2. TOW. [Can the reader find ten "common" words of ≤ 5 letters that fill all the remaining gaps between -10 and 30?]

3. The alphabetic codes A + T = I + N and B − E = O − R, so we would have either f(AT) = f(IN) or f(BE) = f(OR). Note that instructions 4 and 5 of Table 1 resolve this dilemma rather well, while keeping rI1 from having too wide a range.

4. The smallest m such that

$$n^{-m} m! \sum_{0 \leq k \leq m} \binom{n}{m-k} \binom{m-k}{k} 2^{-k} < \frac{1}{2}$$

is 88. If you invite 88 people, the chance of a birthday trio is .5111, but if only 87 people come it is lowered to .4995.

5. The hash function is bad since it assumes at most 26 different values, and some of them occur much more often than the others. Even with double hashing (letting $h_2(K) = 1$ plus the second byte of K, say, and $M = 101$) the search will be slowed down more than the time saved by faster hashing. Also $M = 100$ is too small, since FORTRAN programs often have more than 100 distinct variables.

6. Not on MIX, since arithmetic overflow will almost always occur (dividend too large). [It would be nice to be able to compute (wK) mod M, especially if linear probing were being used with $c = 1$, but unfortunately most computers disallow this since the quotient overflows.]

7. If $R(x)$ is a multiple of $P(x)$, then $R(\alpha^j) = 0$ in $GF(2^k)$ for all $j \in S$. Let $R(x) = x^{a_1} + \cdots + x^{a_s}$, where $a_1 > \cdots > a_s \geq 0$ and $s \leq t$, and select $t - s$ further values $a_{s+1}, \ldots, a_t$ such that $a_1, \ldots, a_t$ are distinct nonnegative integers less than n. The Vandermonde matrix

$$\begin{pmatrix} \alpha^{a_1} & \ldots & \alpha^{a_t} \\ \alpha^{2a_1} & \ldots & \alpha^{2a_t} \\ \vdots & & \vdots \\ \alpha^{ta_1} & \ldots & \alpha^{ta_t} \end{pmatrix}$$

is singular, since the sum of its first s columns is zero. But this contradicts the fact that $\alpha^{a_1}, \ldots, \alpha^{a_t}$ are distinct elements of $GF(2^k)$. (See exercise 1.2.3–37.)

[The idea of polynomial hashing originated with M. Hanan, S. Muroga, F. P. Palermo, N. Raver, and G. Schay; see *IBM J. Research & Development* 7 (1963), 121–129; *U.S. Patent* 3311888 (March 28, 1967).]

8. By induction. The strong induction hypotheses can be supplemented by the fact that $\{(-1)^k(rq_k + q_{k-1})\theta\} = (-1)^k\big(r(q_k\theta - p_k) + (q_{k-1}\theta - p_{k-1})\big)$ for $0 \leq r \leq a_k$. The "record low" values of $\{n\theta\}$ occur for $n = q_1, q_2 + q_1, 2q_2 + q_1, \ldots, a_2q_2 + q_1 = 0q_4 + q_3,\ q_4 + q_3, \ldots, a_4q_4 + q_3 = 0q_6 + q_5, \ldots$; the "record high" values occur for $n = q_0,\ q_1 + q_0, \ldots, a_1q_1 + q_0 = 0q_3 + q_2, \ldots$. These are the steps when interval number 0 of a new length is formed.

9. We have $\phi^{-1} = /1, 1, 1, \ldots/$ and $\phi^{-2} = /2, 1, 1, \ldots/$. Let $\theta = /a_1, a_2, \ldots/$, and $\theta_k = /a_{k+1}, a_{k+2}, \ldots/$, and $Q_k = q_k + q_{k-1}\theta_{k-2}$ in the notation of exercise 8. If $a_1 > 2$, the very first break is bad. The three sizes of intervals in exercise 8 are, respectively, $(1 - r\theta_{k-1})/Q_k$, θ_{k-1}/Q_k, and $\big(1 - (r-1)\theta_{k-1}\big)/Q_k$, so the ratio of the first length to the second is $(a_k - r) + \theta_k$. This will be less than $\frac{1}{2}$ when $r = a_k$ and $a_{k+1} \geq 2$; hence $\{a_2, a_3, \ldots\}$ must all equal 1 if there are to be no bad breaks. [For related theorems, cf. R. L. Graham and J. H. van Lint, *Canadian J. Math.* **20** (1968), 1020–1024, and the references cited there.]

11. There would be a problem if $K = 0$. If keys were required to be nonzero as in Program L, this change would be worth while, and we could also represent empty positions by 0.

12. We can store K in KEY[0], replacing lines 14–19 by

```
        STA   TABLE(KEY)          A - S1
        CMPA  TABLE,2(KEY)        A - S1
        JE    3F                  A - S1
2H      ENT1  0,2                 C - 1 - S2
        LD2   TABLE,1(LINK)       C - 1 - S2
        CMPA  TABLE,2(KEY)        C - 1 - S2
        JNE   2B                  C - 1 - S2
3H      J2Z   5F                  A - S1
        ENT1  0,2                   S2
        JMP   SUCCESS               S2
```

The "savings" in time is $C - 1 - 5A + S + 4S1$, which is actually a net *loss* because C is rarely more than 5. (An inner loop shouldn't always be optimized!)

13. Let the table entries be of two distinguishable types, as in Algorithm C, with an additional one-bit TAG[i] field in each entry. This solution uses circular lists, with TAG[i] = 1 in the first word of each list.

A1. [Initialize.] Set $i \leftarrow j \leftarrow h(K) + 1$, $Q \leftarrow q(K)$.

A2. [Is there a list?] If TABLE[i] is empty, set TAG[i] $\leftarrow$ 1 and go to A8. Otherwise if TAG[i] = 0, go to A7.

A3. [Compare.] If Q = KEY[i], the algorithm terminates successfully.

A4. [Advance to next.] If LINK[i] $\neq j$, set $i \leftarrow$ LINK[i] and go back to A3.

A5. [Find empty node.] Decrease R one or more times until finding a value such that TABLE[R] is empty. If $R = 0$, the algorithm terminates with overflow; otherwise set LINK[i] $\leftarrow R$.

A6. [Prepare to insert.] Set $i \leftarrow R$, TAG[R] $\leftarrow$ 0, and go to A8.

A7. [Displace a record.] Repeatedly set $i \leftarrow$ LINK[i] one or more times until LINK[i] = j. Then do step A5. Then set TABLE[R] $\leftarrow$ TABLE[j], $i \leftarrow j$, TAG[j] $\leftarrow$ 1.

A8. [Insert new key.] Mark TABLE[i] as an occupied node, with KEY[i] $\leftarrow Q$, LINK[i] $\leftarrow j$. ∎

(Note that if TABLE[i] is occupied it is possible to determine the corresponding full key K, given only the value of i. We have $q(K)$ = KEY[i], and then if we set $i \leftarrow$ LINK[i] zero or more times until TAG[i] = 1 we will have $h(K) = i - 1$.)

14. To insert a new key K: Set Q $\Leftarrow$ AVAIL, TAG(Q) $\leftarrow$ 1, and store K in this word. [Alternatively, if keys are short, omit this and substitute K for Q in what follows.] Then set R $\Leftarrow$ AVAIL, TAG(R) $\leftarrow$ 1, AUX(R) $\leftarrow$ Q, LINK(R) $\leftarrow \Lambda$. Set P $\leftarrow h(K)$, and then:

If TAG(P) = 0, set TAG(P) $\leftarrow$ 2, AUX(P) $\leftarrow$ R.

If TAG(P) = 1, set S $\Leftarrow$ AVAIL, CONTENTS(S) $\leftarrow$ CONTENTS(P), TAG(P) $\leftarrow$ 2, AUX(P) $\leftarrow$ R, LINK(P) $\leftarrow$ S.

If TAG(P) = 2, set LINK(R) $\leftarrow$ AUX(P), AUX(P) $\leftarrow$ R.

To retrieve a key K: Set P $\leftarrow h(K)$, and then:

If TAG(P) $\neq$ 2, K is not present.

If TAG(P) = 2, set P $\leftarrow$ AUX(P). Then set P $\leftarrow$ LINK(P) zero or more times until either AUX(P) points to a word containing K [or if keys are short, AUX(P) = K] or LINK(P) = Λ.

(Elcock's original scheme, *Comp. J.* **8** (1965), 242–243, actually used TAG = 2 and TAG = 3 to distinguish between lists of length one (when we can save one word of space) and longer lists. This would be a worthwhile improvement, since we presumably have such a large scatter table that almost all lists have length one.)

15. Knowing that there is always an empty node makes the inner search loop faster, since we need not maintain a counter to determine how many times step L2 is performed. The shorter program amply compensates for this one wasted cell. [On the other hand, there is a neat way to avoid the variable N and to allow the table to become completely full, in Algorithm L, without slowing down the method appreci-

ably except when the table actually does overflow: Simply check whether $i < 0$ happens twice! This trick does not apply to Algorithm D.]

16. No; 0 always leads to SUCCESS, whether it has been inserted or not, and SUCCESS occurs with different values of i at different times.

17. The second probe would then always be to position 0.

18. The code in (31) costs about $3(A - S1)$ units, compared to (30), and it saves $4u$ times the difference between (26), (27) and (28), (29). For a successful search, (31) is advantageous only when the table is more than about 94 percent full, and it never saves more than about $\frac{1}{2}u$ of time. For an unsuccessful search, (31) is advantageous when the table is more than about 71 percent full.

20. We want to show that

$$\binom{j}{2} \equiv \binom{k}{2} \text{ (modulo } 2^m\text{)} \qquad \text{and} \qquad 1 \le j \le k \le 2^m$$

implies $j = k$. Observe that $j(j-1) \equiv k(k-1)$(modulo 2^{m+1}) implies $(k-j) \times (k+j-1) \equiv 0$. If $k-j$ is odd, $k+j-1$ must be a multiple of 2^{m+1}, but that's impossible since $2 \le k+j-1 \le 2^{m+1}-2$. Hence $k-j$ is even, so $k+j-1$ is odd, so $k-j$ is a multiple of 2^{m+1}, so $k = j$. [Conversely, if M is not a power of 2, this probe sequence does not work.]

The probe sequence has secondary clustering, and it increases the running time of Program D (as modified in (30)) by about $\frac{1}{2}(C-1) - (A - S1)$ units since $B \approx \binom{C+1}{3}/M$ will now be negligible. This is a small improvement, until the table gets about 60 percent full.

21. If N is decreased, Algorithm D may fail since it may reach a state with no empty spaces and it will loop indefinitely. On the other hand, if N isn't decreased, Algorithm D may signal overflow when there still is room. The latter alternative is the lesser of the two evils, because rehashing can be used to get rid of deleted cells. (Note that in this case Algorithm D should increase N and test for overflow *only* when inserting an item into a previously *empty* position, since N represents the number of nonempty positions.) We could also have two counters.

22. Suppose that positions $j-1, j-2, \ldots, j-k$ are occupied and $j-k-1$ is empty (modulo M). The keys which probe position j and find it occupied before being inserted are precisely those keys in positions $j-1$ through $j-k$ whose hash address does not lie between $j-1$ and $j-k$; these "problematical keys" appear in order of insertion. Algorithm R moves the first such key into position j, and repeats the process on a smaller range of problematical positions until no problematical keys remain.

23. If possible, replace KEY[i] by a later key on the list that starts at LINK[i], by recomputing the hash addresses and searching down the list for each such key. (Recall that the lists are almost always short.) Algorithm C should also be changed to use a doubly-linked list of available space instead of R.

24. $P(P-1)(P-2)P(P-1)P(P-1)/MP(MP-1)\ldots(MP-6) = M^{-7} \times \big(1 - (5 - 21/M)P^{-1} + O(P^{-2})\big)$. In general, the probability of a hash sequence $a_1 \ldots a_N$ is $(\prod_{0 \le j < M} P^{\underline{b_j}})/(MP)^{\underline{N}} = M^{-N} + O(P^{-1})$, where b_j is the number of a_i that equal j.

25. Let the $(N+1)$st key hash to location a; P_k is M^{-N} times the number of hash sequences that leave the k locations $a, a-1, \ldots, a-k+1$ (modulo M) occupied and $a-k$ empty. The number of such sequences with $a+1, \ldots, a+t$ occupied and $a+t+1$ empty is $g(M, N, t+k)$, by circular symmetry of the algorithm.

26. $\dfrac{9!}{2!2!4!1!} f(3,2)f(3,2)f(5,4)f(2,1) = 2^2 3^5 5^4 7 = 4252500.$

27. Following the hint,

$$s(n,x,y) = \sum_{k\ge 0} \binom{n}{k} x(x+k)^k (y-k)^{n-k-1}(y-n) + n\sum_{k\ge 0}\binom{n-1}{k-1}(x+k)^k(y-k)^{n-k-1}(y-n).$$

In the first sum, replace k by $n-k$ and apply Abel's formula; in the second, replace k by $k+1$. Now

$$g(M,N,k) = \binom{N}{k}(k+1)^{k-1}(M-k-1)^{N-k-1}(M-N-1),$$

and

$$M^N \sum (k+1)P_k = \sum_{k\ge 0}\binom{k+2}{2} g(M,N,k)$$

$$= \frac{1}{2}\left(\sum_{k\ge 0}(k+1)g(M,N,k) + \sum_{k\ge 0}(k+1)^2 g(M,N,k)\right).$$

The first sum is $M^N \sum P_k = M^N$, and the second is $s(N,1,M-1) = M^N + N\big(2M^{N-1} + (N-1)(3M^{N-2}+\cdots)\big) = M^N Q_1(M,N)$. [See J. Riordan, *Combinatorial Identities* (New York: Wiley, 1968), 18–23, for further study of sums like $s(n,x,y)$.]

28. Let $t(n,x,y) = \sum_{k\ge 0}\binom{n}{k}(x+k)^{k+2}(y-k)^{n-k-1}(y-n)$; then as in exercise 27 we find $t(n,x,y) = xs(n,x,y) + nt(n-1,x+1,y-1)$, $t(N,1,M-1) = M^N\big(3Q_3(M,N) - 2Q_2(M,N)\big)$. Thus $\sum(k+1)^2 P_k = M^{-N}\sum\big(\frac{1}{3}(k+1)^3 + \frac{1}{2}(k+1)^2 + \frac{1}{6}(k+1)\big)g(M,N,k) = Q_3(M,N) - \frac{2}{3}Q_2(M,N) + \frac{1}{2}Q_1(M,N) + \frac{1}{6}$. Subtracting $(C'_N)^2$ gives the variance, which is approximately $\frac{3}{4}(1-\alpha)^{-4} - \frac{2}{3}(1-\alpha)^{-3} - \frac{1}{12}$. The standard deviation is often larger than the mean; for example, when $\alpha = .9$ the mean is 50.5 and the standard deviation is $\frac{1}{2}\sqrt{27333} \approx 83$.

29. Let $M = m+1$, $N = n$; the safe parking sequences are precisely those in which location 0 is empty when Algorithm L is applied to the hash sequence $(M-a_1)\ldots(M-a_n)$. Hence the answer is $f(m+1,n) = (m+1)^n - n(m+1)^{n-1}$. [This parking problem originated with A. G. Konheim and B. Weiss, *SIAM J. Applied Math.* **14** (1966), 1266–1274, who were the first to publish a correct analysis of Algorithm L.]

30. Obviously if the cars get parked they define such a permutation. Conversely, if $p_1 p_2 \ldots p_n$ exists, let $q_1 q_2 \ldots q_n$ be the inverse permutation ($q_i = j$ iff $p_j = i$), and let b_i be the number of a_j that equal i. Every car will be parked if we can prove that

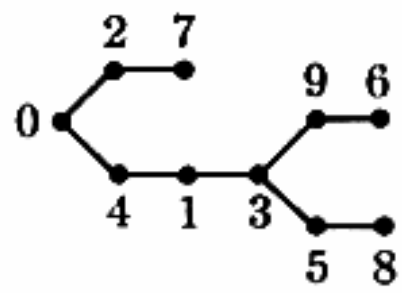
$b_n \le 1$, $b_{n-1} + b_n \le 2$, etc.; equivalently $b_1 \ge 1$, $b_1 + b_2 \ge 2$, etc. But this is clearly true, since the k elements $a_{q_1}, \ldots, a_{q_k}$ are all $\le k$.

[Let r_j be the "left influence" of q_j, namely $r_j = k$ iff $q_{j-1} < q_j, \ldots, q_{j-k+1} < q_j$ and either $j = k$ or $q_{j-k} > q_j$. Of all permutations $p_1 \ldots p_n$ that dominate a given wakeup sequence $a_1 \ldots a_n$, the "park immediately" algorithm finds the smallest one (in lexicographic order). Konheim and Weiss observed that the number of wakeup sequences leading to a given permutation $p_1 \ldots p_n$ is $\prod_{1 \le j \le n} r_j$; it is remarkable that the sum of these products, taken over all permutations $q_1 \ldots q_n$, is $(n+1)^{n-1}$.]

31. Many interesting connections are possible, and the following two are the author's favorites:

a) In the notation of the previous answer, the counts $b_1, b_2, \ldots, b_n$ correspond to a full parking sequence iff $(b_1, b_2, \ldots, b_n, 0)$ is a valid sequence of *degrees* of tree nodes in preorder. (Cf. 2.3.3–9, which illustrates postorder.) Each such tree corresponds to $n!/b_1! \ldots b_n!$ distinct labeled free trees on $\{0, \ldots, n\}$, since we can let 0 be the label of the root, and for $k = 1, 2, \ldots, n$ we can successively choose the labels of the sons of the kth node in preorder in $(b_k + \cdots + b_n)!/b_k!(b_{k+1} + \cdots + b_n)!$ ways from the remaining unused labels, attaching labels from left to right in increasing order. And each such sequence of counts corresponds to $n!/b_1! \ldots b_n!$ wakeup sequences.

b) Dominique Foata has given the following pretty one-to-one correspondence: Let $a_1 \ldots a_n$ be a safe parking sequence, which leaves car q_j parked in space j. A labeled free tree on $\{0, 1, \ldots, n\}$ is constructed by drawing a line from j to 0 when $a_j = 1$, and from j to $q_{a_j - 1}$ otherwise, for $1 \le j \le n$. (Think of the tree nodes as cars; car j is connected to the car that eventually winds up parked just before where wife j woke up.) Thus the example shown in exercise 29 corresponds to the free tree



The sequence of parked cars may be obtained from the tree by topological sorting, assuming that arrows emanate from the root 0, choosing the smallest "source" at each step. From this sequence, $a_1 \ldots a_n$ can be reconstructed.

Actually the two constructions in (a) and (b) are strongly related. For further information, see Dominique Foata and John Riordan, "Mappings of acyclic and parking functions" (to appear).

32. Let subscripts be treated cyclically, so that $c_M = c_0$, $c_{M+1} = c_1$, etc. There is no solution with $c_j = b_j + c_{j+1} - 1$ for all j, since the sum over all j would give $\sum c_j = \sum b_j + \sum c_j - M < \sum c_j$. Hence every solution has $M - \sum b_j$ values of j such that $b_j = c_{j+1} = 0$. If $(c'_0, \ldots, c'_{M-1})$ is a different solution we must have $c'_{j+1} > 0$ for all such j; but this implies $c'_{j+2} > c_{j+2}$, $c'_{j+3} > c_{j+3}, \ldots$, a contradiction. The solution can be found by defining $c_{M-1}, c_{M-2}, \ldots$ on the assumption that $c_0 = 0$; then if c_0 turns out to be greater than 0, it suffices to redefine $c_{M-1}, c_{M-2}, \ldots$ until no more changes are made. [A slightly more efficient algorithm, for computing $\sum c_j$ from the b's, has been given by T. C. Lowe and D. C. Roberts (to appear).]

33. The individual probabilities are not independent, since the condition $b_0 + b_1 + \cdots + b_{M-1} = N$ was not taken into account; the derivation allows a nonzero probability that $\sum b_j$ has any given nonnegative value. Equations (46) are not strictly correct; they imply, for example, that q_k is positive for all k, contradicting the fact that c_j can never exceed $N - 1$.

34. (a) There are $\binom{N}{k}$ ways to choose the set of j such that a_j has a particular value, and $(M-1)^{N-k}$ ways to assign values to the other a's. Therefore

$$P_{Nk} = \binom{N}{k} (M-1)^{N-k}/M^N.$$

(b) $P_N(z) = B(z)$ in (50). (c) $C'_N = \sum(k + \delta_{k0})P_{Nk} = P'_N(1) + P_N(0)$. For C_N, consider the total number of probes to find all keys. A list of length k contributes $\binom{k+1}{2}$ to the total; hence

$$C_N = M \sum \binom{k+1}{2} P_{Nk}/N = (M/N)\left(\tfrac{1}{2}P''_N(1) + P'_N(1)\right).$$

Thus we obtain (18), (19).

35. $\sum(1 + \frac{1}{2}k - (k+1)^{-1} + \delta_{k0})P_{Nk} = 1 + N/2M - M(1 - (1-1/M)^{N+1})/(N+1) + (1-1/M)^N \approx 1 + \frac{1}{2}\alpha - (1 - e^{-\alpha})/\alpha + e^{-\alpha}$.

36. $\sum(\delta_{k0} + k)^2 P_{Nk} = \sum(\delta_{k0} + k^2)P_{Nk} = P(0) + P''(1) + P'(1)$. Subtracting C'^2_N gives the answer, $(M-1)N/M^2 + (1-1/M)^N(1 - 2N/M - (1-1/M)^N) \approx \alpha + e^{-\alpha}(1 - 2\alpha - e^{-\alpha}) \leq 1 - e^{-1} - e^{-2} = 0.4968$.

37. Let S_N be the average value of $(C_N - 1)^2$; then

$$\begin{aligned}
M^N N^2 S_N &= \sum \binom{N}{k_1, \ldots, k_M} \left(\binom{k_1}{2} + \cdots + \binom{k_M}{2} \right)^2 \\
&= M(M-1) \sum \binom{N}{k_1, \ldots, k_M} \binom{k_1}{2}\binom{k_2}{2} + M \sum \binom{N}{k_1, \ldots, k_M} \binom{k_1}{2}^2 \\
&= M(M-1) \sum \binom{N}{k}\binom{N-k}{j}\binom{k}{2}\binom{j}{2} (M-2)^{N-k-j} \\
&\quad + M \sum \binom{N}{k}\binom{k}{2}^2 (M-1)^{N-k} \\
&= M(M-1)(\tfrac{1}{4}M^{N-4}N^{\underline{4}}) + M(M^{N-4})(\tfrac{1}{4}N^{\underline{4}} + N^{\underline{3}} + \tfrac{1}{2}N^{\underline{2}}).
\end{aligned}$$

The variance is $S_N - ((N-1)/2M)^2 = (M-1)(N-1)/2NM^2$. (Hence the standard deviation is $O(M^{-1/2})$ as $M \to \infty$.)

38. The average number of probes is $\sum P_{Nk}(2H_{k+1} - 2 + \delta_{k0})$ in the unsuccessful case, $(M/N)\sum P_{Nk}k(2(1+1/k)H_k - 3)$ in the successful case, by Eqs. 6.2.2–5,6. These sums are respectively equal to $2f(N) + 2M(1 - (1-1/M)^{N+1})/(N+1) + (1-1/M)^N - 2$ and $2(M/N)f(N) + 2f(N-1) + 2M(1 - (1-1/M)^N)/N - 3$, where $f(N) = \sum P_{Nk}H_k$. Exercise 5.2.2–57 tells us that $f(N) = \ln\alpha + \gamma + E_1(\alpha) + O(M^{-1})$ when $N = \alpha M$, $M \to \infty$.

[Tree hashing was first proposed by P. F. Windley, *Comp. J.* **3** (1960), 84–88. The above analysis shows that tree hashing is not enough better than simple chaining

to justify the extra link fields (the lists are short anyway); and when M is small it is not enough better than pure tree search to justify the hashing time.]

39. $c(k_1, k_2, \ldots) = (M - N + 1)c(k_1 - 1, k_2, \ldots) + (k_1 + 1)c(k_1 + 1, k_2 - 1, k_3, \ldots) + 2(k_2 + 1)c(k_1, k_2 + 1, k_3 - 1, \ldots) + \cdots$. [This is similar to the recurrence in exercise 1.2.5–21, but it does not seem to have a simple solution.] Hence

$$\begin{aligned} S_{N+1} &= (M - N) \sum_{j\ge 1, k_1+2k_2+\cdots=N+1} \binom{j}{2} k_j\, c(k_1 - 1, k_2, \ldots) \\ &\quad + \sum_{j\ge 1, k_1+2k_2+\cdots=N+1} \binom{j}{2} k_j(k_1 + 1)c(k_1 + 1, k_2 - 1, \ldots) + \cdots \\ &= (M - N)S_N + \sum_{j\ge 1, k_1+2k_2+\cdots=N} c(k_1, k_2, \ldots) \times \\ &\qquad \left(\left(\binom{j}{2} k_j - \binom{1}{2} + \binom{2}{2}\right) k_1 + \left(\binom{j}{2} k_j - \binom{2}{2} + \binom{3}{2}\right) 2k_2 + \cdots\right) \\ &= MS_N + \sum_{j\ge 1, k_1+2k_2+\cdots=N} j^2 k_j\, c(k_1, k_2, \ldots) = MS_N + 2S_N + NM^N. \end{aligned}$$

Consequently $S_N = (N - 1)M^{N-1} + (N - 2)M^{N-2}(M + 2) + \cdots + M(M + 2)^{N-2} = \frac{1}{4}(M(M + 2)^N - M^{N+1} - 2NM^N)$.

Consider the total number of probes in unsuccessful searches, summed over all M values of $h(K)$; each list of length k contributes $k + \delta_{k0} + \binom{k}{2}$ to the total, hence $M^{N+1}C'_N = M^{N+1} + S_N$.

40. Define U_N to be like S_N in exercise 39, but with $\binom{j}{2}$ replaced by $\binom{j+1}{3}$. We find $U_{N+1} = (M + 3)U_N + S_N + NM^N$, hence $U_N = \frac{1}{36}(M^N(M - 6N) - 9M(M + 2)^N + 8M(M + 3)^N)$. The variance is $2U_N/M^{N+1} + C'_N - C'^2_N$, which approaches $\frac{35}{144} - \frac{1}{12}\alpha - \frac{1}{4}\alpha^2 + (\frac{1}{4}\alpha - \frac{5}{8})e^{2\alpha} + \frac{4}{9}e^{3\alpha} - \frac{1}{16}e^{4\alpha}$ for $N = \alpha M$, $M \to \infty$. When $\alpha = 1$ this is about 4.50, so the standard deviation is bounded by about 2.12.

41. Let V_N be the average length of the block of occupied cells at the "high" end of the table. The probability that this block has length k is $A_{Nk}(M - 1 - k)^{\underline{N-k}}/M^N$, where A_{Nk} is the number of hash sequences (35) such that Algorithm C leaves the first $N - k$ and the last k cells occupied and such that the subsequence $1, 2, \ldots, N - k$ appears in increasing order. Therefore

$$\begin{aligned} M^N V_N &= \sum kA_{Nk}(M - 1 - k)^{\underline{N-k}} = M^{N+1} - \sum (M - k)A_{Nk}(M - 1 - k)^{\underline{N-k}} \\ &= M^{N+1} - (M - N)\sum A_{Nk}(M - k)^{\underline{N-k}} = M^{N+1} - (M - N)(M + 1)^N. \end{aligned}$$

Now $T_N = (N/M)(1 + V_N - T_0 - \cdots - T_{N-1})$, since $T_0 + \cdots + T_{N-1}$ is the average number of times R has previously decreased and N/M is the probability that it decreases this time. The solution to this recurrence is $T_N = (N/M)(1 + 1/M)^N$. (Such a simple formula deserves a simpler proof!)

42. $S1_N$ is the number of items that were inserted with $A = 0$, divided by N.

44. Number the positions of the array from 1 to m, left to right. Considering the set of all $\binom{n}{k}$ sequences of operations with k "p-steps" and $n - k$ "q-steps" to be equally

likely, let $g(m, n+1, k, r)$ be $\binom{n}{k}$ times the probability that the first $r-1$ positions become occupied and the rth remains empty. Thus $g(m, l, k, r)$ is $(m-1)^{-(l-1-k)}$, times the sum over all configurations

$$1 \le a_1 < \cdots < a_k < l; \qquad (c_1, \ldots, c_{l-1-k}), \qquad 2 \le c_i \le m,$$

of the probability that the first empty location is r, when the a_jth operation is a p-step and the remaining $l-1-k$ operations are q-steps that begin by selecting positions $c_1, \ldots, c_{l-1-k}$, respectively. By summing over all configurations subject to the further condition that the a_jth operation occupies position b_j, for specified

$$1 \le b_1 < \cdots < b_k < r,$$

we obtain the recurrence

$$g(m, l, k+1, r) = \sum_{\substack{a<l \\ b<r \\ 1\le b\le a}} \frac{(l-b-1)!}{(l-r)!} \frac{(m-r)!}{(m-b)!} (m-l+1) g(m, a, k, b);$$

$$g(m, l, 0, r) = \frac{(l-1)!}{(l-r)!} \frac{(m-r)!}{m!} (m-l+1) \left(P_l + (1-\delta_{r1}) \frac{m}{l-1} (1-P_l) \right),$$

where $P_l = (m/(m-1))^{l-1}$. It follows that if $G(m, l, k) = \sum_{1\le r\le l} (m+1-r) \times g(m, l, k, r)$, we have

$$G(m, l, k+1) = \frac{m-l+1}{m-l+2} \sum_{1\le a<l} G(m, a, k); \qquad G(m, l, 0) = \frac{m-l+1}{m-l+2} (m+P_l).$$

The answer to the stated problem is $m - \sum_{0\le k\le n} p^k q^{n-k} G(m, n+1, k)$, which (after some maneuvering) equals $m - ((m-n)/(m-n+1))(Q_n + mR + pSR)$, where

$$Q_j = P_{j+1} q^j,$$

$$R = \left(1 - \frac{p}{m+1}\right)\left(1 - \frac{p}{m}\right) \cdots \left(1 - \frac{p}{m-n+2}\right) = \prod_{0\le j<n} \left(1 - \frac{p}{m+1-j}\right),$$

$$S = \frac{\left(1 - \frac{1}{m+1}\right) Q_0}{\left(1 - \frac{p}{m+1}\right)} + \frac{\left(1 - \frac{1}{m}\right) Q_1}{\left(1 - \frac{p}{m+1}\right)\left(1 - \frac{p}{m}\right)} + \cdots + \frac{\left(1 - \frac{1}{m-n+2}\right) Q_{n-1}}{R}$$

$$= \sum_{0\le j<n} \frac{(1 - 1/(m+1-j)) Q_j}{\prod_{0\le i\le j} (1 - p/(m+1-i))}.$$

When $p = 1/m$, $Q_j = 1$ for all j. Letting $w = m+1$, $n = \alpha w$, $w \to \infty$, we find $\ln R = -p(H_w - H_{w(1-\alpha)}) + O(p^2)$; hence $R = 1 + w^{-1} \ln(1-\alpha) + O(w^{-2})$; and similarly $S = \alpha w + O(1)$. Thus the answer is $(1-\alpha)^{-1} - 1 - \alpha - \ln(1-\alpha) + O(w^{-1})$.

Notes: The simpler problem "with probability p occupy the leftmost, otherwise occupy any randomly chosen empty position" is solved by taking $P_j = 1$ in the above, and the answer is $m - (m+1)(m-n)R/(m-n+1)$. To get C'_N for random probing with secondary clustering, set $n = N-1$, $m = M$ and add 1 to the above answer.

46. Define the numbers $\left[\!\left[{n \atop k}\right]\!\right]$ for $k \geq 0$ by the rule

$$\sum_k \binom{x+k}{k} \left[\!\left[{n \atop k}\right]\!\right] = (x+n+1)^n$$

for all x and all nonnegative integers n. Setting $x = -1, -2, \ldots, -n-1$ implies that

$$\left[\!\left[{n \atop k}\right]\!\right] = \sum_j \binom{k}{j} (-1)^j (n-j)^n \qquad \text{for} \qquad 0 \leq k \leq n;$$

then setting $x = 0$ implies that we may take $\left[\!\left[{n \atop k}\right]\!\right] = 0$ for all $k > n$, so the two sides of the defining equation are polynomials in x of degree n that agree on $n+1$ points. It follows that the numbers $\left[\!\left[{n \atop k}\right]\!\right]$ have the stated property.

Let $f(N, r)$ be the number of hash sequences $a_1 \ldots a_N$ that leave the first r locations occupied and the next one empty. There are $\binom{M-r-1}{N-r-1}$ possible patterns of occupied cells, and each pattern occurs as many times as there are sequences $a'_1 \ldots a'_N$, $1 \leq a'_i \leq N$, that contain each of the numbers $r+1, r+2, \ldots, N$ at least once. By inclusion-exclusion, there are $\left[\!\left[{N \atop N-r}\right]\!\right]$ such sequences; hence

$$f(N, r) = \binom{M-r-1}{N-r} \left[\!\left[{N \atop N-r}\right]\!\right].$$

Now $C'_N = 1 + M^{-N-1} \sum_{0 \leq r \leq N} f(N, r) \left(\sum_{0 \leq a < r} r + \sum_{r < a < M} ((N-r)/(M-r-1))(r+1) \right) = 1 + M^{-N-1} \sum_{0 \leq r \leq N} f(N, r)\,(N + (N-1)r)$. Let

$$S_n(x) = \sum_k k \binom{x+k}{k} \left[\!\left[{n \atop k}\right]\!\right];$$

we have

$$(x+1)^{-1} S_n(x) + \sum_k \binom{x+k}{k} \left[\!\left[{n \atop k}\right]\!\right] = \sum_k \binom{x+1+k}{k} \left[\!\left[{n \atop k}\right]\!\right];$$

hence $S_n(x) = (x+1)\big((x+n+2)^n - (x+n+1)^n\big)$. It follows that $C'_N = N(1+1/M) - (N-1)(1-N/M)(1+1/M)^N \approx N\big(1-(1-\alpha)e^{\alpha}\big)$; and $C_N = (N-1)\big((1+1/M)/2 + (1+1/M)^N\big) + (3M^2+6M+2)\big((1+1/M)^N - 1\big)/N - (3M+2)(1+1/M)^N$, which is $(e-2.5)M + O(1)$ when $N = M-1$.

For further properties of the numbers $\left[\!\left[{n \atop k}\right]\!\right]$, cf. John Riordan, *Combinatorial Identities* (New York: Wiley, 1968), 228.

47. The analysis of Algorithm L applies, almost word-for-word! Any probe sequence with cyclic symmetry, and which explores only positions adjacent to those previously examined, will have the same behavior.

48. $C'_N = 1 + p + p^2 + \cdots$, where $p = N/M$ is the probability that a random location is filled; hence $C'_N = M/(M-N)$, and $C_N = N^{-1} \sum_{0 \leq k < N} C'_k = N^{-1} M (H_M - H_{M-N})$. These values are approximately equal to those for uniform

probing, but slightly higher because of the chance of multiple probes in the same place. Indeed, for $4 = N < M \leq 16$, linear probing is better!

In practice we wouldn't use infinitely many hash functions, some other scheme like linear probing would ultimately be used as a last resort. This method is inferior to those described in the text, but it is of historical importance because it suggested Morris's method which led to Algorithm D. See *CACM* **6** (1963), 101, where M. D. McIlroy credits the idea to V. A. Vyssotsky; the same technique had been discovered as early as 1956 by A. W. Holt, who used it successfully in the GPX system for the UNIVAC.

49. $C'_N - 1 = \sum_{k>b} (k-b)P_{Nk} \approx \sum_{k>b} (k-b)e^{-\alpha b}(\alpha b)^k/k! = \alpha b t_b(\alpha)$. [*Note:* In general if $P(z)$ is any probability generating function,

$$\sum_{b\geq 0}\left(\sum_{k>b}(k-b)P_k\right)z^b = P'(1)/(1-z) + z(P(z)-1)/(1-z)^2.]$$

$$C_N - 1 = (M/N)\sum_{k>b}\binom{k-b+1}{2}P_{Nk}$$

$$= (M/2N)\sum_{k>b}\big(k(k-1) - 2k(b-1) + b(b-1)\big)P_{Nk}$$

$$\approx \tfrac{1}{2}e^{-b\alpha}(b\alpha)^b b!^{-1}\big(b + b\alpha - 2b + 2 + (b\alpha^2 - 2\alpha(b-1) + b - 1)R(\alpha, b)\big).$$

[The analysis of successful search with chaining was first carried out by W. P. Heising in 1957. The simple expressions in (57), (58) were found by J. A. van der Pool in 1971; he also considered how to minimize a function that represents the combined cost of storage space and number of accesses. Since $\sum_{k>b}(k-b)^2P_{Nk} = (2N/M) \times (C_N - 1) - (C'_N - 1)$, we can determine the variance of C'_N and of the number of overflows per bucket. The variance of the total number of overflows may be approximated by M times the variance in a single bucket, but this is actually too high because the total number of records is constrained to be N. The true variance may be found as in exercise 37. Cf. also the derivation of the chi-square test in Section 3.3.1C.]

50. And next that $Q_0(M, N-1) = (M/N)\big(Q_0(M, N) - 1\big)$.

51. $R(\alpha, n) = \alpha^{-1}\big(n!e^{\alpha n}(\alpha n)^{-n} - Q_0(\alpha n, n)\big)$.

52. Cf. Eq. 1.2.11.3–9 and exercise 3.1–14.

53. By Eq. 1.2.11.3–8, $\alpha(\alpha n)^n R(\alpha, n) = e^{\alpha n}\gamma(n+1, \alpha n)$; hence by the suggested exercise $R(\alpha, n) = (1-\alpha)^{-1} - (1-\alpha)^{-3}n^{-1} + O(n^{-2})$. [This asymptotic formula can be obtained more directly by the method of (43), by noting that the coefficient of α^k in $R(\alpha, n)$ is

$$1 - \binom{k+2}{2}n^{-1} + O(k^4 n^{-2}).$$

In fact, the coefficient of α^k is

$$\sum_{r\geq 0}(-1)^r n^{-r}\left\{{r+k+1 \atop k+1}\right\}$$

by Eq. 1.2.9–28.]

55. If $B(z)C(z) = \sum s_i z^i$, we have $c_0 = s_0 + \cdots + s_b$, $c_1 = s_{b+1}$, $c_2 = s_{b+2}, \ldots$; hence $B(z)C(z) = z^b C(z) + Q(z)$. Now $P(z) = z^b$ has $b - 1$ roots $|q_j| < 1$, determined as the solutions to $e^{\alpha(q_j - 1)} = \omega^{-j} q_j$, $\omega = e^{2\pi i/b}$. To solve $e^{\alpha(q-1)} = \omega^{-1} q$, let $t = \alpha q$ and $z = \alpha\omega e^{-\alpha}$ so that $t = ze^t$. By Lagrange's formula we get

$$\frac{1}{1-q} = 1 + \sum_{r\ge 0} r \sum_{n\ge r} \frac{n^{n-r-1}\omega^n \alpha^{n-r} e^{-n\alpha}}{(n-r)!}$$

$$= 1 + \sum_{r\ge 1} r \sum_{m\ge 0} \frac{\alpha^m}{m!} (-1)^m \sum_{n\ge r} \binom{m}{n-r} (-1)^{n-r} \omega^n n^{m-1}.$$

By Abel's limit theorem, letting $|\omega| \to 1$ from inside the unit circle, this can be rearranged to

$$\frac{1-\alpha\omega}{1-\omega} + \sum_{m\ge 2} \frac{\alpha^m}{m!} (-1)^m \sum_{n\ge 0} \binom{m-2}{n} (-1)^n \omega^{n+1} (n+1)^{m-1}.$$

Now replacing ω by ω^j and summing for $1 \le j < b$ reduces this to

$$\frac{b-1}{2} + \alpha\frac{b-1}{2} + \sum_{m\ge 2} \alpha^m \left(-\frac{1}{2} + \frac{(-1)^m}{m!} b \sum_{n\ge 1} \binom{m-2}{nb-1} (-1)^{nb-1} (nb)^{m-1} \right)$$

and the desired result follows after some more juggling since

$$t_n(\alpha) = (-1)^{n-1}(n!\alpha)^{-1} \sum_{m>n} (-n\alpha)^m / m(m-1)(m-n-1)!.$$

This analysis was first begun by M. Tainiter, *JACM* **10** (1963), 307–315.

58. 0 1 2 3 4 and 0 2 4 1 3, plus additive shifts of 1 1 1 1 1 mod 5, each with probability $\frac{1}{10}$. [For $M = 6$, we need 30 permutations, and a solution exists starting with $\frac{1}{20} \times$ 0 1 2 3 4 5, $\frac{1}{60} \times$ 0 1 3 2 5 4, $\frac{1}{60} \times$ 0 2 4 3 1 5, $\frac{1}{20} \times$ 0 2 3 4 5 1, $\frac{1}{30} \times$ 0 3 4 1 2 5. For $M = 7$, we need 49, and a solution is generated by $\frac{1}{35} \times$ 0 1 2 3 4 5 6, $\frac{2}{105} \times$ 0 1 5 3 2 4 6, $\frac{1}{35} \times$ 0 2 4 3 5 1 6, $\frac{2}{105} \times$ 0 2 6 3 1 4 5, $\frac{1}{35} \times$ 0 3 6 1 4 2 5, $\frac{1}{105} \times$ 0 3 2 6 4 1 5, $\frac{1}{105} \times$ 0 3 1 5 4 2 6.]

59. No permutation can have a probability larger than

$$1 \Big/ \binom{M}{\lfloor M/2 \rfloor},$$

so there must be at least

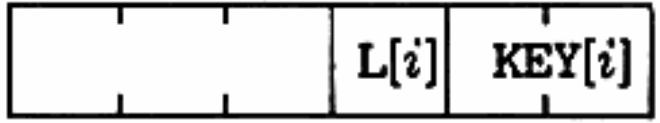
$$\binom{M}{\lfloor M/2 \rfloor} = \exp\left(M \ln 2 + O(\log M)\right)$$

permutations with nonzero probability.

63. MH_M, by exercise 3.3.2–8; the standard deviation is $\pi M/\sqrt{6}$. [This is the "mean time to failure" of the deletion method that simply marks cells "deleted."]

65. The keys can be stored in a separate table, allocated sequentially (assuming that deletions, if any, are LIFO). The scatter table entries point to this "names table",

e.g., TABLE[i] might have the form



,

where L[i] is the number of words in the key stored at locations KEY[i], KEY[i] + 1, ...

The rest of the scatter table entry might be used in any of several ways: (a) as a link for Algorithm C; (b) as part of the information associated with the key; or (c) as a "secondary hash code." The latter idea, suggested by Robert Morris, sometimes speeds up a search [we take a careful look at the key in KEY[i] only if $h_2(K)$ matches its secondary hash code, for some function $h_2(K)$].

SECTION 6.5

1. The path described in the hint can be converted by changing each downward step that runs from $(i-1, j)$ to a "new record low" value $(i, j-1)$ into an upward step. If c such changes are made, the path ends at $(n, n-2t+2c)$, where $c \geq 0$ and $c \geq 2t-n$; hence $n-2t+2c \geq n-2k$. In the permutation corresponding to the changed path, the smallest c elements of list B correspond to the downward steps that changed, and list A contains the $t-c$ elements corresponding to downward steps that didn't change.

When $t = k$ it is not difficult to see that the construction is reversible; hence exactly $\binom{n}{k}$ permutations are constructed. Note that, according to this proof, the contents of lists A and C may appear in arbitrary order.

Notes: We have counted these paths in another way in exercise 2.2.1–4. When $k = \lfloor n/2 \rfloor$ this construction proves *Sperner's Lemma*, which states that it is impossible to have more than $\binom{n}{\lfloor n/2 \rfloor}$ subsets of $\{1, 2, \ldots, n\}$ with no subset contained in another. [Emanuel Sperner, *Math. Zeitschrift* **27** (1928), 544–548.] For if we had such a collection of subsets, each of the $\binom{n}{k}$ permutations can have at most one of the subsets appearing in the initial positions, yet each subset appears in some permutation. The construction used here is a disguised form of a more general construction by which N. G. de Bruijn et al. [*Nieuw Archief voor Wiskunde* (2) **23** (1951), 191–193] proved the multiset generalization of Sperner's Lemma: "Let M be a multiset containing n elements (counting multiplicities). The collection of all $\lfloor n/2 \rfloor$-element submultisets of M is the largest possible collection such that no submultiset is contained in another." For example, the largest such collection when $M = \{a, a, b, b, c, c\}$ consists of the seven submultisets $\{a, a, b\}$, $\{a, a, c\}$, $\{a, b, b\}$, $\{a, b, c\}$, $\{a, c, c\}$, $\{b, b, c\}$, $\{b, c, c\}$. This would correspond to seven permutations of six attributes A_1, B_1, A_2, B_2, A_3, B_3 in which all queries involving A_i also involve B_i.

2. Let a_{ijk} be a list of all references to records having (i, j, k) as the respective values of the three attributes, and assume that a_{011} is the shortest of the three lists a_{011}, a_{101}, a_{110}. Then a minimum-length list is $a_{001}a_{011}a_{111}a_{101}a_{100}a_{110}a_{111}a_{011}a_{010}$. However, if a_{011} is empty and so is either of a_{001}, a_{010}, or a_{100}, the length can be shortened by deleting one of the two occurrences of a_{111}.

3. (a) Anise seed, possibly in combination with nutmeg and/or vanilla extract. (b) None.

4. Let p_t be the probability that the query involves exactly t bit positions, and let P_t be the probability that t given positions are all 1 in a random record. Then the

answer is $\sum_t p_t P_t$. By the principle of inclusion and exclusion,

$$P_t = \sum_{j\geq 0} (-1)^j \binom{t}{j} f(n-j, k, r)/f(n, k, r),$$

where $f(n, k, r)$ is the number of possible choices of r distinct k-bit attribute codes in an n-bit field, namely,

$$f(n, k, r) = \binom{\binom{n}{k}}{r}.$$

And (cf. exercise 1.3.3–26)

$$p_t = \sum_{i\geq 0} (-1)^i \binom{t+i}{t}\binom{n}{t+i} P_{t+i}|_{r=q} = \binom{n}{t} \sum_{j\geq 0} (-1)^j \binom{t}{j} f(t-j, k, q)/f(n, k, q).$$

Notes: The above calculations were first carried out, in more general form, by G. Orosz and L. Takács, *J. of Documentation* **12** (1956), 231–234. The mean $\sum t p_t$ is easily shown to be $n\big((1 - f(n-1, k, q)/f(n, k, q)\big)$. Another assumption, that the random attribute codes in records and queries are not necessarily distinct, as in the techniques of Harrison and Bloom, can be analyzed by the same method, setting $f(n, k, r) = \binom{n}{k}^r$. When the parameters are in appropriate ranges, we have $P_t \approx (1 - e^{-kr/n})^t$ and $\sum p_t P_t \approx P_{n(1-\exp(-kq/n))}$.

6. $L(k) = \sum_j \binom{m_1}{j}\binom{m_2}{k-j} L_1(j) L_2(k-j)/\binom{m_1+m_2}{k}$. [Hence if $L_1(k) \approx N_1\alpha^{-k}$ and $L_2(k) \approx N_2\alpha^{-k}$, then $L(k) \approx N_1 N_2 \alpha^{-k}$.]

7. (a) $L(1) = 3$, $L(2) = 1\frac{3}{4}$. (b) $L(1) = 3\frac{1}{4}$, $L(2) = 2\frac{1}{3}$, $L(3) = 1\frac{9}{16}$. [*Note:* Earl Sacerdoti has suggested the mapping

$$0\ 0\ *\ * \rightarrow 1$$
$$0\ 1\ *\ * \rightarrow 2$$
$$1\ 0\ *\ * \rightarrow 3$$
$$1\ 1\ *\ * \rightarrow 4$$

which has a worse "worst case" but a better average case: $L(1) = 3$, $L(2) = 2\frac{1}{6}$, $L(3) = 1\frac{1}{2}$. A similar mapping exists whenever $m = 2^n$.]

10. (a) There must be $\frac{1}{6}v(v-1)$ triples, and x_v must occur in $\frac{1}{2}v$ of them. (b) Since v is odd, there is a unique triple $\{x_i, y_j, z\}$ for each i, and so S' is readily shown to be a Steiner triple system. The pairs missing in K' are $\{z, x_2\}$, $\{x_2, y_2\}$, $\{y_2, x_3\}$, $\{x_3, y_3\}$, $\ldots$, $\{x_{v-1}, y_{v-1}\}$, $\{y_{v-1}, x_v\}$, $\{x_v, z\}$. (d) Starting with the case $v = 1$ and applying the operations $v \rightarrow 2v-2$, $v \rightarrow 2v+1$ yields all nonnegative numbers not of the form $3k+2$, because the cases $6k + (0, 1, 3, 4)$ come respectively from the smaller cases $3k + (1, 0, 1, 3)$.

11. Take a Steiner triple system on $2v+1$ objects. Call one of the objects z and name the other objects in such a way that the triples containing z are $\{z, x_i, \bar{x}_i\}$; delete these triples.

12. $\{k, (k+1) \bmod 14, (k+4) \bmod 14, (k+6) \bmod 14\}$, for $0 \leq k < 14$, where $(k+7) \bmod 14$ is the complement of k. [Complemented systems are a special case of so-called *group divisible* block designs; cf. Bose, Shrikhande, and Bhattacharya, *Ann. Math. Statistics* **24** (1953), 167–195.]

APPENDIX A

TABLES OF NUMERICAL QUANTITIES

Table 1

Quantities which are frequently used in standard subroutines and in analysis of computer programs. (40 decimal places)

$\sqrt{2} = 1.41421\ 35623\ 73095\ 04880\ 16887\ 24209\ 69807\ 85697-$
$\sqrt{3} = 1.73205\ 08075\ 68877\ 29352\ 74463\ 41505\ 87236\ 69428+$
$\sqrt{5} = 2.23606\ 79774\ 99789\ 69640\ 91736\ 68731\ 27623\ 54406+$
$\sqrt{10} = 3.16227\ 76601\ 68379\ 33199\ 88935\ 44432\ 71853\ 37196-$
$\sqrt[3]{2} = 1.25992\ 10498\ 94873\ 16476\ 72106\ 07278\ 22835\ 05703-$
$\sqrt[3]{3} = 1.44224\ 95703\ 07408\ 38232\ 16383\ 10780\ 10958\ 83919-$
$\sqrt[4]{2} = 1.18920\ 71150\ 02721\ 06671\ 74999\ 70560\ 47591\ 52930-$
$\ln 2 = 0.69314\ 71805\ 59945\ 30941\ 72321\ 21458\ 17656\ 80755+$
$\ln 3 = 1.09861\ 22886\ 68109\ 69139\ 52452\ 36922\ 52570\ 46475-$
$\ln 10 = 2.30258\ 50929\ 94045\ 68401\ 79914\ 54684\ 36420\ 76011+$
$1/\ln 2 = 1.44269\ 50408\ 88963\ 40735\ 99246\ 81001\ 89213\ 74266+$
$1/\ln 10 = 0.43429\ 44819\ 03251\ 82765\ 11289\ 18916\ 60508\ 22944-$
$\pi = 3.14159\ 26535\ 89793\ 23846\ 26433\ 83279\ 50288\ 41972-$
$1^\circ = \pi/180 = 0.01745\ 32925\ 19943\ 29576\ 92369\ 07684\ 88612\ 71344+$
$1/\pi = 0.31830\ 98861\ 83790\ 67153\ 77675\ 26745\ 02872\ 40689+$
$\pi^2 = 9.86960\ 44010\ 89358\ 61883\ 44909\ 99876\ 15113\ 53137-$
$\sqrt{\pi} = \Gamma(1/2) = 1.77245\ 38509\ 05516\ 02729\ 81674\ 83341\ 14518\ 27975+$
$\Gamma(1/3) = 2.67893\ 85347\ 07747\ 63365\ 56929\ 40974\ 67764\ 41287-$
$\Gamma(2/3) = 1.35411\ 79394\ 26400\ 41694\ 52880\ 28154\ 51378\ 55193+$
$e = 2.71828\ 18284\ 59045\ 23536\ 02874\ 71352\ 66249\ 77572+$
$1/e = 0.36787\ 94411\ 71442\ 32159\ 55237\ 70161\ 46086\ 74458+$
$e^2 = 7.38905\ 60989\ 30650\ 22723\ 04274\ 60575\ 00781\ 31803+$
$\gamma = 0.57721\ 56649\ 01532\ 86060\ 65120\ 90082\ 40243\ 10422-$
$\ln \pi = 1.14472\ 98858\ 49400\ 17414\ 34273\ 51353\ 05871\ 16473-$
$\phi = 1.61803\ 39887\ 49894\ 84820\ 45868\ 34365\ 63811\ 77203+$
$e^\gamma = 1.78107\ 24179\ 90197\ 98523\ 65041\ 03107\ 17954\ 91696+$
$e^{\pi/4} = 2.19328\ 00507\ 38015\ 45655\ 97696\ 59278\ 73822\ 34616+$
$\sin 1 = 0.84147\ 09848\ 07896\ 50665\ 25023\ 21630\ 29899\ 96226-$
$\cos 1 = 0.54030\ 23058\ 68139\ 71740\ 09366\ 07442\ 97660\ 37323+$
$\zeta(3) = 1.20205\ 69031\ 59594\ 28539\ 97381\ 61511\ 44999\ 07650-$
$\ln \phi = 0.48121\ 18250\ 59603\ 44749\ 77589\ 13424\ 36842\ 31352-$
$1/\ln \phi = 2.07808\ 69212\ 35027\ 53760\ 13226\ 06117\ 79576\ 77422-$
$-\ln \ln 2 = 0.36651\ 29205\ 81664\ 32701\ 24391\ 58232\ 66946\ 94543-$

Table 2

Quantities which are frequently used in standard subroutines and in analysis of computer programs, in *octal* notation. The *name* of each quantity, appearing at the left of the equal sign, is given in decimal notation.

$0.1 =$	*0.06314 63146 31463 14631 46314 63146 31463 14631 4632*
$0.01 =$	*0.00507 53412 17270 24365 60507 53412 17270 24365 6051*
$0.001 =$	*0.00040 61115 64570 65176 76355 44264 16254 02030 4467*
$0.0001 =$	*0.00003 21556 13530 70414 54512 75170 33021 15002 3522*
$0.00001 =$	*0.00000 24761 32610 70664 36041 06077 17401 56063 3442*
$0.000001 =$	*0.00000 02061 57364 05536 66151 55323 07746 44470 2603*
$0.0000001 =$	*0.00000 00153 27745 15274 53644 12741 72312 20354 0215*
$0.00000001 =$	*0.00000 00012 57143 56106 04303 47374 77341 01512 6333*
$0.000000001 =$	*0.00000 00001 04560 27640 46655 12262 71426 40124 2174*
$0.0000000001 =$	*0.00000 00000 06676 33766 35367 55653 37265 34642 0163*
$\sqrt{2} =$	*1.32404 74631 77167 46220 42627 66115 46725 12575 1744*
$\sqrt{3} =$	*1.56663 65641 30231 25163 54453 50265 60361 34073 4222*
$\sqrt{5} =$	*2.17067 36334 57722 47602 57471 63003 00563 55620 3202*
$\sqrt{10} =$	*3.12305 40726 64555 22444 02242 57101 41466 33775 2253*
$\sqrt[3]{2} =$	*1.20505 05746 15345 05342 10756 65334 25574 22415 0303*
$\sqrt[3]{3} =$	*1.34233 50444 22175 73134 67363 76133 05334 31147 6012*
$\sqrt[4]{2} =$	*1.14067 74050 61556 12455 72152 64430 60271 02755 7314*
$\ln 2 =$	*0.54271 02775 75071 73632 57117 07316 30007 71366 5364*
$\ln 3 =$	*1.06237 24752 55006 05227 32440 63065 25012 35574 5534*
$\ln 10 =$	*2.23273 06735 52524 25405 56512 66542 56026 46050 5071*
$1/\ln 2 =$	*1.34252 16624 53405 77027 35750 37766 40644 35175 0435*
$1/\ln 10 =$	*0.33626 75425 11562 41614 52325 33525 27655 14756 0622*
$\pi =$	*3.11037 55242 10264 30215 14230 63050 56006 70163 2112*
$1^\circ = \pi/180 =$	*0.01073 72152 11224 72344 25603 54276 63351 22056 1154*
$1/\pi =$	*0.24276 30155 62344 20251 23760 47257 50765 15156 7007*
$\pi^2 =$	*11.67517 14467 62135 71322 25561 15466 30021 40654 3410*
$\sqrt{\pi} = \Gamma(1/2) =$	*1.61337 61106 64736 65247 47035 40510 15273 34470 1776*
$\Gamma(1/3) =$	*2.53347 35234 51013 61316 73106 47644 54653 00106 6605*
$\Gamma(2/3) =$	*1.26523 57112 14154 74312 54572 37655 60126 23231 0245*
$e =$	*2.55760 52130 50535 51246 52773 42542 00471 72363 6166*
$1/e =$	*0.27426 53066 13167 46761 52726 75436 02440 52371 0336*
$e^2 =$	*7.30714 45615 23355 33460 63507 35040 32664 25356 5022*
$\gamma =$	*0.44742 14770 67666 06172 23215 74376 01002 51313 2552*
$\ln \pi =$	*1.11206 40443 47503 36413 65374 52661 52410 37511 4606*
$\phi =$	*1.47433 57156 27751 23701 27634 71401 40271 66710 1501*
$e^\gamma =$	*1.61772 13452 61152 65761 22477 36553 53327 17554 2126*
$e^{\pi/4} =$	*2.14275 31512 16162 52370 35530 11342 53525 44307 0217*
$\sin 1 =$	*0.65665 24436 04414 73402 03067 23644 11612 07474 1451*
$\cos 1 =$	*0.42450 50037 32406 42711 07022 14666 27320 70675 1232*
$\zeta(3) =$	*1.14735 00023 60014 20470 15613 42561 31715 10177 0662*
$\ln \phi =$	*0.36630 26256 61213 01145 13700 41004 52264 30700 4065*
$1/\ln \phi =$	*2.04776 60111 17144 41512 11436 16575 00355 43630 4065*
$-\ln \ln 2 =$	*0.27351 71233 67265 63650 17401 56637 26334 31455 5701*

Tables 1 and 2 contain several hitherto unpublished 40-digit values which have been furnished to the author by Dr. John W. Wrench, Jr.

For high-precision values of constants not found in this list, see J. Peters, *Ten Place Logarithms of the Numbers from 1 to 100000*, Appendix to Volume 1 (New York: F. Ungar Publ. Co., 1957); and *Handbook of Mathematical Functions*, ed. by M. Abramowitz and I. A. Stegun (Washington, D.C.: U. S. Govt. Printing Office, 1964), Chapter 1.

Table 3

Values of harmonic numbers, Bernoulli numbers, and Fibonacci numbers for small values of n.

n	H_n	B_n	F_n	n
0	0	1	0	0
1	1	−1/2	1	1
2	3/2	1/6	1	2
3	11/6	0	2	3
4	25/12	−1/30	3	4
5	137/60	0	5	5
6	49/20	1/42	8	6
7	363/140	0	13	7
8	761/280	−1/30	21	8
9	7129/2520	0	34	9
10	7381/2520	5/66	55	10
11	83711/27720	0	89	11
12	86021/27720	−691/2730	144	12
13	1145993/360360	0	233	13
14	1171733/360360	7/6	377	14
15	1195757/360360	0	610	15
16	2436559/720720	−3617/510	987	16
17	42142223/12252240	0	1597	17
18	14274301/4084080	43867/798	2584	18
19	275295799/77597520	0	4181	19
20	55835135/15519504	−174611/330	6765	20
21	18858053/5173168	0	10946	21
22	19093197/5173168	854513/138	17711	22
23	444316699/118982864	0	28657	23
24	1347822955/356948592	−236364091/2730	46368	24
25	34052522467/8923714800	0	75025	25

For any x, let $H_x = \sum_{n\geq 1}\left(\frac{1}{n} - \frac{1}{n+x}\right)$. Then

$$H_{1/2} = 2 - 2\ln 2,$$
$$H_{1/3} = 3 - \tfrac{1}{2}\pi/\sqrt{3} - \tfrac{3}{2}\ln 3,$$
$$H_{2/3} = \tfrac{3}{2} + \tfrac{1}{2}\pi/\sqrt{3} - \tfrac{3}{2}\ln 3,$$
$$H_{1/4} = 4 - \tfrac{1}{2}\pi - 3\ln 2,$$
$$H_{3/4} = \tfrac{4}{3} + \tfrac{1}{2}\pi - 3\ln 2,$$
$$H_{1/5} = 5 - \tfrac{1}{2}\pi\phi\sqrt{\frac{2+\phi}{5}} - \tfrac{1}{2}(3-\phi)\ln 5 - (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{2/5} = \tfrac{5}{2} - \tfrac{1}{2}\pi/\phi\sqrt{2+\phi} - \tfrac{1}{2}(2+\phi)\ln 5 + (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{3/5} = \tfrac{5}{3} + \tfrac{1}{2}\pi/\phi\sqrt{2+\phi} - \tfrac{1}{2}(2+\phi)\ln 5 + (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{4/5} = \tfrac{5}{4} + \tfrac{1}{2}\pi\phi\sqrt{\frac{2+\phi}{5}} - \tfrac{1}{2}(3-\phi)\ln 5 - (\phi - \tfrac{1}{2})\ln(2+\phi),$$
$$H_{1/6} = 6 - \tfrac{1}{2}\pi\sqrt{3} - 2\ln 2 - \tfrac{3}{2}\ln 3,$$
$$H_{5/6} = \tfrac{6}{5} + \tfrac{1}{2}\pi\sqrt{3} - 2\ln 2 - \tfrac{3}{2}\ln 3,$$

and, in general, when $0 < p < q$ (cf. exercise 1.2.9–19),

$$H_{p/q} = \frac{q}{p} - \tfrac{1}{2}\pi\cot\frac{p}{q}\pi - \ln 2q + 2\sum_{1\leq n<q/2}\cos\frac{2\pi np}{q}\ln\sin\frac{n}{q}\pi.$$

INDEX TO NOTATIONS

In the following formulas, letters which are not further qualified have the following significance:

j, k	integer-valued arithmetic expression
m, n	nonnegative integer-valued arithmetic expression
x, y, z	real-valued arithmetic expression
f	real-valued function
P	pointer-valued expression, i.e., either Λ or an address within a computer.
α	string of symbols

Formal symbolism	Meaning	Section reference
NODE(P)	the node (group of variables which are individually distinguished by their field names) whose address is P, P $\neq \Lambda$	2.1
F(P)	the variable in NODE(P) whose field name is F	2.1
CONTENTS(P)	contents of computer "word" whose address is P	2.1
LOC(V)	address of variable V within a computer	2.1
A_n or $A[n]$	the nth element of linear array A	1.1
A_{mn} or $A[m, n]$	the element in row m, column n of rectangular array A	1.1
$V \leftarrow E$	give variable V the value of expression E	1.1
$U \leftrightarrow V$	interchange the values of variables U and V	1.1
P $\Leftarrow$ AVAIL	set the value of pointer variable P to the address of a new node, or signal memory overflow if there is no room for a new node	2.2.3
AVAIL $\Leftarrow$ P	NODE(P) is returned to free storage; all its fields lose their identity	2.2.3

Formal symbolism	Meaning	Section reference
$\text{top}(S)$	node at the top of a nonempty stack S	2.2.1
$X \Leftarrow S$	pop up S to X: set $X \leftarrow \text{top}(S)$; delete $\text{top}(S)$ from nonempty stack S	2.2.1
$S \Leftarrow X$	push down X onto S: insert the value or group of values denoted by X as a new entry on the top of stack S	2.2.1
$(B \Rightarrow E_1; E_2)$	conditional expression: denotes E_1 if B is true, E_2 if B is false	8.1
δ_{jk}	Kronecker delta: $(j = k \Rightarrow 1;\ 0)$	1.2.6
$\sum_{R(k)} f(k)$	sum of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$\prod_{R(k)} f(k)$	product of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$\min_{R(k)} f(k)$	minimum value of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$\max_{R(k)} f(k)$	maximum value of all $f(k)$ such that k is an integer and relation $R(k)$ is true	1.2.3
$j \backslash k$	j divides k: $k \bmod j = 0$	1.2.4
$U \backslash V$	set difference: $\{x \mid x \text{ is in } U \text{ but not in } V\}$	
$\gcd(j, k)$	greatest common divisor of j and k: $(j = k = 0 \Rightarrow 0;\quad \max_{d \backslash j, d \backslash k} d)$	1.1
$\det(A)$	determinant of square matrix A	1.2.3
A^T	transpose of rectangular array A: $A^T[j, k] = A[k, j]$	1.2.3
x^y	x to the y power, x positive	1.2.2
α^R	left-right reversal of α	
x^k	x to the kth power: $\left(k \geq 0 \Rightarrow \prod_{0 \leq j < k} x;\quad 1/x^{-k}\right)$	1.2.2
$x^{\overline{k}}$	x upper k: $\left(k \geq 0 \Rightarrow x(x+1)\cdots(x+k-1) = \prod_{0 \leq j < k} (x+j);\quad 1/(x+k)^{\overline{-k}}\right)$	1.2.6

Formal symbolism	Meaning	Section reference
$x^{\underline{k}}$	x lower k: $\left(k \geq 0 \Rightarrow x(x-1)\cdots(x-k+1) = \prod_{0 \leq j < k} (x-j);\right.$ $\left. 1/(x-k)^{\overline{-k}}\right) = (-1)^k(-x)^{\overline{k}}$	1.2.6
$n!$	n factorial: $1 \cdot 2 \cdot \cdots \cdot n = n^{\underline{n}}$	1.2.5
$\binom{x}{k}$	binomial coefficient: $(k < 0 \Rightarrow 0;\ x^{\underline{k}}/k!)$	1.2.6
$\binom{n}{n_1, n_2, \ldots, n_m}$	multinomial coefficient, $n = n_1 + n_2 + \cdots + n_m$	1.2.6
$\begin{bmatrix} n \\ m \end{bmatrix}$	Stirling number of first kind: $\sum_{0<k_1<k_2<\cdots<k_{n-m}<n} k_1 k_2 \cdots k_{n-m}$	1.2.6
$\left\{ {n \atop m} \right\}$	Stirling number of second kind: $\sum_{0 \leq k_1 \leq k_2 \leq \cdots \leq k_{n-m} \leq m} k_1 k_2 \cdots k_{n-m}$	1.2.6
$\left\langle {n \atop k} \right\rangle$	Eulerian number	5.1.3.
$\{a \mid R(a)\}$	set of all a for which the relation $R(a)$ is true	
$\{a_1, \ldots, a_n\}$	the set or multiset $\{a_k \mid 1 \leq k \leq n\}$	
$\{n_1 \cdot a_1, \ldots, n_m \cdot a_m\}$	the multiset containing n_k occurrences of a_k	5.1.2
$\|M\|$	cardinality: the number of elements in the multiset M	
$\|x\|$	absolute value of x: $(x < 0 \Rightarrow -x;\ x)$	
$\|\alpha\|$	length of α	
$\lfloor x \rfloor$	floor of x, greatest integer function: $\max_{k \leq x} k$	1.2.4
$\lceil x \rceil$	ceiling of x, least integer function: $\min_{k \geq x} k$	1.2.4
$x \bmod y$	mod function: $(y = 0 \Rightarrow x;\ x - y\lfloor x/y \rfloor)$	1.2.4
$x \equiv y \pmod z$	relation of congruence: $x \bmod z = y \bmod z$	1.2.4

Formal symbolism	Meaning	Section reference
$\log_b x$	logarithm, base b, of x (real positive $b \neq 1$): $x = b^{\log_b x}$	1.2.2
$\ln x$	natural logarithm: $\log_e x$	1.2.2
$\exp x$	exponential of x: e^x	1.2.2
$\langle X_n \rangle$	the infinite sequence $X_0, X_1, X_2, \ldots$ (here n is a letter which is part of the symbol)	1.2.9
$f'(x)$	derivative of f at x	1.2.9
$f''(x)$	second derivative of f at x	1.2.10
$f^{(n)}(x)$	nth derivative: $(n = 0 \Rightarrow f(x);$ $g'(x) \quad$ where $\quad g(x) = f^{(n-1)}(x))$	1.2.11.2
$f(x) \Big\vert_y^z$	$f(z) - f(y)$	
$H_n^{(x)}$	$1 + 1/2^x + \cdots + 1/n^x = \sum_{1 \leq k \leq n} 1/k^x$	1.2.7
H_n	harmonic number: $H_n^{(1)}$	1.2.7
F_n	Fibonacci number: $(n \leq 1 \Rightarrow n; F_{n-1} + F_{n-2})$	1.2.8
B_n	Bernoulli number	1.2.11.2
$\Re(w)$	real part of the complex number w	
$\Im(w)$	imaginary part of the complex number w	
$\zeta(x)$	zeta function: $H_\infty^{(x)}$ when $x > 1$	1.2.7
$\Gamma(x)$	gamma function: $\gamma(x, \infty)$; $(x - 1)!$ when x is a positive integer	1.2.5
$\gamma(x, y)$	incomplete gamma function	1.2.11.3
γ	Euler's constant	1.2.7
e	base of natural logarithms: $\sum_{k \geq 0} 1/k!$	1.2.2
ϵ	empty string (of length zero)	
∞	infinity: larger than any number; or an artificially high key value	5
$M \uplus N$	sum of multisets M and N; e.g., $\{a, a, b\} \uplus \{a, b, c\} = \{a, a, a, b, b, c\}$	4.6.3
Λ	null link (pointer to no address)	2.1
$\emptyset$	empty set (set with no elements)	
ϕ	golden ratio, $\frac{1}{2}(1 + \sqrt{5})$	1.2.8

Formal symbolism	Meaning	Section reference
$\varphi(n)$	Euler's totient function: $\sum_{\substack{0\le k<n \\ \gcd(k,n)=1}} 1$	1.2.4
$O(f(n))$	big-oh of $f(n)$ as $n \to \infty$: a quantity whose magnitude is less than some constant times $f(n)$, for all large n	1.2.11.1
$O(f(x))$	big-oh of $f(x)$, for small x (or for x in some specified range)	1.2.11.1
(min x_1, ave x_2, max x_3, dev x_4)	a random variable having minimum value x_1, average ("expected") value x_2, maximum value x_3, standard deviation x_4	1.2.10
mean(g)	mean value of probability distribution represented by generating function g: $g'(1)$	1.2.10
var(g)	variance of probability distribution represented by generating function g: $g''(1) + g'(1) - g'(1)^2$	1.2.10
P$	address of symmetric order successor of NODE(P) in a binary tree	2.3.1
∎	end of algorithm, program, or proof	1.1
␣	one blank space	1.3.1
rA	register A (accumulator) of MIX	1.3.1
rX	register X (extension) of MIX	1.3.1
rI1, . . . , rI6	(index) registers I1, . . . , I6 of MIX	1.3.1
rJ	(jump) register J of MIX	1.3.1
(L:R)	partial field of MIX word, $0 \le L \le R \le 5$	1.3.1
OP ADDRESS,I(F)	notation for MIX instruction	1.3.1, 1.3.2
u	unit of time in MIX	1.3.1
*	"self" in MIXAL	1.3.2
0F, 1F, 2F, . . . , 9F	"forward" local symbol in MIXAL	1.3.2
0B, 1B, 2B, . . . , 9B	"backward" local symbol in MIXAL	1.3.2
0H, 1H, 2H, . . . , 9H	"here" local symbol in MIXAL	1.3.2

INDEX AND GLOSSARY

If you don't find it in the Index,
look very carefully through the entire catalogue.

—*Consumer's Guide*, Sears, Roebuck and Co. (1897)

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information; an answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.

Any inaccuracies in this index may be explained by the fact that it has been sorted with the help of a computer. For additional definitions of computer terminology, see Volume 1 and the *IFIP-ICC Vocabulary of Information Processing* (Amsterdam: North-Holland Publ. Co., 1966).

THE ART OF COMPUTER PROGRAMMING

PRE-FASCICLE 2A

A DRAFT OF SECTION 7.2.1.1: GENERATING ALL n-TUPLES

DONALD E. KNUTH *Stanford University*

ADDISON–WESLEY

Internet page `http://www-cs-faculty.stanford.edu/~knuth/taocp.html` contains current information about this book and related books.

See also `http://www-cs-faculty.stanford.edu/~knuth/sgb.html` for information about *The Stanford GraphBase*, including downloadable software for dealing with the graphs used in many of the examples in Chapter 7.

Zeroth printing (revision 6), 30 March 2002

PREFACE

I am grateful to all my friends,
and record here and now my most especial appreciation
to those friends who, after a decent interval,
stopped asking me, "How's the book coming?"

— PETER J. GOMES (1996)

THIS BOOKLET contains draft material that I'm circulating to experts in the field, in hopes that they can help remove its most egregious errors before too many other people see it. I am also, however, posting it on the Internet for courageous and/or random readers who don't mind the risk of reading a few pages that have not yet reached a very mature state. *Beware:* This material has not yet been proofread as thoroughly as the manuscripts of Volumes 1, 2, and 3 were at the time of their first printings. Those volumes, alas, were subsequently found to contain thousands of mistakes.

Given this caveat, I hope that my errors this time will not be so numerous and/or obtrusive that you will be discouraged from reading the material carefully. I did try to make it both interesting and authoritative, as far as it goes. But the field is so vast, I cannot hope to have surrounded it enough to corral it completely. Therefore I beg you to let me know about any deficiencies you discover.

To put the material in context, this is Section 7.2.1.1 of a long, long chapter on combinatorial algorithms. Chapter 7 will eventually fill three volumes (namely Volumes 4A, 4B, and 4C), assuming that I'm able to remain healthy. It will begin with a short review of graph theory, with emphasis on some highlights of significant graphs in The Stanford GraphBase (from which I will be drawing many examples). Then comes Section 7.1, which deals with the topic of bitwise manipulations. (I drafted about 60 pages about that subject in 1977, but those pages need extensive revision; meanwhile I've decided to work for awhile on the material that follows it, so that I can get a better feel for how much to cut.) Section 7.2 is about generating all possibilities, and it begins with Section 7.2.1: Generating Basic Combinatorial Patterns. That sets the stage for the main contents of this booklet, Section 7.2.1.1, where I get the ball rolling at last by dealing with the generation of n-tuples. Then will come Section 7.2.1.2 (about permutations), Section 7.2.1.3 (about combinations), etc. Section 7.2.2 will deal with backtracking in general. And so it will go on, if all goes well; an outline of the entire Chapter 7 as currently envisaged appears on the `taocp` webpage that is cited on page ii.

Even the apparently lowly topic of n-tuple generation turns out to be surprisingly rich, with ties to Sections 1.2.4, 1.3.3, 2.3.1, 2.3.4.2, 3.2.2, 3.5, 4.1, 4.3.1, 4.5.2, 4.5.3, 4.6.1, 4.6.2, 4.6.4, 5.2.1, and 6.3 of the first three volumes. I strongly believe in building up a firm foundation, so I have discussed this topic much more thoroughly than I will be able to do with material that is newer or less basic. To my surprise, I came up with 112 exercises, a new record, even though — believe it or not — I had to eliminate quite a bit of the interesting material that appears in my files.

Some of the material is new, to the best of my knowledge, although I will not be at all surprised to learn that my own little "discoveries" have been discovered before. Please look, for example, at the exercises that I've classed as research problems (rated with difficulty level 46 or higher), namely exercises 43, 46, 47, 53, 55, 62, 66, and 83. Are these problems still open? The questions in exercises 53 and 83 might not have been posed previously, but they seem to deserve attention. Please let me know if you know of a solution to any of these intriguing problems. And of course if no solution is known today but you do make progress on any of them in the future, I hope you'll let me know.

I urgently need your help also with respect to some exercises that I made up as I was preparing this material. I certainly don't like to get credit for things that have already been published by others, and most of these results are quite natural "fruits" that were just waiting to be "plucked." Therefore please tell me if you know who I should have credited, with respect to the ideas found in exercises 15, 16, 31, 37, 38, 69, 73, 76, 86, 87, 89, 90, and/or 109.

I shall happily pay a finder's fee of \$2.56 for each error in this draft when it is first reported to me, whether that error be typographical, technical, or historical. The same reward holds for items that I forgot to put in the index. And valuable suggestions for improvements to the text are worth 32¢ each. (Furthermore, if you find a better solution to an exercise, I'll actually reward you with immortal glory instead of mere money, by publishing your name in the eventual book:–)

Happy reading!

Stanford, California D. E. K.
August 2001 (revised, September 2001)

7.2. GENERATING ALL POSSIBILITIES

All present or accounted for, sir.
— Traditional American military saying

All present and correct, sir.
— Traditional British military saying

7.2.1. Generating Basic Combinatorial Patterns

Our goal in this section is to study methods for running through all of the possibilities in some combinatorial universe, because we often face problems in which an exhaustive examination of all cases is necessary or desirable. For example, we might want to look at all permutations of a given set.

Some authors call this the task of *enumerating* all of the possibilities; but that's not quite the right word, because "enumeration" most often means that we merely want to *count* the total number of cases, not that we actually want to look at them all. If somebody asks you to enumerate the permutations of $\{1, 2, 3\}$, you are quite justified in replying that the answer is $3! = 6$; you needn't give the more complete answer $\{123, 132, 213, 231, 312, 321\}$.

Other authors speak of *listing* all the possibilities; but that's not such a great word either. No sensible person would want to make a list of the $10! = 3{,}628{,}800$ permutations of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ by printing them out on thousands of sheets of paper, nor even by writing them all in a computer file. All we really want is to have them present momentarily in some data structure, so that a program can examine each permutation one at a time.

So we will speak of *generating* all of the combinatorial objects that we need, and *visiting* each object in turn. Just as we studied algorithms for tree traversal in Section 2.3.1, where the goal was to visit every node of a tree, we turn now to algorithms that systematically traverse a combinatorial space of possibilities.

He's got 'em on the list—
he's got 'em on the list;
And they'll none of 'em be missed—
they'll none of 'em be missed.
— WILLIAM S. GILBERT, *The Mikado* (1885)

7.2.1.1. Generating all n-tuples. Let's start small, by considering how to run through all 2^n strings that consist of n binary digits. Equivalently, we want to visit all n-tuples $(a_1, \ldots, a_n)$ where each a_j is either 0 or 1. This task is also, in essence, equivalent to examining all subsets of a given set $\{x_1, \ldots, x_n\}$, because we can say that x_j is in the subset if and only if $a_j = 1$.

Of course such a problem has an absurdly simple solution. All we need to do is start with the binary number $(0 \ldots 00)_2 = 0$ and repeatedly add 1 until we reach $(1 \ldots 11)_2 = 2^n - 1$. We will see, however, that even this utterly trivial problem has astonishing points of interest when we look into it more deeply. And our study of n-tuples will pay off later when we turn to the generation of more difficult kinds of patterns.

In the first place, we can see that the binary-notation trick extends to other kinds of n-tuples. If we want, for example, to generate all $(a_1, \ldots, a_n)$ in which each a_j is one of the decimal digits $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, we can simply count from $(0 \ldots 00)_{10} = 0$ to $(9 \ldots 99)_{10} = 10^n - 1$ in the decimal number system. And if we want more generally to run through all cases in which

$$0 \le a_j < m_j \qquad \text{for } 1 \le j \le n, \tag{1}$$

where the upper limits m_j might be different in different components of the vector $(a_1, \ldots, a_n)$, the task is essentially the same as repeatedly adding unity to the number

$$\begin{bmatrix} a_1, & a_2, & \ldots, & a_n \\ m_1, & m_2, & \ldots, & m_n \end{bmatrix} \tag{2}$$

in a mixed-radix number system; see Eq. 4.1–(9) and exercise 4.3.1–9.

We might as well pause to describe the process more formally:

Algorithm M (*Mixed-radix generation*). This algorithm visits all n-tuples that satisfy (1), by repeatedly adding 1 to the mixed-radix number in (2) until overflow occurs. Auxiliary variables a_0 and m_0 are introduced for convenience.

M1. [Initialize.] Set $a_j \leftarrow 0$ for $0 \le j \le n$, and set $m_0 \leftarrow 2$.

M2. [Visit.] Visit the n-tuple $(a_1, \ldots, a_n)$. (The program that wants to examine all n-tuples now does its thing.)

M3. [Prepare to add one.] Set $j \leftarrow n$.

M4. [Carry if necessary.] If $a_j = m_j - 1$, set $a_j \leftarrow 0$, $j \leftarrow j - 1$, and repeat this step.

M5. [Increase, unless done.] If $j = 0$, terminate the algorithm. Otherwise set $a_j \leftarrow a_j + 1$ and go back to step M2. ∎

Algorithm M is simple and straightforward, but we shouldn't forget that nested loops are even simpler, when n is a fairly small constant. When $n = 4$, we could for example write out the following instructions:

```
For a1 = 0, 1, ..., m1 − 1 (in this order) do the following:
  For a2 = 0, 1, ..., m2 − 1 (in this order) do the following:
    For a3 = 0, 1, ..., m3 − 1 (in this order) do the following:
      For a4 = 0, 1, ..., m4 − 1 (in this order) do the following:
        Visit (a1, a2, a3, a4).
```
(3)

These instructions are equivalent to Algorithm M, and they are easily expressed in any programming language.

Gray binary code. Algorithm M runs through all $(a_1, \ldots, a_n)$ in lexicographic order, as in a dictionary. But there are many situations in which we prefer to visit those n-tuples in some other order. The most famous alternative arrangement is the so-called Gray binary code, which lists all 2^n strings of n bits in such a way

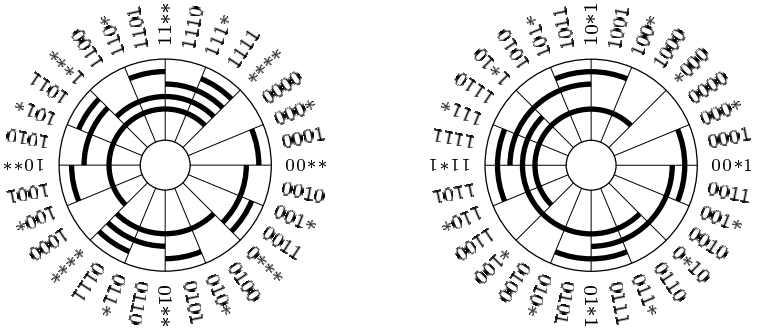


Fig. 10. (a) Lexicographic binary code. (b) Gray binary code.

that only one bit changes each time, in a simple and regular way. For example, the Gray binary code for $n = 4$ is

$$0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100,$$
$$1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000. \tag{4}$$

Such codes are especially important in applications where analog information is being converted to digital or vice versa. For example, suppose we want to identify our current position on a rotating disk that has been divided into 16 sectors, using four sensors that each distinguish black from white. If we use lexicographic order to mark the tracks from 0000 to 1111, as in Fig. 10(a), wildly inaccurate measurements can occur at the boundaries between sectors; but the code in Fig. 10(b) never gives a bad reading.

Gray binary code can be defined in many equivalent ways. For example, if Γ_n stands for the Gray binary sequence of n-bit strings, we can define Γ_n recursively by the two rules

$$\begin{aligned} \Gamma_0 &= \epsilon; \\ \Gamma_{n+1} &= 0\Gamma_n,\ 1\Gamma_n^R. \end{aligned} \tag{5}$$

Here ϵ denotes the empty string, $0\Gamma_n$ denotes the sequence Γ_n with 0 prefixed to each string, and $1\Gamma_n^R$ denotes the sequence Γ_n in *reverse order* with 1 prefixed to each string. Since the last string of Γ_n equals the first string of Γ_n^R, it is clear from (5) that exactly one bit changes in every step of Γ_{n+1} if Γ_n enjoys the same property.

Another way to define the sequence $\Gamma_n = g(0), g(1), \ldots, g(2^n-1)$ is to give an explicit formula for its individual elements $g(k)$. Indeed, since Γ_{n+1} begins with $0\Gamma_n$, the infinite sequence

$$\begin{aligned} \Gamma_\infty &= g(0), g(1), g(2), g(3), g(4), \ldots \\ &= (0)_2, (1)_2, (11)_2, (10)_2, (110)_2, \ldots \end{aligned} \tag{6}$$

is a permutation of all the nonnegative integers, if we regard each string of 0s and 1s as a binary integer with leading 0s suppressed. Then Γ_n consists of the first 2^n elements of (6), converted to n-bit strings by means of leading 0s if needed.

When $k = 2^n + r$, where $0 \le r < 2^n$, relation (5) tells us that $g(k)$ is equal to $2^n + g(2^n - 1 - r)$. Therefore we can prove by induction on n that the integer k whose binary representation is $(\dots b_2 b_1 b_0)_2$ has a Gray binary equivalent $g(k)$ with the representation $(\dots a_2 a_1 a_0)_2$, where

$$a_j = b_j \oplus b_{j+1}, \qquad \text{for } j \ge 0. \tag{7}$$

(See exercise 6.) For example, $g\big((111001000011)_2\big) = (100101100010)_2$. Conversely, if $g(k) = (\dots a_2 a_1 a_0)_2$ is given, we can find $k = (\dots b_2 b_1 b_0)_2$ by inverting the system of equations (7), obtaining

$$b_j = a_j \oplus a_{j+1} \oplus a_{j+2} \oplus \cdots, \qquad \text{for } j \ge 0; \tag{8}$$

this infinite sum is really finite because $a_{j+t} = 0$ for all large t.

One of the many pleasant consequences of Eq. (7) is that $g(k)$ can be computed very easily with bitwise arithmetic:

$$g(k) = k \oplus \lfloor k/2 \rfloor. \tag{9}$$

Similarly, the inverse function in (8) satisfies

$$g^{[-1]}(l) = l \oplus \lfloor l/2 \rfloor \oplus \lfloor l/4 \rfloor \oplus \cdots; \tag{10}$$

this function, however, requires more computation (see exercise 7.1–00). We can also deduce from (7) that, if k and k' are any nonnegative integers,

$$g(k \oplus k') = g(k) \oplus g(k'). \tag{11}$$

Yet another consequence is that the $(n+1)$-bit Gray binary code can be written

$$\Gamma_{n+1} = 0\Gamma_n,\ 1(\Gamma_n \oplus 10\dots0);$$

this pattern is evident, for example, in (4). Comparing with (5), we see that reversing the order of Gray binary code is equivalent to complementing the first bit:

$$\Gamma_n^R = \Gamma_n \oplus 1\overbrace{0\dots0}^{n-1}. \tag{12}$$

The exercises below show that the function $g(k)$ defined in (7), and its inverse $g^{[-1]}$ defined in (8), have many further properties and applications of interest. Sometimes we think of these as functions taking binary strings to binary strings; at other times we regard them as functions from integers to integers, via binary notation, with leading zeros irrelevant.

Gray binary code is named after Frank Gray, a physicist who became famous for helping to devise the method long used for compatible color television broadcasting [*Bell System Tech. J.* **13** (1934), 464–515]. He invented Γ_n for applications to pulse code modulation, a method for analog transmission of digital signals [see *Bell System Tech. J.* **30** (1951), 38–40; *U.S. Patent 2632058* (17 March 1953); W. R. Bennett, *Introduction to Signal Transmission* (1971), 238–240]. But the idea of "Gray binary code" was known long before he worked on it; for example, it appeared in *U.S. Patent 2307868* by George Stibitz (12 January 1943). More significantly, Γ_5 was used in a telegraph machine demonstrated in 1878 by Émile Baudot, after whom the term "baud" was later named. At

about the same time, a similar but less systematic code for telegraphy was independently devised by Otto Schäffler [see *Journal Télégraphique* **4** (1878), 252–253; *Annales Télégraphiques* **6** (1879), 361, 382–383].*

In fact, Gray binary code is implicitly present in a classic toy that has fascinated people for centuries, now generally known as the "Chinese ring puzzle" in English, although Englishmen used to call it the "tiring irons." Fig. 11 shows a seven-ring example. The challenge is to remove the rings from the bar, and the rings are interlocked in such a way that only two basic types of move are possible (although this may not be immediately apparent from the illustration):

a) The rightmost ring can be removed or replaced at any time;
b) Any other ring can be removed or replaced if and only if the ring to its right is on the bar and all rings to the right of that one are off.

We can represent the current state of the puzzle in binary notation, writing 1 if a ring is on the bar and 0 if it is off; thus Fig. 11 shows the rings in state 1011000. (The second ring from the left is encoded as 0, because it lies entirely above the bar.)

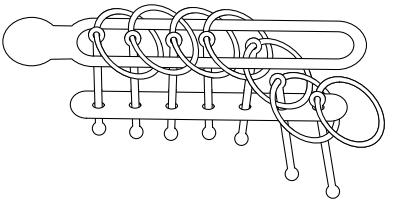


Fig. 11.
The Chinese ring puzzle.

A French magistrate named Louis Gros demonstrated an explicit connection between Chinese rings and binary numbers, in a booklet called *Théorie du Baguenodier* [sic] (Lyon: Aimé Vingtrinier, 1872) that was published anonymously. If the rings are in state $a_{n-1}\ldots a_0$, and if we define the binary number $k = (b_{n-1}\ldots b_0)_2$ by Eq. (8), he showed that exactly k more steps are necessary and sufficient to solve the puzzle. Thus Gros is the true inventor of Gray binary code.

Certainly no home should be without
this fascinating, historic, and instructive puzzle.
— HENRY E. DUDENEY (1901)

When the rings are in any state other than $00\ldots0$ or $10\ldots0$, exactly two moves are possible, one of type (a) and one of type (b). Only one of these moves advances toward the desired goal; the other is a step backward that will need to be undone. A type (a) move changes k to $k \oplus 1$; thus we want to do it when k is odd, since this will decrease k. A type (b) move from a position that ends in $(10^{j-1})_2$ for $1 \le j < n$ changes k to $k \oplus (1^{j+1})_2 = k \oplus (2^{j+1} - 1)$. When k is

* Some authors have asserted that Gray code was invented by Elisha Gray, who developed a printing telegraph machine at the same time as Baudot and Schäffler. Such claims are untrue, although Elisha did get a raw deal with respect to priority for inventing the telephone [see L. W. Taylor, *Amer. Physics Teacher* **5** (1937), 243–251].

even, we want to decrease k by 1, which means that k must be a multiple of 2^j but not a multiple of 2^{j+1}; in other words,

$$j = \rho(k), \tag{13}$$

where ρ is the "ruler function" of Eq. 7.1–(00). Therefore the rings follow a nice pattern when the puzzle is solved properly: If we number them 0, 1, ..., $n-1$ (starting at the free end), the sequence of ring moves on or off the bar is the sequence of numbers that ends with ..., $\rho(4)$, $\rho(3)$, $\rho(2)$, $\rho(1)$.

Going backwards, successively putting rings on or off until we reach the ultimate state $10\ldots0$ (which, as John Wallis observed in 1693, is more difficult to reach than the supposedly harder state $11\ldots1$), yields an algorithm for counting in Gray binary code:

Algorithm G (*Gray binary generation*). This algorithm visits all binary n-tuples $(a_{n-1},\ldots,a_1,a_0)$ by starting with $(0,\ldots,0,0)$ and changing only one bit at a time, also maintaining a parity bit a_∞ such that

$$a_\infty = a_{n-1} \oplus \cdots \oplus a_1 \oplus a_0. \tag{14}$$

It successively complements bits $\rho(1)$, $\rho(2)$, $\rho(3)$, ..., $\rho(2^n-1)$ and then stops.

G1. [Initialize.] Set $a_j \leftarrow 0$ for $0 \le j < n$; also set $a_\infty \leftarrow 0$.

G2. [Visit.] Visit the n-tuple $(a_{n-1},\ldots,a_1,a_0)$.

G3. [Change parity.] Set $a_\infty \leftarrow 1 - a_\infty$.

G4. [Choose j.] If $a_\infty = 1$, set $j \leftarrow 0$. Otherwise let $j \ge 1$ be minimum such that $a_{j-1} = 1$. (After the kth time we have performed this step, $j = \rho(k)$.)

G5. [Complement coordinate j.] Terminate if $j = n$; otherwise set $a_j \leftarrow 1 - a_j$ and return to G2. ∎

The parity bit a_∞ comes in handy if we are computing a sum like

$$X_{000} - X_{100} - X_{010} + X_{110} - X_{001} + X_{101} + X_{011} - X_{111}$$

or

$$X_\emptyset - X_a - X_b + X_{ab} - X_c + X_{ac} + X_{bc} - X_{abc},$$

where the sign depends on the parity of a binary string or the number of elements in a subset. Such sums arise frequently in "inclusion-exclusion" formulas such as Eq. 1.3.3–(29). The parity bit is also necessary, for efficiency: Without it we could not easily choose between the two ways of determining j, which correspond to performing a type (a) or type (b) move in the Chinese ring puzzle. But the most important feature of Algorithm G is that step G5 makes only a single coordinate change, so that only a simple change is usually needed to the terms X that we are summing or to whatever other structures we are concerned with as we visit each n-tuple.

It is impossible, of course, to remove all ambiguity in the lowest-order digit except by a scheme like one the Irish railways are said to have used of removing the last car of every train because it is too susceptible to collision damage.

— G. R. STIBITZ and J. A. LARRIVEE, *Mathematics and Computers* (1957)

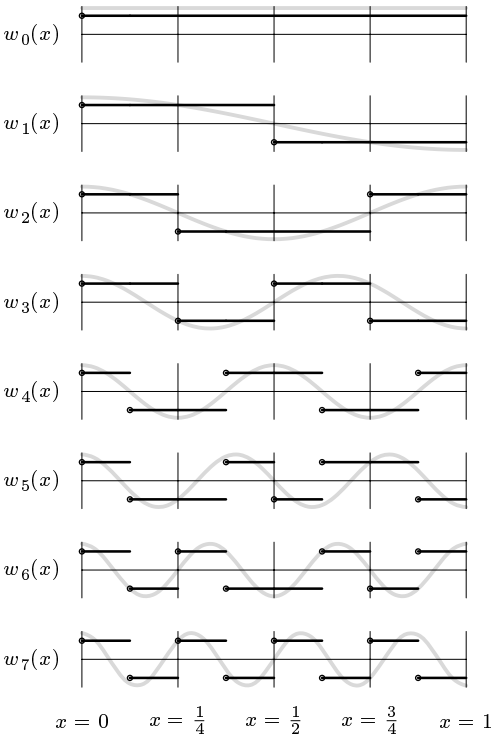


Fig. 12. Walsh functions $w_k(x)$ for $0 \le k < 8$, with the analogous trigonometric functions $\sqrt{2}\cos k\pi x$ shown in gray for comparison.

Another key property of Gray binary code was discovered by J. L. Walsh in connection with an important sequence of functions now known as *Walsh functions* [see *Amer. J. Math.* **45** (1923), 5–24]. Let $w_0(x) = 1$ for all real numbers x, and

$$w_k(x) = (-1)^{\lfloor 2x \rfloor \lceil k/2 \rceil} w_{\lfloor k/2 \rfloor}(2x), \qquad \text{for } k > 0. \tag{15}$$

For example, $w_1(x) = (-1)^{\lfloor 2x \rfloor}$ changes sign whenever x is an integer or an integer plus $\frac{1}{2}$. It follows that $w_k(x) = w_k(x+1)$ for all k, and that $w_k(x) = \pm 1$ for all x. More significantly, $w_k(0) = 1$ and $w_k(x)$ *has exactly* k *sign changes in the interval* $(0\,..\,1)$, so that it approaches $(-1)^k$ as x approaches 1 from the left. Therefore $w_k(x)$ behaves rather like a trigonometric function $\cos k\pi x$ or $\sin k\pi x$, and we can represent other functions as a linear combination of Walsh functions in much the same way as they are traditionally represented as Fourier series. This fact, together with the simple discrete nature of $w_k(x)$, makes Walsh functions extremely useful in computer calculations related to information transmission, image processing, and many other applications.

Fig. 12 shows the first eight Walsh functions together with their trigonometric cousins. Engineers commonly call $w_k(x)$ the Walsh function of *sequency* k, by analogy with the fact that $\cos k\pi x$ and $\sin k\pi x$ have *frequency* $k/2$. [See, for example, the book *Sequency Theory: Foundations and Applications* (New York: Academic Press, 1977), by H. F. Harmuth.]

Although Eq. (15) may look formidable at first glance, it actually provides an easy way to see by induction why $w_k(x)$ has exactly k sign changes as claimed. If k is even, say $k = 2l$, we have $w_{2l}(x) = w_l(2x)$ for $0 \le x < \frac{1}{2}$; the effect is simply to compress the function $w_l(x)$ into half the space, so $w_{2l}(x)$ has accumulated l sign changes so far. Then $w_{2l}(x) = (-1)^l w_l(2x) = (-1)^l w_l(2x-1)$ in the range $\frac{1}{2} \le x < 1$; this concatenates another copy of $w_l(2x)$, flipping the sign if necessary to avoid a sign change at $x = \frac{1}{2}$. The function $w_{2l+1}(x)$ is similar, but it *forces* a sign change when $x = \frac{1}{2}$.

What does this have to do with Gray binary code? Walsh discovered that his functions could all be expressed neatly in terms of simpler functions called *Rademacher functions* [Hans Rademacher, *Math. Annalen* **87** (1922), 112–138],

$$r_k(x) \;=\; (-1)^{\lfloor 2^k x \rfloor}, \tag{16}$$

which take the value $(-1)^{c-k}$ when $(\ldots c_2 c_1 c_0 . c_{-1} c_{-2} \ldots)_2$ is the binary representation of x. Indeed, we have $w_1(x) = r_1(x)$, $w_2(x) = r_1(x) r_2(x)$, $w_3(x) = r_2(x)$, and in general

$$w_k(x) = \prod_{j \ge 0} r_{j+1}(x)^{b_j \oplus b_{j+1}} \qquad \text{when } k = (b_{n-1} \ldots b_1 b_0)_2. \tag{17}$$

(See exercise 33.) Thus the exponent of $r_{j+1}(x)$ in $w_k(x)$ is the jth bit of the Gray binary number $g(k)$, according to (7), and we have

$$w_k(x) \;=\; r_{\rho(k)+1}(x)\, w_{k-1}(x), \qquad \text{for } k > 0. \tag{18}$$

Equation (17) implies the handy formula

$$w_k(x)\, w_{k'}(x) \;=\; w_{k \oplus k'}(x), \tag{19}$$

which is much simpler than the corresponding product formulas for sines and cosines. This identity follows easily because $r_j(x)^2 = 1$ for all j and x, hence $r_j(x)^{a \oplus b} = r_j(x)^{a+b}$. It implies in particular that $w_k(x)$ is *orthogonal* to $w_{k'}(x)$ when $k \ne k'$, in the sense that the average value of $w_k(x) w_{k'}(x)$ is zero. We also can use (17) to define $w_k(x)$ for fractional values of k like $1/2$ or $13/8$.

The *Walsh transform* of 2^n numbers $(X_0, \ldots, X_{2^n-1})$ is the vector defined by the equation $(x_0, \ldots, x_{2^n-1})^T = W_n (X_0, \ldots, X_{2^n-1})^T$, where W_n is the $2^n \times 2^n$ matrix having $w_j(k/2^n)$ in row j and column k, for $0 \le j, k < 2^n$. For example, Fig. 12 tells us that the Walsh transform when $n = 3$ is

$$\begin{pmatrix} x_{000} \\ x_{001} \\ x_{010} \\ x_{011} \\ x_{100} \\ x_{101} \\ x_{110} \\ x_{111} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & \overline{1} & \overline{1} & \overline{1} & \overline{1} \\ 1 & 1 & \overline{1} & \overline{1} & \overline{1} & \overline{1} & 1 & 1 \\ 1 & 1 & \overline{1} & \overline{1} & 1 & 1 & \overline{1} & \overline{1} \\ 1 & \overline{1} & \overline{1} & 1 & 1 & \overline{1} & \overline{1} & 1 \\ 1 & \overline{1} & \overline{1} & 1 & \overline{1} & 1 & 1 & \overline{1} \\ 1 & \overline{1} & 1 & \overline{1} & \overline{1} & 1 & \overline{1} & 1 \\ 1 & \overline{1} & 1 & \overline{1} & 1 & \overline{1} & 1 & \overline{1} \end{pmatrix} \begin{pmatrix} X_{000} \\ X_{001} \\ X_{010} \\ X_{011} \\ X_{100} \\ X_{101} \\ X_{110} \\ X_{111} \end{pmatrix}. \tag{20}$$

(Here $\overline{1}$ stands for -1, and the subscripts are conveniently regarded as binary strings 000–111 instead of as the integers 0–7.) The *Hadamard transform* is defined similarly, but with the matrix H_n in place of W_n, where H_n has $(-1)^{j\cdot k}$ in row j and column k; here '$j\cdot k$' denotes the dot product $a_{n-1}b_{n-1}+\cdots+a_0b_0$ of the binary representations $j=(a_{n-1}\ldots a_0)_2$ and $k=(b_{n-1}\ldots b_0)_2$. For example, the Hadamard transform for $n=3$ is

$$\begin{pmatrix} x'_{000}\\ x'_{001}\\ x'_{010}\\ x'_{011}\\ x'_{100}\\ x'_{101}\\ x'_{110}\\ x'_{111}\end{pmatrix} = \begin{pmatrix} 1&1&1&1&1&1&1&1\\ 1&\overline{1}&1&\overline{1}&1&\overline{1}&1&\overline{1}\\ 1&1&\overline{1}&\overline{1}&1&1&\overline{1}&\overline{1}\\ 1&\overline{1}&\overline{1}&1&1&\overline{1}&\overline{1}&1\\ 1&1&1&1&\overline{1}&\overline{1}&\overline{1}&\overline{1}\\ 1&\overline{1}&1&\overline{1}&\overline{1}&1&\overline{1}&1\\ 1&1&\overline{1}&\overline{1}&\overline{1}&\overline{1}&1&1\\ 1&\overline{1}&\overline{1}&1&\overline{1}&1&1&\overline{1}\end{pmatrix} \begin{pmatrix} X_{000}\\ X_{001}\\ X_{010}\\ X_{011}\\ X_{100}\\ X_{101}\\ X_{110}\\ X_{111}\end{pmatrix}. \tag{21}$$

This is the same as the discrete Fourier transform on an n-dimensional cube, Eq. 4.6.4–(38), and we can evaluate it quickly "in place" by adapting the method of Yates discussed in Section 4.6.4:

Given	First step	Second step	Third step
X_{000}	$X_{000}+X_{001}$	$X_{000}+X_{001}+X_{010}+X_{011}$	$X_{000}+X_{001}+X_{010}+X_{011}+X_{100}+X_{101}+X_{110}+X_{111}$
X_{001}	$X_{000}-X_{001}$	$X_{000}-X_{001}+X_{010}-X_{011}$	$X_{000}-X_{001}+X_{010}-X_{011}+X_{100}-X_{101}+X_{110}-X_{111}$
X_{010}	$X_{010}+X_{011}$	$X_{000}+X_{001}-X_{010}-X_{011}$	$X_{000}+X_{001}-X_{010}-X_{011}+X_{100}+X_{101}-X_{110}-X_{111}$
X_{011}	$X_{010}-X_{011}$	$X_{000}-X_{001}-X_{010}+X_{011}$	$X_{000}-X_{001}-X_{010}+X_{011}+X_{100}-X_{101}-X_{110}+X_{111}$
X_{100}	$X_{100}+X_{101}$	$X_{100}+X_{101}+X_{110}+X_{111}$	$X_{000}+X_{001}+X_{010}+X_{011}-X_{100}-X_{101}-X_{110}-X_{111}$
X_{101}	$X_{100}-X_{101}$	$X_{100}-X_{101}+X_{110}-X_{111}$	$X_{000}-X_{001}+X_{010}-X_{011}-X_{100}+X_{101}-X_{110}+X_{111}$
X_{110}	$X_{110}+X_{111}$	$X_{100}+X_{101}-X_{110}-X_{111}$	$X_{000}+X_{001}-X_{010}-X_{011}-X_{100}-X_{101}+X_{110}+X_{111}$
X_{111}	$X_{110}-X_{111}$	$X_{100}-X_{101}-X_{110}+X_{111}$	$X_{000}-X_{001}-X_{010}+X_{011}-X_{100}+X_{101}+X_{110}-X_{111}$

Notice that the rows of H_3 are a permutation of the rows of W_3. This is true in general, so we can obtain the Walsh transform by permuting the elements of the Hadamard transform. Exercise 36 discusses the details.

Going faster. When we're running through 2^n possibilities, we usually want to reduce the computation time as much as possible. Algorithm G needs to complement only one bit a_j per visit to $(a_{n-1},\ldots,a_0)$, but it loops in step G4 while choosing an appropriate value of j. Another approach has been suggested by Gideon Ehrlich [*JACM* **20** (1973), 500–513], who introduced the notion of *loopless* combinatorial generation: With a loopless algorithm, the number of operations performed between successive visits is required to be bounded in advance, so there never is a long wait before a new pattern has been generated.

We learned some tricks in Section 7.1 about quick ways to determine the number of leading or trailing 0s in a binary number. Those methods could be used in step G4 to make Algorithm G loopless, assuming that n isn't unreasonably large. But Ehrlich's method is quite different, and much more versatile, so it provides us with a new weapon in our arsenal of techniques for efficient computation. Here is how his approach can be used to generate binary n-tuples.

Algorithm L (*Loopless Gray binary generation*). This algorithm, like Algorithm G, visits all binary n-tuples $(a_{n-1},\ldots,a_0)$ in the order of the Gray binary code. But instead of maintaining a parity bit, it uses an array of "focus pointers" $(f_n,\ldots,f_0)$, whose significance is discussed below.

L1. [Initialize.] Set $a_j \leftarrow 0$ and $f_j \leftarrow j$ for $0 \le j < n$; also set $f_n \leftarrow n$. (A loopless algorithm is allowed to have loops in its initialization step, as long as the initial setup is reasonably efficient; after all, every program needs to be loaded and launched.)

L2. [Visit.] Visit the n-tuple $(a_{n-1},\ldots,a_1,a_0)$.

L3. [Choose j.] Set $j \leftarrow f_0$, $f_0 \leftarrow 0$. (If this is the kth time we are performing the present step, j is now equal to $\rho(k)$.) Terminate if $j = n$; otherwise set $f_j \leftarrow f_{j+1}$ and $f_{j+1} \leftarrow j+1$.

L4. [Complement coordinate j.] Set $a_j \leftarrow 1 - a_j$ and return to L2. ∎

For example, the computation proceeds as follows when $n = 4$. Elements a_j have been underlined in this table if the corresponding bit b_j is 1 in the binary string $b_3b_2b_1b_0$ such that $a_3a_2a_1a_0 = g(b_3b_2b_1b_0)$:

a_3	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
a_2	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
a_1	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0
a_0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
f_3	3	3	3	3	3	3	3	3	4	4	4	4	3	3	3	3
f_2	2	2	2	2	3	3	2	2	2	2	2	2	4	4	2	2
f_1	1	1	2	1	1	1	3	1	1	1	2	1	1	1	4	1
f_0	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0	4

Although the binary number $k = (b_{n-1}\ldots b_0)_2$ never appears explicitly in Algorithm L, the focus pointers f_j represent it implicitly in a clever way, so that we can repeatedly form $g(k) = (a_{n-1}\ldots a_0)_2$ by complementing bit $a_{\rho(k)}$ as we should. Let's say that a_j is *passive* when it is underlined, *active* otherwise. Then the focus pointers satisfy the following invariant relations:

1) If a_j is passive and a_{j-1} is active, then f_j is the smallest index $j' > j$ such that $a_{j'}$ is active. (Bits a_{-1} and a_n are considered to be active for purposes of this rule, although they aren't really present in the algorithm.)
2) Otherwise $f_j = j$.

Thus, the rightmost element a_j of a block of passive elements $a_{i-1}\ldots a_{j+1}a_j$ has a focus f_j that points to the element a_i just to the left of that block. All other elements a_j have f_j pointing to themselves.

In these terms, the first two operations '$j \leftarrow f_0$, $f_0 \leftarrow 0$' in step L3 are equivalent to saying, "Set j to the index of the rightmost active element, and activate all elements to the right of a_j." Notice that if $f_0 = 0$, the operation $f_0 \leftarrow 0$ is redundant; but it doesn't do any harm. The other two operations of L3, '$f_j \leftarrow f_{j+1}$, $f_{j+1} \leftarrow j+1$', are equivalent to saying, "Make a_j passive," because we know that a_j and a_{j-1} are both active at this point in the computation.

(Again the operation $f_{j+1} \leftarrow j+1$ might be harmlessly redundant.) The net effect of activation and passivation is therefore equivalent to counting in binary notation, as in Algorithm M, with 1-bits passive and 0-bits active.

Algorithm L is almost blindingly fast, because it does only five assignment operations and one test for termination between each visit to a generated n-tuple. But we can do even better. In order to see how, let's consider an application to recreational linguistics: Rudolph Castown, in *Word Ways* **1** (1968), 165–169, noted that all 16 of the ways to intermix the letters of `sins` with the corresponding letters of `fate` produce words that are found in a sufficiently large dictionary of English: `sine`, `sits`, `site`, etc.; and all but three of these words (namely `fane`, `fite`, and `sats`) are sufficiently common as to be unquestionably part of standard English. Therefore it is natural to ask the analogous question for five-letter words: What two strings of five letters will produce the maximum number of words in the Stanford GraphBase, when letters in corresponding positions are swapped in all 32 possible ways?

To answer this question, we need not examine all $\binom{26}{2}^5 = 3{,}625{,}908{,}203{,}125$ essentially different pairs of strings; it suffices to look at all $\binom{5757}{2} = 16{,}568{,}646$ pairs of words in the GraphBase, provided that at least one of those pairs produces at least 17 words, because every set of 17 or more five-letter words obtainable from two five-letter strings must contain two that are "antipodal" (with no corresponding letters in common). For every such pair, we want to determine as rapidly as possible whether the 32 possible subset-swaps produce a significant number of English words.

Every 5-letter word can be represented as a 25-bit number using 5 bits per letter, from `"a"` = 00000 to `"z"` = 11001. A table of 2^{25} bits or bytes will then determine quickly whether a given five-letter string is a word. So the problem is reduced to generating the 32 bit patterns of the potential words obtainable by mixing the letters of two given words, and looking those patterns up in the table. We can proceed as follows, for each pair of 25-bit words w and w':

W1. [Check the difference.] Set $z \leftarrow w \oplus w'$. Reject the word pair (w, w') if $\big((z-m)\oplus z\oplus m\big)\wedge m' \neq 0$, where $m = 2^{20}+2^{15}+2^{10}+2^5+1$ and $m' = 2^5 m$; this test eliminates cases where w and w' have a common letter in some position. (See 7.1–(oo); it turns out that 10,614,085 of the 16,568,646 word pairs have no such common letters.)

W2. [Form individual masks.] Set $m_0 \leftarrow z \wedge (2^5-1)$, $m_1 \leftarrow z \wedge (2^{10}-2^5)$, $m_2 \leftarrow z \wedge (2^{15}-2^{10})$, $m_3 \leftarrow z \wedge (2^{20}-2^{15})$, and $m_4 \leftarrow z \wedge (2^{25}-2^{20})$, in preparation for the next step.

W3. [Count words.] Set $l \leftarrow 1$ and $A_0 \leftarrow w$; variable l will count the number of words we have found so far, starting with w. Then perform the operations *swap*(4) defined below.

W4. [Print a record-setting solution.] If l exceeds or equals the current maximum, print A_j for $0 \le j < l$. ∎

The heart of this high-speed method is the sequence of operations *swap*(4), which should be expanded inline (for example with a macro-processor) to eliminate all

unnecessary overhead. It is defined in terms of the basic operation

$sw(j)$: Set $w \leftarrow w \oplus m_j$. Then if w is a word, set $A_l \leftarrow w$ and $l \leftarrow l+1$.

Given $sw(j)$, which flips the letters in position j, we define

$$\begin{aligned} swap(0) &= sw(0); \\ swap(1) &= swap(0), sw(1), swap(0); \\ swap(2) &= swap(1), sw(2), swap(1); \\ swap(3) &= swap(2), sw(3), swap(2); \\ swap(4) &= swap(3), sw(4), swap(3). \end{aligned} \tag{22}$$

Thus $swap(4)$ expands into a sequence of 31 steps $sw(0)$, $sw(1)$, $sw(0)$, $sw(2)$, $\ldots$, $sw(0) = sw(\rho(1))$, $sw(\rho(2))$, $\ldots$, $sw(\rho(31))$; these steps will be used 10 million times. We clearly gain speed by embedding the ruler function values $\rho(k)$ directly into our program, instead of recomputing them repeatedly for each word pair via Algorithm M, G, or L.

The winning pair of words generates a set of 21, namely

```
ducks  ducky  duces  dunes  dunks  dinks  dinky
dines  dices  dicey  dicky  dicks  picks  picky
pines  piney  pinky  pinks  punks  punky  pucks
```
(23)

If, for example, $w =$ `ducks` and $w' =$ `piney`, then $m_0 =$ `s` $\oplus$ `y`, so the first operation $sw(0)$ changes `ducks` to `ducky`, which is seen to be a word. The next operation $sw(1)$ applies m_1, which is `k` $\oplus$ `e` in the next-to-last letter position, so it produces the nonword `ducey`. Another application of $sw(0)$ changes `ducey` to `duces` (a legal term generally followed by the word `tecum`). And so on. All word pairs can be processed by this method in at most a few seconds.

Further streamlining is also possible. For example, once we have found a pair that yields k words, we can reject later pairs as soon as they generate $33-k$ nonwords. But the method we've discussed is already quite fast, and it demonstrates the fact that even the loopless Algorithm L can be beaten.

Fans of Algorithm L may, of course, complain that we have speeded up the process only in the small special case $n = 5$, while Algorithm L solves the generation problem for n in general. A similar idea does, however, work also for general values of $n > 5$: We can expand out a program so that it rapidly generates all 32 settings of the rightmost bits $a_4a_3a_2a_1a_0$, as above; then we can apply Algorithm L after every 32 steps, using it to generate successive changes to the other bits $a_{n-1}\ldots a_5$. This approach reduces the amount of work done by Algorithm L by nearly a factor of 32.

Other binary Gray codes. The Gray binary code $g(0)$, $g(1)$, $\ldots$, $g(2^n-1)$ is only one of many ways to traverse all possible n-bit strings while changing only a single bit at each step. Let us say that, in general, a "Gray code" on binary n-tuples is *any* cycle $(v_0, v_1, \ldots, v_{2^n-1})$ that includes every n-tuple and has the property that v_k differs from $v_{(k+1) \bmod 2^n}$ in just one bit position. Thus, in the

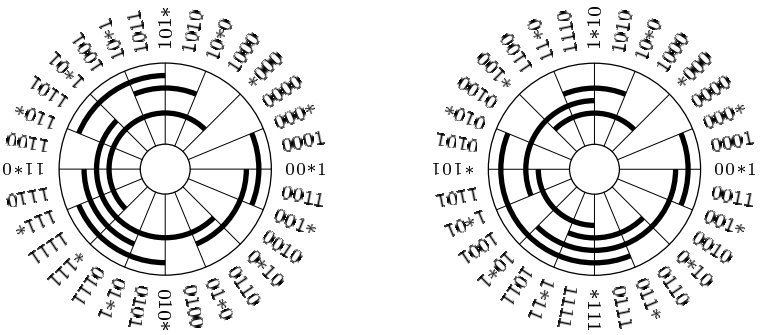


Fig. 13. (a) Complementary Gray code. (b) Balanced Gray code.

terminology of graph theory, a Gray code is an oriented Hamiltonian circuit on the n-cube. We can assume that subscripts have been chosen so that $v_0 = 0 \ldots 0$.

If we think of the v's as binary numbers, there are integers $\delta_0 \ldots \delta_{2^n-1}$ such that

$$v_{(k+1) \bmod 2^n} \;=\; v_k \oplus 2^{\delta_k}, \qquad \text{for } 0 \le k < 2^n; \tag{24}$$

this so-called "delta sequence" is another way to describe a Gray code. For example, the delta sequence for standard Gray binary when $n = 3$ is 01020102; it is essentially the ruler function $\delta_k = \rho(k+1)$ of (13), but the final value δ_{2^n-1} is $n-1$ instead of n, so that the cycle closes. The individual elements δ_k always lie in the range $0 \le \delta_k < n$, and they are called "coordinates."

Let $d(n)$ be the number of different delta sequences that define an n-bit Gray code, and let $c(n)$ be the number of "canonical" delta sequences in which each coordinate k appears before the first appearance of $k+1$. Then $d(n) = n!\, c(n)$, because every permutation of the coordinate numbers in a delta sequence obviously produces another delta sequence. The only possible canonical delta sequences for $n \le 3$ are easily seen to be

$$00; \qquad 0101; \qquad 01020102 \quad \text{and} \quad 01210121. \tag{25}$$

Therefore $c(1) = c(2) = 1$, $c(3) = 2$; $d(1) = 1$, $d(2) = 2$, and $d(3) = 12$. A straightforward computer calculation, using techniques for the enumeration of Hamiltonian circuits that we will study later, establishes the next values,

$$\begin{aligned} c(4) &= 112; & d(4) &= 2688; \\ c(5) &= 15{,}109{,}096; & d(5) &= 1{,}813{,}091{,}520. \end{aligned} \tag{26}$$

No simple pattern is evident, and the numbers grow quite rapidly (see exercise 45); therefore it's a fairly safe bet that nobody will ever know the exact values of $c(8)$ and $d(8)$.

Since the number of possibilities is so huge, people have often attempted to find Gray codes that have additional useful properties. For example, Fig. 13(a) shows a 4-bit Gray code in which every string $a_3a_2a_1a_0$ is diametrically opposite to its complement $\overline{a}_3\overline{a}_2\overline{a}_1\overline{a}_0$. Such codes are possible whenever the number of bits is even (see exercise 49).

An even more interesting Gray code, found by G. C. Tootill [*Proc. IEE* **103**, Part B Supplement (1956), 435], is shown in Fig. 13(b). This one has the same number of changes in each of the four coordinate tracks, hence all coordinates share equally in the activities. Gray codes that are balanced in a similar way can in fact be constructed for all larger values of n, by using the following versatile method to extend a code from n bits to $n+2$ bits:

Theorem D. *Let $\alpha_1 j_1 \alpha_2 j_2 \ldots \alpha_l j_l$ be a delta sequence for an n-bit Gray code, where each j_k is a single coordinate, each α_k is a possibly empty sequence of coordinates, and l is odd. Then*

$$\begin{aligned}&\alpha_1(n{+}1)\alpha_1^R n\alpha_1\\ &\quad j_1\alpha_2 n\alpha_2^R(n{+}1)\alpha_2\ j_2\alpha_3(n{+}1)\alpha_3^R n\alpha_3\ \ldots\ j_{l-1}\alpha_l(n{+}1)\alpha_l^R n\alpha_l\\ &\qquad\qquad (n{+}1)\alpha_l^R j_{l-1}\alpha_{l-1}^R\ldots\alpha_2^R j_1\alpha_1^R n\end{aligned}\tag{27}$$

is the delta sequence of an $(n+2)$-bit Gray code.

For example, if we start with the sequence $01\underline{0}201\underline{02}$ for $n=3$ and let the three underlined elements be j_1, j_2, j_3, the new sequence (27) for a 5-bit code is

$$01410301020131024201043401020103.\tag{28}$$

Proof. Let α_k have length m_k and let v_{kt} be the vertex reached if we start at $0\ldots0$ and apply the coordinate changes $\alpha_1 j_1 \ldots \alpha_{j-1} j_{k-1}$ and the first t of α_k. We need to prove that all vertices $00v_{kt}$, $01v_{kt}$, $10v_{kt}$, and $11v_{kt}$ occur when (27) is used, for $1\le k\le l$ and $0\le t\le m_k$. (The leftmost coordinate is $n{+}1$.)

Starting with $000\ldots0 = 00v_{10}$, we proceed to obtain the vertices

$$00v_{11},\ \ldots,\ 00v_{1m_1},\ 10v_{1m_1},\ \ldots,\ 10v_{10}, 11v_{10},\ \ldots,\ 11v_{1m_1};$$

then j_1 yields $11v_{20}$, which is followed by

$$11v_{21},\ \ldots,\ 11v_{2m_2},\ 10v_{2m_2},\ \ldots,\ 10v_{20}, 00v_{20},\ \ldots,\ 00v_{2m_2};$$

then comes $00v_{30}$, etc., and we eventually reach $11v_{lm_l}$. The glorious finale then uses the third line of (27) to generate all the missing vertices $01v_{lm_l}$, $\ldots$, $01v_{10}$ and take us back to $000\ldots0$. ∎

The *transition counts* $(c_0,\ldots,c_{n-1})$ of a delta sequence are defined by letting c_j be the number of times $\delta_k = j$. For example, (28) has transition counts $(12,8,4,4,4)$, and it arose from a sequence with transition counts $(4,2,2)$. If we choose the original delta sequence carefully and underline appropriate elements j_k, we can obtain transition counts that are as equal as possible:

Corollary B. *For all $n\ge4$, there is an n-bit Gray code with transition counts $(c_0,c_1,\ldots,c_{n-1})$ that satisfy the condition*

$$|c_j - c_k| \le 2 \qquad \text{for } 0\le j<k<n.\tag{29}$$

(This is the best possible balance condition, because each c_j must be an even number, and we must have $c_0+c_1+\cdots+c_{n-1}=2^n$. Indeed, condition (29)

holds if and only if $n-r$ of the counts are equal to $2q$ and r are equal to $2q+2$, where $q=\lfloor 2^{n-1}/n\rfloor$ and $r=2^{n-1} \bmod n$.)

Proof. Given a delta sequence for an n-bit Gray code with transition counts $(c_0,\ldots,c_{n-1})$, the counts for code (27) are obtained by starting with the values $(c'_0,\ldots,c'_{n-1},c'_n,c'_{n+1})=(4c_0,\ldots,4c_{n-1},l{+}1,l{+}1)$, then subtracting 2 from c'_{j_k} for $1\le k<l$ and subtracting 4 from c'_{j_l}. For example, when $n=3$ we can obtain a balanced 5-bit Gray code having transition counts $(8-2,16-10,8,6,6)=(6,6,8,6,6)$ if we apply Theorem D to the delta sequence $\underline{01}2\underline{1}0\underline{12}\underline{1}$. Exercise 51 works out the details for other values of n. ∎

Another important class of n-bit Gray codes in which each of the coordinate tracks has equal responsibility arises when we consider *run lengths*, namely the distances between consecutive appearances of the same δ value. Standard Gray binary code has run length 2 in the least significant position, and this can lead to a loss of accuracy when precise measurements need to be made [see, for example, the discussion by G. M. Lawrence and W. E. McClintock, *Proc. SPIE* **2831** (1996), 104–111]. But all runs have length 4 or more in the remarkable 5-bit Gray code whose delta sequence is

$$(0123042103210423)^2. \tag{30}$$

Let $r(n)$ be the maximum value r such that an n-bit Gray code can be found in which all runs have length $\ge r$. Clearly $r(1)=1$, and $r(2)=r(3)=r(4)=2$; and it is easy to see that $r(n)$ must be less than n when $n>2$, hence (30) proves that $r(5)=4$. Exhaustive computer searches establish the values $r(6)=4$ and $r(7)=5$. Indeed, a fairly straightforward backtrack calculation for the case $n=7$ needs a tree of only about 60 million nodes to determine that $r(7)<6$, and exercise 61(a) constructs a 7-bit code with no run shorter than 5. The exact values of $r(n)$ are unknown for $n\ge 8$; but $r(10)$ is almost certainly 8, and interesting constructions are known by which we can prove that $r(n)=n-O(\log n)$ as $n\to\infty$. (See exercises 60–64.)

***Binary Gray paths.** We have defined an n-bit Gray code as a periodic sequence $v_0, v_1, \ldots$, in which v_k is adjacent to v_{k+1} in the n-cube for all $k\ge 0$, each binary n-tuple occurs in $(v_0,v_1,\ldots,v_{2^n-1})$, and $v_{k+2^n}=v_k$. The periodic property is nice, but not always essential; and sometimes we can do better without it. Therefore we say that an n-bit *Gray path* is any ordering $v_0, v_1, \ldots, v_{2^n-1}$ of the 2^n possible n-tuples in such a way that v_k is adjacent to v_{k+1} for $0\le k<2^n-1$. In other words, a Gray code is a Hamiltonian *circuit* on the vertices of the n-cube, but a Gray path is simply a Hamiltonian *path* on that graph.

The most important binary Gray paths that are not also Gray codes are n-bit paths $v_0, v_1, \ldots, v_{2^n-1}$ that are *monotonic*, in the sense that

$$\nu(v_k)\ \le\ \nu(v_{k+2}) \qquad \text{for } 0\le k<2^n-2. \tag{31}$$

(Here, as elsewhere, we use ν to denote the "weight" or the "sideways sum" of a binary string, namely the number of 1s that it has.) Trial and error shows that

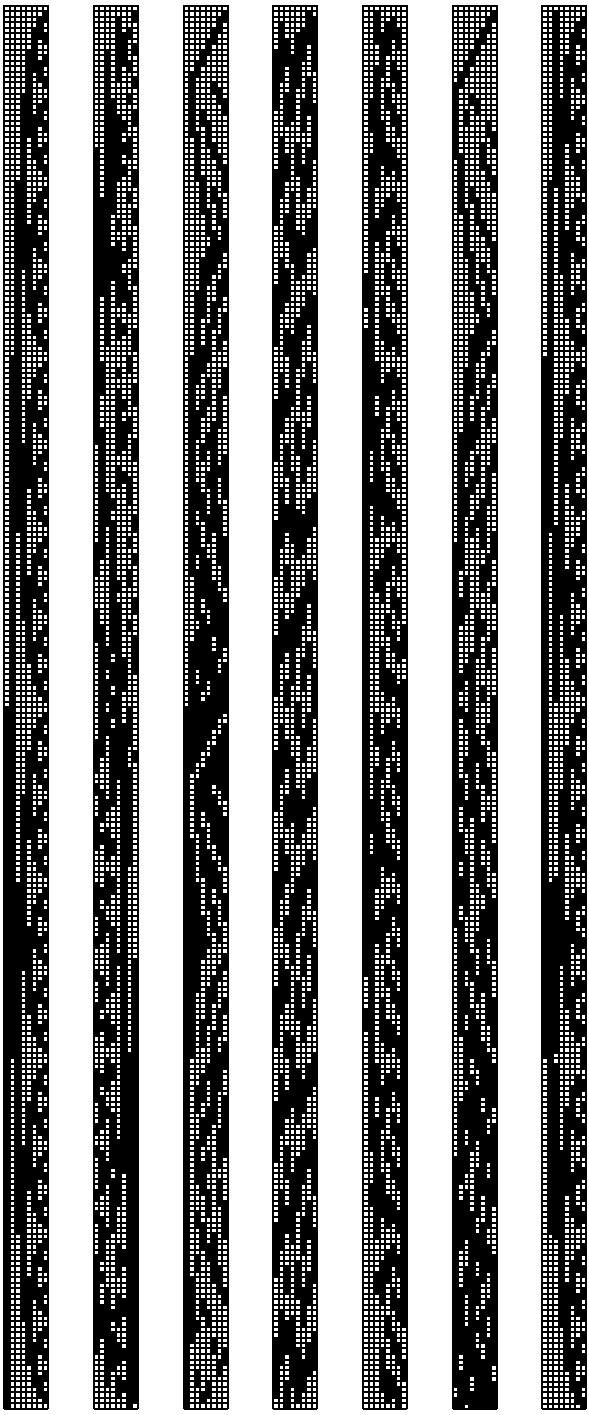


Fig. 14. Examples of 8-bit Gray paths:

a) standard;
b) balanced;
c) complementary;
d) long-run;
e) nonlocal;
f) monotonic;
g) trend-free.

there are essentially only two monotonic n-bit Gray paths for each $n \le 4$, one starting with 0^n and the other starting with $0^{n-1}1$. The two for $n = 3$ are

$$000,\ 001,\ 011,\ 010,\ 110,\ 100,\ 101,\ 111; \tag{32}$$

$$001,\ 000,\ 010,\ 110,\ 100,\ 101,\ 111,\ 011. \tag{33}$$

The two for $n = 4$ are slightly less obvious, but not really difficult to discover.

Since $\nu(v_{k+1}) = \nu(v_k) \pm 1$ whenever v_k is adjacent to v_{k+1}, we obviously can't strengthen (31) to the requirement that all n-tuples be strictly sorted by weight. But relation (31) is strong enough to determine the weight of each v_k, given k and the weight of v_0, because we know that exactly $\binom{n}{j}$ of the n-tuples have weight j.

Fig. 14 summarizes our discussions so far, by illustrating seven of the zillions of Gray codes and Gray paths that make a grand tour through all 256 of the possible 8-bit bytes. Black squares represent ones and white squares represent zeros. Fig. 14(a) is the standard Gray binary code, while Fig. 14(b) is balanced with exactly $256/8 = 32$ transitions in each coordinate position. Fig. 14(c) is a Gray code analogous to Fig. 13(a), in which the bottom 128 codes are complements of the top 128. In Fig. 14(d), the transitions in each coordinate position never occur closer than five steps apart; in other words, all run lengths are at least 5. The code in Fig. 14(e) is *nonlocal* in the sense of exercise 59. Fig. 14(f) shows a monotonic path for $n = 8$; notice how black it gets near the bottom. Finally, Fig. 14(g) illustrates a Gray path that is totally nonmonotonic, in the sense that the center of gravity of the black squares lies exactly at the halfway point in each column. Standard Gray binary code has this property in seven of the coordinate positions, but Fig. 14(g) achieves perfect black-white weight balance in all eight. Such paths are called *trend-free*; they are important in the design of agricultural and other experiments (see exercises 75 and 76).

Carla Savage and Peter Winkler [*J. Combinatorial Theory* **A70** (1995), 230–248] found an elegant way to construct monotonic binary Gray paths for all $n > 0$. Such paths are necessarily built from subpaths P_{nj} in which all transitions are between n-tuples of weights j and $j + 1$. Savage and Winkler defined suitable subpaths recursively by letting $P_{10} = 0, 1$ and, for all $n > 0$,

$$P_{(n+1)j} = 1P^{\pi_n}_{n(j-1)},\ 0P_{nj}; \tag{34}$$

$$P_{nj} = \emptyset \quad \text{if } j < 0 \text{ or } j \ge n. \tag{35}$$

Here π_n is a permutation of the coordinates that we will specify later, and the notation P^π means that every element $a_{n-1} \ldots a_1 a_0$ of the sequence P is replaced by $b_{n-1} \ldots b_1 b_0$, where $b_{j\pi} = a_j$. (We don't define P^π by letting $b_j = a_{j\pi}$, because we want $(2^j)^\pi$ to be $2^{j\pi}$.) It follows, for example, that

$$P_{20} = 0P_{10} = 00,\ 01 \tag{36}$$

because $P_{1(-1)}$ is vacuous; also

$$P_{21} = 1P^{\pi_1}_{10} = 10,\ 11 \tag{37}$$

because P_{11} is vacuous and π_1 must be the identity permutation. In general, P_{nj} is a sequence of n-bit strings containing exactly $\binom{n-1}{j}$ strings of weight j interleaved with $\binom{n-1}{j}$ strings of weight $j+1$.

Let α_{nj} and ω_{nj} be the first and last elements of P_{nj}. Then we easily find

$$\omega_{nj} = 0^{n-j-1}1^{j+1}, \qquad \text{for } 0 \le j < n; \tag{38}$$

$$\alpha_{n0} = 0^n, \qquad \text{for } n > 0; \tag{39}$$

$$\alpha_{nj} = 1\alpha_{(n-1)(j-1)}^{\pi_{n-1}}, \qquad \text{for } 1 \le j < n. \tag{40}$$

In particular, α_{nj} always has weight j, and ω_{nj} always has weight $j+1$. We will define permutations π_n of $\{0, 1, \ldots, n-1\}$ so that both of the sequences

$$P_{n0},\ P_{n1}^R,\ P_{n2},\ P_{n3}^R,\ \ldots \tag{41}$$

$$\text{and } P_{n0}^R,\ P_{n1},\ P_{n2}^R,\ P_{n3},\ \ldots \tag{42}$$

are monotonic binary Gray paths for $n = 1, 2, 3, \ldots$. In fact, the monotonicity is clear, so only the Grayness is in doubt; and the sequences (41), (42) link up nicely because the adjacencies

$$\alpha_{n0} \text{—} \alpha_{n1} \text{—} \cdots \text{—} \alpha_{n(n-1)}, \qquad \omega_{n0} \text{—} \omega_{n1} \text{—} \cdots \text{—} \omega_{n(n-1)} \tag{43}$$

follow immediately from (34), regardless of the permutations π_n. Thus the crucial point is the transition at the comma in formula (34), which makes $P_{(n+1)j}$ a Gray subpath if and only if

$$\omega_{n(j-1)}^{\pi_n} = \alpha_{nj} \qquad \text{for } 0 < j < n. \tag{44}$$

For example, when $n = 2$ and $j = 1$ we need $(01)^{\pi_2} = \alpha_{21} = 10$, by (38)–(40); hence π_2 must transpose coordinates 0 and 1. The general formula (see exercise 71) turns out to be

$$\pi_n = \sigma_n \pi_{n-1}^2, \tag{45}$$

where σ_n is the n-cycle $(n{-}1\ \ldots\ 1\,0)$. The first few cases are therefore

$$\pi_1 = (0), \qquad \pi_4 = (0\,3),$$
$$\pi_2 = (0\,1), \qquad \pi_5 = (0\,4\,3\,2\,1),$$
$$\pi_3 = (0\,2\,1), \qquad \pi_6 = (0\,5\,2\,4\,1\,3);$$

no simple "closed form" for the magic permutations π_n is apparent. Exercise 73 shows that the Savage–Winkler paths can be generated efficiently.

Nonbinary Gray codes. We have studied the case of binary n-tuples in great detail, because it is the simplest, most classical, most applicable, and most thoroughly explored part of the subject. But of course there are numerous applications in which we want to generate $(a_1, \ldots, a_n)$ with coordinates in the more general ranges $0 \le a_j < m_j$, as in Algorithm M. Gray codes and Gray paths apply to this case as well.

Consider, for example, decimal digits, where we want $0 \le a_j < 10$ for each j. Is there a decimal way to count that is analogous to the Gray binary code, changing only one digit at a time? Yes; in fact, *two* natural schemes are

available. In the first, called *reflected Gray decimal*, the sequence for counting up to a thousand with 3-digit strings has the form

$$000, 001, \ldots, 009, 019, 018, \ldots, 011, 010, 020, 021, \ldots, 091, 090, 190, 191, \ldots, 900,$$

with each coordinate moving alternately from 0 up to 9 and then back down from 9 to 0. In the second, called *modular Gray decimal*, the digits always increase by 1 mod 10, therefore they "wrap around" from 9 to 0:

$$000, 001, \ldots, 009, 019, 010, \ldots, 017, 018, 028, 029, \ldots, 099, 090, 190, 191, \ldots, 900.$$

In both cases the digit that changes on step k is determined by the radix-ten ruler function $\rho_{10}(k)$, the largest power of 10 that divides k. Therefore each n-tuple of digits occurs exactly once: We generate 10^j different settings of the rightmost j digits before changing any of the others, for $1 \le j \le n$.

In general, the reflected Gray code in any mixed-radix system can be regarded as a permutation of the nonnegative integers, a function that maps an ordinary mixed-radix number

$$k = \left[\begin{matrix} b_{n-1}, & \ldots, & b_1, & b_0 \\ m_{n-1}, & \ldots, & m_1, & m_0 \end{matrix}\right] = b_{n-1}m_{n-2}\ldots m_1 m_0 + \cdots + b_1 m_0 + b_0 \tag{46}$$

into its reflected-Gray equivalent

$$rg(k) = \left[\begin{matrix} a_{n-1}, & \ldots, & a_1, & a_0 \\ m_{n-1}, & \ldots, & m_1, & m_0 \end{matrix}\right] = a_{n-1}m_{n-2}\ldots m_1 m_0 + \cdots + a_1 m_0 + a_0, \tag{47}$$

just as (7) does this in the special case of binary numbers. Let

$$A_j = \left[\begin{matrix} a_{n-1}, & \ldots, & a_j \\ m_{n-1}, & \ldots, & m_j \end{matrix}\right], \qquad B_j = \left[\begin{matrix} b_{n-1}, & \ldots, & b_j \\ m_{n-1}, & \ldots, & m_j \end{matrix}\right], \tag{48}$$

so that

$$A_j = m_j A_{j+1} + a_j \qquad \text{and} \qquad B_j = m_j B_{j+1} + b_j. \tag{49}$$

The rule connecting the a's and b's is not difficult to derive by induction:

$$a_j = \begin{cases} b_j, & \text{if } B_{j+1} \text{ is even;} \\ m_j - 1 - b_j, & \text{if } B_{j+1} \text{ is odd.} \end{cases} \tag{50}$$

(Here we are numbering the coordinates of the n-tuples $(a_{n-1}, \ldots, a_1, a_0)$ and $(b_{n-1}, \ldots, b_1, b_0)$ from right to left, for consistency with (7) and the conventions of mixed-radix notation in Eq. 4.1–(9). Readers who prefer notations like $(a_1, \ldots, a_n)$ can change j to $n - j$ in all the formulas if they wish.) Going the other way, we have

$$b_j = \begin{cases} a_j, & \text{if } a_{j+1} + a_{j+2} + \cdots \text{ is even;} \\ m_j - 1 - a_j, & \text{if } a_{j+1} + a_{j+2} + \cdots \text{ is odd.} \end{cases} \tag{51}$$

Curiously, rule (50) and its inverse in (51) are exactly the same when all of the radices m_j are odd. In Gray ternary code, for example, when $m_0 = m_1 = \cdots = 3$, we have $rg\big((10010211012)_3\big) = (12210211010)_3$ and also $rg\big((12210211010)_3\big) = (10010211012)_3$. Exercise 78 proves (50) and (51), and discusses similar formulas

that hold in the modular case. People often call these sequences "Gray codes," although strictly speaking they might only be Gray paths (see exercise 79).

We can in fact generate such Gray sequences looplessly, generalizing Algorithms M and L:

Algorithm H (*Loopless reflected mixed-radix Gray generation*). This algorithm visits all n-tuples $(a_{n-1},\ldots,a_0)$ such that $0\le a_j<m_j$ for $0\le j<n$, changing only one coordinate by ±1 at each step. It maintains an array of focus pointers $(f_n,\ldots,f_0)$ to control the actions as in Algorithm L, together with an array of directions $(d_{n-1},\ldots,d_0)$. We assume that each radix m_j is ≥ 2.

H1. [Initialize.] Set $a_j\leftarrow 0$, $f_j\leftarrow j$, and $d_j\leftarrow 1$, for $0\le j<n$; also set $f_n\leftarrow n$.

H2. [Visit.] Visit the n-tuple $(a_{n-1},\ldots,a_1,a_0)$.

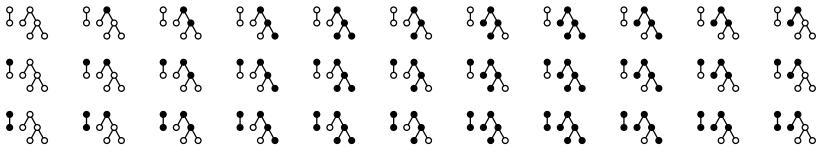
H3. [Choose j.] Set $j\leftarrow f_0$ and $f_0\leftarrow 0$. (As in Algorithm L, j was the rightmost active coordinate; all elements to its right have now been activated.)

H4. [Change coordinate j.] Terminate if $j=n$; otherwise set $a_j\leftarrow a_j+d_j$.

H5. [Reflect?] If $a_j=0$ or $a_j=m_j-1$, set $d_j\leftarrow -d_j$, $f_j\leftarrow f_{j+1}$, and $f_{j+1}\leftarrow j+1$. (Coordinate j has thus become passive.) Return to H2. ∎

A similar algorithm generates the modular variation (see exercise 77).

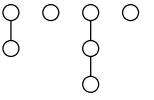
***Subforests.** An interesting and instructive generalization of Algorithm H, discovered by Y. Koda and F. Ruskey [*J. Algorithms* **15** (1993), 324–340], sheds further light on the subject of Gray codes and loopless generation. Suppose we have a forest of n nodes, and we want to visit all of its "principal subforests," namely all subsets of nodes S such that if x is in S and x is not a root, the parent of x is also in S. For example, the 7-node forest has 33 such subsets, corresponding to the black nodes in the following 33 diagrams:



(52)

Notice that if we read the top row from left to right, the middle row from right to left, and the bottom row from left to right, the status of exactly one node changes at each step.

If the given forest consists of degenerate nonbranching trees, the principal subforests are equivalent to mixed-radix numbers. For example, a forest like

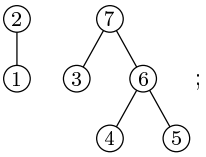


has $3\times2\times4\times2$ principal subforests, corresponding to 4-tuples (x_1,x_2,x_3,x_4) such that $0\le x_1<3$, $0\le x_2<2$, $0\le x_3<4$, and $0\le x_4<2$; the value of x_j is the number of nodes selected in the jth forest. When the algorithm of Koda

and Ruskey is applied to such a forest, it will visit the subforests in the same order as the reflected Gray code on radices $(3, 2, 4, 2)$.

Algorithm K (*Loopless reflected subforest generation*). Given a forest whose nodes are $(1, \ldots, n)$ when arranged in postorder, this algorithm visits all binary n-tuples $(a_1, \ldots, a_n)$ such that $a_p \geq a_q$ whenever p is a parent of q. (Thus, $a_p = 1$ means that p is a node in the current subforest.) Exactly one bit a_j changes between one visit and the next. Focus pointers $(f_0, f_1, \ldots, f_n)$ analogous to those of Algorithm L are used together with additional arrays of pointers $(l_0, l_1, \ldots, l_n)$ and $(r_0, r_1, \ldots, r_n)$, which represent a doubly linked list called the "current fringe." The current fringe contains all nodes of the current subforest and their children; r_0 points to its leftmost node and l_0 to its rightmost.

An auxiliary array $(c_0, c_1, \ldots, c_n)$ defines the forest as follows: If p has no children, $c_p = 0$; otherwise c_p is the leftmost (smallest) child of p. Also c_0 is the leftmost root of the forest itself. When the algorithm begins, we assume that $r_p = q$ and $l_q = p$ whenever p and q are consecutive children of the same family. Thus, for example, the forest in (52) has the postorder numbering



;

therefore we should have $(c_0, \ldots, c_7) = (2, 0, 1, 0, 0, 0, 4, 3)$ and $r_2 = 7$, $l_7 = 2$, $r_3 = 6$, $l_6 = 3$, $r_4 = 5$, and $l_5 = 4$ at the beginning of step K1 in this case.

K1. [Initialize.] Set $a_j \leftarrow 0$ and $f_j \leftarrow j$ for $1 \leq j \leq n$, thereby making the initial subforest empty and all nodes active. Set $f_0 \leftarrow 0$, $l_0 \leftarrow n$, $r_n \leftarrow 0$, $r_0 \leftarrow c_0$, and $l_{c_0} \leftarrow 0$, thereby putting all roots into the current fringe.

K2. [Visit.] Visit the subforest defined by $(a_1, \ldots, a_n)$.

K3. [Choose p.] Set $q \leftarrow l_0$, $p \leftarrow f_q$. (Now p is the rightmost active node of the fringe.) Also set $f_q \leftarrow q$ (thereby activating all nodes to p's right).

K4. [Check a_p.] Terminate the algorithm if $p = 0$. Otherwise go to K6 if $a_p = 1$.

K5. [Insert p's children.] Set $a_p \leftarrow 1$. Then, if $c_p \neq 0$, set $q \leftarrow r_p$, $l_q \leftarrow p - 1$, $r_{p-1} \leftarrow q$, $r_p \leftarrow c_p$, $l_{c_p} \leftarrow p$ (thereby putting p's children to the right of p in the fringe). Go to K7.

K6. [Delete p's children.] Set $a_p \leftarrow 0$. Then, if $c_p \neq 0$, set $q \leftarrow r_{p-1}$, $r_p \leftarrow q$, $l_q \leftarrow p$ (thereby removing p's children from the fringe).

K7. [Make p passive.] (At this point we know that p is active.) Set $f_p \leftarrow f_{l_p}$ and $f_{l_p} \leftarrow l_p$. Return to K2. ∎

The reader is encouraged to play through this algorithm on examples like (52), in order to understand the beautiful mechanism by which the fringe grows and shrinks at just the right times.

***Shift register sequences.** A completely different way to generate all n-tuples of m-ary digits is also possible: We can generate one digit at a time, and repeatedly work with the n most recently generated digits, thus passing from one n-tuple $(x_0, x_1, \ldots, x_{n-1})$ to another one $(x_1, \ldots, x_{n-1}, x_n)$ by shifting an appropriate new digit in at the right. For example, Fig. 15 shows how all 5-bit numbers can be obtained as blocks of 5 consecutive bits in a certain cyclic pattern of length 32. This general idea has already been discussed in some of the exercises of Sections 2.3.4.2 and 3.2.2, and we now are ready to explore it further.

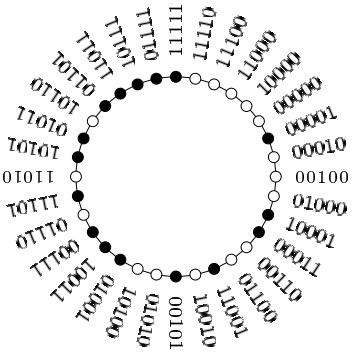


Fig. 15.
A de Bruijn cycle for 5-bit numbers.

Algorithm S (*Generic shift register generation*). This algorithm visits all n-tuples $(a_1, \ldots, a_n)$ such that $0 \le a_j < m$ for $1 \le j \le n$, provided that a suitable function f is used in step S3.

S1. [Initialize.] Set $a_j \leftarrow 0$ for $-n < j \le 0$ and $k \leftarrow 1$.

S2. [Visit.] Visit the n-tuple $(a_{k-n}, \ldots, a_{k-1})$. Terminate if $k = m^n$.

S3. [Advance.] Set $a_k \leftarrow f(a_{k-n}, \ldots, a_{k-1})$, $k \leftarrow k + 1$, and return to S2. ▮

Every function f that makes Algorithm S valid corresponds to a cycle of m^n radix-m digits such that every combination of n digits occurs consecutively in the cycle. For example, the case $m = 2$ and $n = 5$ illustrated in Fig. 15 corresponds to the binary cycle

$$00000100011001010011101011011111; \tag{53}$$

and the first m^2 digits of the infinite sequence

$$00110212203132330414243 44\ldots \tag{54}$$

yield an appropriate cycle for $n = 2$ and arbitrary m. Such cycles are commonly called m-ary *de Bruijn cycles*, because N. G. de Bruijn treated the binary case for arbitrary n in *Indagationes Mathematicæ* **8** (1946), 461–467.

Exercise 2.3.4.2–23 proves that exactly $m!^{m^{n-1}}/m^n$ functions f have the required properties. That's a huge number, but only a few of those functions are known to be efficiently computable. We will discuss three kinds of f that appear to be the most useful.

Table 1
PARAMETERS FOR ALGORITHM A

3 : 1	8 : 1,5	13 : 1,3	18 : 7	23 : 5	28 : 3
4 : 1	9 : 4	14 : 1,11	19 : 1,5	24 : 1,3	29 : 2
5 : 2	10 : 3	15 : 1	20 : 3	25 : 3	30 : 1,15
6 : 1	11 : 2	16 : 2,3	21 : 2	26 : 1,7	31 : 3
7 : 1	12 : 3,4	17 : 3	22 : 1,7	27 : 1,7	32 : 1,27

The entries '$n : s$' or '$n : s,t$' mean that the polynomials $x^n + x^s + 1$ or $x^n + (x^s + 1)(x^t + 1)$ are primitive modulo 2. Additional values up to $n = 168$ have been tabulated by W. Stahnke, *Math. Comp.* **27** (1973), 977–980.

The first important case occurs when m is a prime number, and f is the almost-linear recurrence

$$f(x_1, \ldots, x_n) = \begin{cases} c_1, & \text{if } (x_1, x_2, \ldots, x_n) = (0, 0, \ldots, 0); \\ 0, & \text{if } (x_1, x_2, \ldots, x_n) = (1, 0, \ldots, 0); \\ (c_1 x_1 + c_2 x_2 + \cdots + c_n x_n) \bmod m, & \text{otherwise.} \end{cases} \tag{55}$$

Here the coefficients $(c_1, \ldots, c_n)$ must be such that

$$x^n - c_n x^{n-1} - \cdots - c_2 x - c_1 \tag{56}$$

is a primitive polynomial modulo m, in the sense discussed following Eq. 3.2.2–(9). The number of such polynomials is $\varphi(m^n - 1)/n$, large enough to allow us to find one in which only a few of the c's are nonzero. [This construction goes back to a pioneering paper of Willem Mantel, *Nieuw Archief voor Wiskunde* (2) **1** (1897), 172–184.]

For example, suppose $m = 2$. We can generate binary n-tuples with a very simple loopless procedure:

Algorithm A (*Almost-linear bit-shift generation*). This algorithm visits all n-bit vectors, by using either a special offset s [Case 1] or two special offsets s and t [Case 2], as found in Table 1.

A1. [Initialize.] Set $(x_0, x_1, \ldots, x_{n-1}) \leftarrow (1, 0, \ldots, 0)$ and $k \leftarrow 0$, $j \leftarrow s$. In Case 2, also set $i \leftarrow t$ and $h \leftarrow s + t$.

A2. [Visit.] Visit the n-tuple $(x_{k-1}, \ldots, x_0, x_{n-1}, \ldots, x_{k+1}, x_k)$.

A3. [Test for end.] If $x_k \neq 0$, set $r \leftarrow 0$; otherwise set $r \leftarrow r + 1$, and go to A6 if $r = n - 1$. (We have just seen r consecutive zeros.)

A4. [Shift.] Set $k \leftarrow (k - 1) \bmod n$ and $j \leftarrow (j - 1) \bmod n$. In Case 2 also set $i \leftarrow (i - 1) \bmod n$ and $h \leftarrow (h - 1) \bmod n$.

A5. [Compute a new bit.] Set $x_k \leftarrow x_k \oplus x_j$ [Case 1] or $x_k \leftarrow x_k \oplus x_j \oplus x_i \oplus x_h$ [Case 2]. Return to A2.

A6. [Finish.] Visit $(0, \ldots, 0)$ and terminate. ∎

Appropriate offset parameters s and possibly t almost certainly exist for all n, because primitive polynomials are so abundant; for example, eight different choices of (s, t) would work when $n = 32$, and Table 1 merely lists the smallest.

However, a rigorous proof of existence in all cases lies well beyond the present state of mathematical knowledge.

Our first construction of de Bruijn cycles, in (55), was algebraic, relying for its validity on the theory of finite fields. A similar method that works when m is not a prime number appears in exercise 3.2.2–21. Our next construction, by contrast, will be purely combinatorial. In fact, it is strongly related to the idea of modular Gray m-ary codes.

Algorithm R (*Recursive de Bruijn cycle generation*). Suppose $f()$ is a coroutine that will output the successive digits of an m-ary de Bruijn cycle of length m^n, beginning with n zeros, when it is invoked repeatedly. This algorithm is a similar coroutine that outputs a cycle of length m^{n+1}, provided that $n \geq 2$. It maintains three private variables x, y, and t; variable x should initially be zero.

R1. [Output.] Output x. Go to R3 if $x \neq 0$ and $t \geq n$.

R2. [Invoke f.] Set $y \leftarrow f()$.

R3. [Count ones.] If $y = 1$, set $t \leftarrow t + 1$; otherwise set $t \leftarrow 0$.

R4. [Skip one?] If $t = n$ and $x \neq 0$, go back to R2.

R5. [Adjust x.] Set $x \leftarrow (x + y) \bmod m$ and return to R1. ∎

For example, let $m = 3$ and $n = 2$. If $f()$ produces the infinite 9-cycle

$$001102122\ \ 001102122\ \ 0\ldots, \tag{57}$$

then Algorithm R will produce the following infinite 27-cycle at step R1:

$$\begin{aligned} y &= \ 001021220011110212200102122\ 001\ldots \\ t &= \ 001001000012340010000100100\ 001\ldots \\ x &= 000110102220120020211122121\ 0001\ldots \end{aligned}$$

The proof that Algorithm R works correctly is interesting and instructive (see exercise 93). And the proof of the next algorithm, which *doubles* the window size n, is even more so (see exercise 95).

Algorithm D (*Doubly recursive de Bruijn cycle generation*). Suppose $f()$ and $g()$ are coroutines that each will output the successive digits of m-ary de Bruijn cycles of length m^n when invoked repeatedly, beginning with n zeros. (The two cycles might be identical, but they must be generated by independent coroutines because we will consume their values at different rates of speed.) This algorithm is a similar coroutine that outputs a cycle of length m^{2n}. It maintains six private variables x, y, t, x', y', and t'; variables x and x' should initially be m.

The special parameter r must be set to a constant value such that

$$0 \leq r \leq m \qquad \text{and} \qquad \gcd(m^n - r,\ m^n + r) = 2. \tag{58}$$

The best choice is usually $r = 1$ when m is odd and $r = 2$ when m is even.

D1. [Possibly invoke f.] If $t \neq n$ or $x \geq r$, set $y \leftarrow f()$.

D2. [Count repeats.] If $x \neq y$, set $x \leftarrow y$ and $t \leftarrow 1$. Otherwise set $t \leftarrow t + 1$.

D3. [Output from f.] Output the current value of x.

D4. [Invoke g.] Set $y' \leftarrow g()$.

D5. [Count repeats.] If $x' \neq y'$, set $x' \leftarrow y'$ and $t' \leftarrow 1$. Otherwise set $t' \leftarrow t'+1$.

D6. [Possibly reject g.] If $t' = n$ and $x' < r$ and either $t < n$ or $x' < x$, go to D4. If $t' = n$ and $x' < r$ and $x' = x$, go to D3.

D7. [Output from g.] Output the current value of x'. Return to D3 if $t' = n$ and $x' < r$; otherwise return to D1. ∎

The basic idea of Algorithm D is to output from $f()$ and $g()$ alternately, making special adjustments when either sequence generates n consecutive x's for $x < r$. For example, when $f()$ and $g()$ produce the 9-cycle (57), we take $r = 1$ and get

t in step D2: 12 31211112 12312111 12123121 11121231 21111212 ...
x in step D3: 00001102122 00011021 22000110 21220001 102122000 ...
t' in step D6: 12121111212121111212121111212121111212121111212121 ...
x' in step D7: 0 11021220 11021220 11021220 11021220 11021220 1 ...;

so the 81-cycle produced in steps D3 and **D7** is $0\mathbf{0}0\mathbf{0}1\mathbf{0}1\mathbf{1}0\mathbf{1}\mathbf{2}\ldots2\mathbf{2}\mathbf{2}\mathbf{2}\,0\mathbf{0}0\mathbf{0}1\ldots$.

The case $m = 2$ of Algorithm R was discovered by Abraham Lempel [*IEEE Trans.* **C-19** (1970), 1204–1209]; Algorithm D was not discovered until more than 25 years later [C. J. Mitchell, T. Etzion, and K. G. Paterson, *IEEE Trans.* **IT-42** (1996), 1472–1478]. By using them together, starting with simple coroutines for $n = 2$ based on (54), we can build up an interesting family of cooperating coroutines that will generate a de Bruijn cycle of length m^n for any desired $m \geq 2$ and $n \geq 2$, using only $O(\log n)$ simple computations for each digit of output. (See exercise 96.) Furthermore, in the simplest case $m = 2$, this combination "R&D method" has the property that its kth output can be computed directly, as a function of k, by doing $O(n \log n)$ simple operations on n-bit numbers. Conversely, given any n-bit pattern β, the position of β in the cycle can also be computed in $O(n \log n)$ steps. (See exercises 97–99.) No other family of binary de Bruijn cycles is presently known to have the latter property.

Our third construction of de Bruijn cycles is based on the theory of prime strings, which will be of great importance to us when we study pattern matching in Chapter 9. Suppose $\gamma = \alpha\beta$ is the concatenation of two strings; we say that α is a *prefix* of γ and β is a *suffix*. A prefix or suffix of γ is called *proper* if its length is positive but less than the length of γ. Thus β is a proper suffix of $\alpha\beta$ if and only if $\alpha \neq \epsilon$ and $\beta \neq \epsilon$.

Definition P. *A string is prime if it is nonempty and (lexicographically) less than all of its proper suffixes.*

For example, 01101 is not prime, because it is greater than 01; but 01102 is prime. (We assume that strings are composed of letters, digits, or other symbols from a linearly ordered alphabet. Lexicographic or dictionary order is the normal way to compare strings, so we write $\alpha < \beta$ and say that α is less than β when α is lexicographically less than β. In particular, we always have $\alpha \leq \alpha\beta$, and $\alpha < \alpha\beta$ if and only if $\beta \neq \epsilon$.)

Prime strings have often been called *Lyndon words*, because they were introduced by R. C. Lyndon [*Trans. Amer. Math. Soc.* **77** (1954), 202–215]; Lyndon called them "standard sequences." The simpler term "prime" is justified because of the fundamental factorization theorem in exercise 101. We will, however, continue to pay respect to Lyndon implicitly by often using the letter λ to denote strings that are prime.

Several of the most important properties of prime strings were derived by Chen, Fox, and Lyndon in an important paper on group theory [*Annals of Math.* **68** (1958), 81–95], including the following easy but basic result:

Theorem P. *A nonempty string that is less than all its cyclic shifts is prime.*

(The cyclic shifts of $a_1 \dots a_n$ are $a_2 \dots a_n a_1$, $a_3 \dots a_n a_1 a_2$, $\dots$, $a_n a_1 \dots a_{n-1}$.)

Proof. Suppose $\gamma = \alpha\beta$ is not prime, because $\alpha \neq \epsilon$ and $\gamma \geq \beta \neq \epsilon$; but suppose γ is also less than its cyclic shift $\beta\alpha$. Then the conditions $\beta \leq \gamma < \beta\alpha$ imply that $\gamma = \beta\theta$ for some string $\theta < \alpha$. Therefore, if γ is also less than its cyclic shift $\theta\beta$, we have $\theta < \alpha < \alpha\beta < \theta\beta$. But that is impossible, because α and θ have the same length. ∎

Let $L_m(n)$ be the number of m-ary primes of length n. Every string $a_1 \dots a_n$, together with its cyclic shifts, yields d distinct strings for some divisor d of n, corresponding to exactly one prime of length d. For example, from 010010 we get also 100100 and 001001 by cyclic shifting, and the smallest of the periodic parts $\{010, 100, 001\}$ is the prime 001. Therefore we must have

$$\sum_{d \backslash n} d L_m(d) \;=\; m^n, \qquad \text{for all } m, n \geq 1. \tag{59}$$

This family of equations can be solved for $L_m(n)$ using exercise 4.5.3–28(a), and we obtain

$$L_m(n) \;=\; \frac{1}{n} \sum_{d \backslash n} \mu(d) m^{n/d}. \tag{60}$$

During the 1970s, Harold Fredricksen and James Maiorana discovered a beautifully simple way to generate all of the m-ary primes of length n or less, in increasing order [*Discrete Math.* **23** (1978), 207–210]. Before we are ready to understand their algorithm, we need to consider the *n-extension* of a nonempty string λ, namely the first n characters of the infinite string $\lambda\lambda\lambda\dots$. For example, the 10-extension of 123 is 1231231231. In general if $|\lambda| = k$, its n-extension is $\lambda^{\lfloor n/k \rfloor}\lambda'$, where λ' is the prefix of λ whose length is $n \bmod k$.

Definition Q. *A string is preprime if it is a nonempty prefix of a prime.*

Theorem Q. *A string of length $n > 0$ is preprime if and only if it is the n-extension of a prime string λ of length $k \leq n$. This prime string is uniquely determined.*

Proof. See exercise 105. ∎

Theorem Q states, in essence, that there is a one-to-one correspondence between primes of length $\le n$ and preprimes of length n. The following algorithm generates all of the m-ary ones, in increasing order.

Algorithm F (*Prime and preprime string generation*). This algorithm visits all m-ary n-tuples $(a_1, \ldots, a_n)$ such that the string $a_1 \ldots a_n$ is preprime. It also identifies the index j such that $a_1 \ldots a_n$ is the n-extension of the prime $a_1 \ldots a_j$.

F1. [Initialize.] Set $a_0 \leftarrow -1$, $a_1 \leftarrow \cdots \leftarrow a_n \leftarrow 0$, and $j \leftarrow 1$.

F2. [Visit.] Visit $(a_1, \ldots, a_n)$ with index j.

F3. [Prepare to increase.] Set $j \leftarrow n$. Then if $a_j = m-1$, decrease j until finding $a_j < m-1$.

F4. [Add one.] Terminate if $j = 0$. Otherwise set $a_j \leftarrow a_j + 1$. (Now $a_1 \ldots a_j$ is prime, by exercise 105(a).)

F5. [Make n-extension.] For $k \leftarrow j+1, \ldots, n$ (in this order) set $a_k \leftarrow a_{k-j}$. Return to F2. ∎

For example, Algorithm F visits 32 ternary preprimes when $m = 3$ and $n = 4$:

$$\begin{array}{cccccccc} 0000^{\wedge} & 0011^{\wedge} & 0022^{\wedge} & 0111^{\wedge} & 0122^{\wedge} & 0212^{\wedge} & 1111^{\wedge} & 1212^{\wedge} \\ 0001^{\wedge} & 0012^{\wedge} & 0101^{\wedge} & 0112^{\wedge} & 0202^{\wedge} & 0220^{\wedge} & 1112^{\wedge} & 1221^{\wedge} \\ 0002^{\wedge} & 0020^{\wedge} & 0102^{\wedge} & 0120^{\wedge} & 0210^{\wedge} & 0221^{\wedge} & 1121^{\wedge} & 1222^{\wedge} \\ 0010^{\wedge} & 0021^{\wedge} & 0110^{\wedge} & 0121^{\wedge} & 0211^{\wedge} & 0222^{\wedge} & 1122^{\wedge} & 2222^{\wedge} \end{array} \qquad (61)$$

(The digits preceding '$^{\wedge}$' are the prime strings 0, 0001, 0002, 001, 0011, ..., 2.)

Theorem Q explains why this algorithm is correct, because steps F3 and F4 obviously find the smallest m-ary prime of length $\le n$ that exceeds the previous preprime $a_1 \ldots a_n$. Notice that after a_1 increases from 0 to 1, the algorithm proceeds to visit all the $(m-1)$-ary primes and preprimes, increased by $1 \ldots 1$.

Algorithm F is quite beautiful, but what does it have to do with de Bruijn cycles? Here now comes the punch line: If we output the digits $a_1, \ldots, a_j$ in step F2 whenever j is a divisor of n, the sequence of all such digits forms a de Bruijn cycle! For example, in the case $m = 3$ and $n = 4$, the following 81 digits are output:

$$0\,0001\,0002\,0011\,0012\,0021\,0022\,01\,0102\,0111\,0112$$
$$0121\,0122\,02\,0211\,0212\,0221\,0222\,1\,1112\,1122\,12\,1222\,2. \qquad (62)$$

(We omit the primes 001, 002, 011, ..., 122 of (61) because their length does not divide 4.) The reasons underlying this almost magical property are explored in exercise 108. Notice that the cycle has the correct length, by (59).

There is a sense in which the outputs of this procedure are actually equivalent to the "granddaddy" of all de Bruijn cycle constructions that work for all m and n, namely the construction first published by M. H. Martin in *Bull. Amer. Math. Soc.* **40** (1934), 859–864: Martin's original cycle for $m = 3$ and $n = 4$ was 2222122202211...10000, the twos' complement of (62). In fact, Fredricksen and Maiorana discovered Algorithm F almost by accident while looking for a

simple way to generate Martin's sequence; the explicit connection between their algorithm and preprime strings was not noticed until many years later, when Ruskey, Savage, and Wang carried out a careful analysis of the running time [*J. Algorithms* **13** (1992), 414–430]. The principal results of that analysis appear in exercise 107, namely

i) The average value of $n-j$ in steps F3 and F5 is approximately $m/(m-1)^2$.
ii) The total running time to produce a de Bruijn cycle like (62) is $O(m^n)$.

EXERCISES

1. [*10*] Explain how to generate all n-tuples $(a_1,\ldots,a_n)$ in which $l_j \le a_j \le u_j$, given lower bounds l_j and upper bounds u_j for each coordinate. (Assume that $l_j \le u_j$.)

2. [*15*] What is the 1000000th n-tuple visited by Algorithm M if $n = 10$ and $m_j = j$ for $1 \le j \le n$? *Hint:* $\left[\begin{smallmatrix}0,&0,&1,&2,&3,&0,&2,&7,&1,&0\\1,&2,&3,&4,&5,&6,&7,&8,&9,&10\end{smallmatrix}\right] = 1000000$.

▶ **3.** [*M20*] How many times does Algorithm M perform step M4?

▶ **4.** [*18*] On most computers it is faster to count down to 0 rather than up to m. Revise Algorithm M so that it visits all n-tuples in the opposite order, starting with $(m_1-1,\ldots,m_n-1)$ and finishing with $(0,\ldots,0)$.

▶ **5.** [*20*] Algorithms such as the "fast Fourier transform" (exercise 4.6.4–14) often end with an array of answers in bit-reflected order, having $A[(b_0\ldots b_{n-1})_2]$ in the place where $A[(b_{n-1}\ldots b_0)_2]$ is desired. What is a good way to rearrange the answers into proper order? [*Hint:* Reflect Algorithm M.]

6. [*M17*] Prove (7), the basic formula for Gray binary code.

7. [*20*] Fig. 10(b) shows the Gray binary code for a disk that is divided into 16 sectors. What would be a good Gray-like code to use if the number of sectors were 12 or 60 (for hours or minutes on a clock), or 360 (for degrees in a circle)?

8. [*15*] What's an easy way to run through all n-bit strings of even parity, changing only two bits at each step?

9. [*16*] What move should follow Fig. 11, when solving the Chinese ring puzzle?

▶ **10.** [*M21*] Find a simple formula for the total number of steps A_n or B_n in which a ring is (a) removed or (b) replaced, in the shortest procedure for removing n Chinese rings. For example, $A_3 = 4$ and $B_3 = 1$.

11. [*M22*] (H. J. Purkiss, 1865.) The two smallest rings of the Chinese ring puzzle can actually be taken on or off the bar simultaneously. How many steps does the puzzle require when such accelerated moves are permitted?

▶ **12.** [*23*] The *compositions* of n are the sequences of positive integers that sum to n. For example, the compositions of 4 are 1111, 112, 121, 13, 211, 22, 31, and 4. An integer n has exactly 2^{n-1} compositions, corresponding to all subsets of the points $\{1,\ldots,n-1\}$ that might be used to break the interval $(0\,..\,n)$ into integer-sized subintervals.

a) Design a loopless algorithm to generate all compositions of n, representing each composition as a sequential array of integers $s_1 s_2 \ldots s_j$.
b) Similarly, design a loopless algorithm that represents the compositions implicitly in an array of pointers $q_0 q_1 \ldots q_t$, where the elements of the composition are $(q_0-q_1)(q_1-q_2)\ldots(q_{t-1}-q_t)$ and we have $q_0 = n$, $q_t = 0$. For example, the composition 211 would be represented under this scheme by the pointers $q_0 = 4$, $q_1 = 2$, $q_2 = 1$, $q_3 = 0$, and with $t = 3$.

13. [*21*] Continuing the previous exercise, compute also the multinomial coefficient $C = \binom{n}{s_1,\ldots,s_j}$ for use as the composition $s_1 \ldots s_j$ is being visited.

14. [*20*] Design an algorithm to generate all strings $a_1 \ldots a_j$ such that $0 \le j \le n$ and $0 \le a_i < m_i$ for $1 \le i \le j$, in lexicographic order. For example, if $m_1 = m_2 = n = 2$, your algorithm should successively visit ϵ, 0, 00, 01, 1, 10, 11.

▸ **15.** [*25*] Design a *loopless* algorithm to generate the strings of the previous exercise. All strings of the same length should be visited in lexicographic order as before, but strings of different lengths can be intermixed in any convenient way. For example, 0, 00, 01, ϵ, 10, 11, 1 is an acceptable order when $m_1 = m_2 = n = 2$.

▸ **16.** [*23*] A loopless algorithm obviously cannot generate all binary vectors $(a_1, \ldots, a_n)$ in lexicographic order, because the number of coordinates a_j that need to change between successive visits is not bounded. Show, however, that loopless lexicographic generation does become possible if a *linked* representation is used instead of a sequential one: Suppose there are $2n$ nodes $\{1{:}0, 1{:}1, 2{:}0, 2{:}1, \ldots, n{:}0, n{:}1\}$, each containing a `DIGIT` field and a `LINK` field, where $\texttt{DIGIT}(j{:}b) = b$ for $1 \le j \le n$ and $0 \le b \le 1$. An n-tuple like $(0, 1, 0)$ is represented by

$$\texttt{LINK}(0{:}0) = 1{:}0, \quad \texttt{LINK}(1{:}0) = 2{:}1, \quad \texttt{LINK}(2{:}1) = 3{:}0, \quad \texttt{LINK}(3{:}0) = \Lambda,$$

where 0:0 is a special header node; the other `LINK` fields can have any convenient values.

17. [*20*] A well-known construction called the *Karnaugh map* [M. Karnaugh, *Amer. Inst. Elect. Eng. Trans.* **72**, part I (1953), 593–599] uses Gray binary code in two dimensions to display all 4-bit numbers in a 4×4 torus:

0000	0001	0011	0010
0100	0101	0111	0110
1100	1101	1111	1110
1000	1001	1011	1010

(The entries of a torus "wrap around" at the left and right and also at the top and bottom — just as if they were tiles, replicated infinitely often in a plane.) Show that, similarly, all 6-bit numbers can be arranged in an 8×8 torus so that only one coordinate changes when we move north, south, east, or west from any point.

▸ **18.** [*20*] The *Lee weight* of a vector $u = (u_1, \ldots, u_n)$, where each component satisfies $0 \le u_j < m_j$, is defined to be

$$\nu_L(u) = \sum_{j=1}^{n} \min(u_j, m_j - u_j);$$

and the *Lee distance* between two such vectors u and v is

$$d_L(u, v) = \nu_L(u - v), \qquad \text{where } u - v = \big((u_1 - v_1) \bmod m_1, \ldots, (u_n - v_n) \bmod m_n\big).$$

(This is the minimum number of steps needed to change u to v if we adjust some component u_j by ± 1 (modulo m_j) in each step.)

A quaternary vector has $m_j = 4$ for $1 \le j \le n$, and a binary vector has all $m_j = 2$. Find a simple one-to-one correspondence between quaternary vectors $u = (u_1, \ldots, u_n)$ and binary vectors $u' = (u'_1, \ldots, u'_{2n})$, with the property that $\nu_L(u) = \nu(u')$ and $d_L(u, v) = \nu(u' \oplus v')$.

19. [*21*] (*The octacode.*) Let $g(x) = x^3 + 2x^2 + x - 1$.

a) Use one of the algorithms in this section to evaluate $\sum z_{u_0} z_{u_1} z_{u_2} z_{u_3} z_{u_4} z_{u_5} z_{u_6} z_{u_\infty}$, summed over all 256 polynomials

$$(v_0 + v_1 x + v_2 x^2 + v_3 x^3) g(x) \bmod 4 = u_0 + u_1 x + u_2 x^2 + u_3 x^3 + u_4 x^4 + u_5 x^5 + u_6 x^6$$

for $0 \le v_0, v_1, v_2, v_3 < 4$, where u_∞ is chosen so that $0 \le u_\infty < 4$ and $(u_0 + u_1 + u_2 + u_3 + u_4 + u_5 + u_6 + u_\infty) \bmod 4 = 0$.

b) Construct a set of 256 16-bit numbers that differ from each other in at least six different bit positions. (Such a set, first discovered by Nordstrom and Robinson [*Information and Control* **11** (1967), 613–616], is essentially unique.)

20. [*M36*] The 16-bit codewords in the previous exercise can be used to transmit 8 bits of information, allowing transmission errors to be corrected if any one or two bits are corrupted; furthermore, mistakes will be detected (but not necessarily correctable) if any three bits are received incorrectly. Devise an algorithm that either finds the nearest codeword to a given 16-bit number u' or determines that at least three bits of u' are erroneous. How does your algorithm decode the number $(1100100100001111)_2$? [*Hint:* Use the facts that $x^7 \equiv 1$ (modulo $g(x)$ and 4), and that every quaternary polynomial of degree < 3 is congruent to $x^j + 2x^k$ (modulo $g(x)$ and 4) for some $j, k \in \{0, 1, 2, 3, 4, 5, 6, \infty\}$, where $x^\infty = 0$.]

21. [*M30*] A t-subcube of an n-cube can be represented by a string like $**10**0*$, containing t asterisks and $n - t$ specified bits. If all 2^n binary n-tuples are written in lexicographic order, the elements belonging to such a subcube appear in $2^{t'}$ clusters of consecutive entries, where t' is the number of asterisks that lie to the left of the rightmost specified bit. (In the example given, $n = 8$, $t = 5$, and $t' = 4$.) But if the n-tuples are written in Gray binary order, the number of clusters might be reduced. For example, the $(n-1)$-subcubes $*\ldots*0$ and $*\ldots*1$ occur in only $2^{n-2} + 1$ and 2^{n-2} clusters, respectively, when Gray binary order is used, not in 2^{n-1} of them.

a) Explain how to compute $C(\alpha)$, the number of Gray binary clusters of the subcube defined by a given string α of asterisks, 0s, and 1s. What is $C(**10**0*)$?

b) Prove that $C(\alpha)$ always lies between $2^{t'-1}$ and $2^{t'}$, inclusive.

c) What is the average value of $C(\alpha)$, over all $2^{n-t}\binom{n}{t}$ possible t-subcubes?

▸ **22.** [*22*] A "right subcube" is a subcube such as $0110**$ in which all the asterisks appear after all the specified digits. Any binary trie (Section 6.3) can be regarded as a way to partition a cube into disjoint right subcubes, as in Fig. 16(a). If we interchange the left and right subtries of every right subtrie, proceeding downward from the root, we obtain a *Gray binary trie*, as in Fig. 16(b).

Prove that if the "lieves" of a Gray binary trie are traversed in order, from left to right, consecutive lieves correspond to adjacent subcubes. (Subcubes are adjacent if they contain adjacent vertices. For example, $00**$ is adjacent to $011*$ because the first contains 0010 and the second contains 0110; but $011*$ is not adjacent to $10**$.)

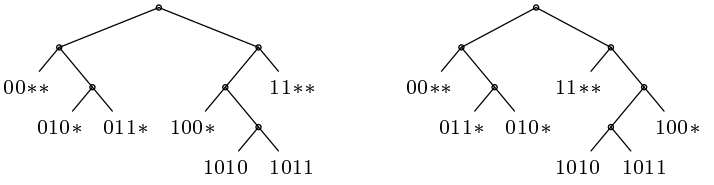


Fig. 16. (a) Normal binary trie. (b) Gray binary trie.

23. [*20*] Suppose $g(k) \oplus 2^j = g(l)$. What is a simple way to find l, given j and k?

24. [*M21*] Consider extending the Gray binary function g to all 2-adic integers (see exercise 4.1–31). What is the corresponding inverse function $g^{[-1]}$?

▸ **25.** [*M25*] Prove that if $g(k)$ and $g(l)$ differ in $t > 0$ bits, and if $0 \le k, l < 2^n$, then $\lceil 2^t/3 \rceil \le |k - l| \le 2^n - \lceil 2^t/3 \rceil$.

26. [*25*] (Frank Ruskey.) For which integers N is it possible to generate all of the nonnegative integers less than N in such a way that only one bit of the binary representation changes at each step?

▸ **27.** [*20*] Let $S_0 = \{1\}$ and $S_{n+1} = 1/(2 + S_n) \cup 1/(2 - S_n)$; thus, for example,

$$S_2 = \left\{ \frac{1}{2 + \frac{1}{2+1}}, \frac{1}{2 + \frac{1}{2-1}}, \frac{1}{2 - \frac{1}{2+1}}, \frac{1}{2 - \frac{1}{2-1}} \right\} = \left\{ \frac{3}{7}, \frac{1}{3}, \frac{3}{5}, 1 \right\},$$

and S_n has 2^n elements that lie between $\frac{1}{3}$ and 1. Compute the 10^{10}th smallest element of S_{100}.

28. [*M26*] A *median* of n-bit strings $\{\alpha_1, \ldots, \alpha_t\}$, where α_k has the binary representation $\alpha_k = a_{k(n-1)} \ldots a_{k0}$, is a string $\hat{\alpha} = a_{n-1} \ldots a_0$ whose bits a_j for $0 \le j < n$ agree with the majority of the bits a_{kj} for $1 \le k \le t$. (If t is even and the bits α_{kj} are half 0 and half 1, the median bit a_j can be either 0 or 1.) For example, the strings $\{0010, 0100, 0101, 1110\}$ have two medians, 0100 and 0110, which we can denote by 01*0.

a) Find a simple way to describe the medians of $G_t = \{g(0), \ldots, g(t-1)\}$, the first t Gray binary strings, when $0 < t \le 2^n$.

b) Prove that if $\alpha = a_{n-1} \ldots a_0$ is such a median, and if $\alpha' = a'_{n-1} \ldots a'_0$ is any element of G_t with $a'_j \ne a_j$, then the string β obtained from α by changing a_j to a'_j is also an element of G_t.

29. [*M24*] If integer values k are transmitted as n-bit Gray binary codes $g(k)$ and received with errors described by a bit pattern $p = (p_{n-1} \ldots p_0)_2$, the average numerical error is

$$\frac{1}{2^n} \sum_{k=0}^{2^n-1} \Big| \big(g^{[-1]}(k) \oplus p\big) - k \Big|,$$

assuming that all values of k are equally likely. Show that this sum is equal to $\sum_{k=0}^{2^n-1} |(k \oplus p) - k|/2^n$, just as if Gray binary code were not used, and evaluate it explicitly.

▸ **30.** [*M27*] (*Gray permutation.*) Design a one-pass algorithm to replace the array elements $(X_0, X_1, X_2, \ldots, X_{2^n-1})$ by $(X_{g(0)}, X_{g(1)}, X_{g(2)}, \ldots, X_{g(2^n-1)})$, using only a constant amount of auxiliary storage. *Hint:* Considering the function $g(n)$ as a permutation of all nonnegative integers, show that the set

$$L = \{0, 1, (10)_2, (100)_2, (100*)_2, (100*0)_2, (100*0*)_2, \ldots\}$$

is the set of *cycle leaders* (the smallest elements of the cycles).

31. [*HM35*] (*Gray fields.*) Let $f_n(x) = g\big(r_n(x)\big)$ denote the operation of reflecting the bits of an n-bit binary string as in exercise 5 and then converting to Gray binary code. For example, the operation $f_3(x)$ takes $(001)_2 \mapsto (110)_2 \mapsto (010)_2 \mapsto (011)_2 \mapsto (101)_2 \mapsto (111)_2 \mapsto (100)_2 \mapsto (001)_2$, hence all of the nonzero possibilities appear in

a single cycle. Therefore we can use f_3 to define a field of 8 elements, with $\oplus$ as the addition operator and with multiplication defined by the rule

$$f_3^{[j]}(1) \times f_3^{[k]}(1) = f_3^{[j+k]}(1) = f_3^{[j]}\big(f_3^{[k]}(1)\big).$$

The functions f_2, f_5, and f_6 have the same nice property. But f_4 does not, because $f_4\big((1011)_2\big) = (1011)_2$.

Find all $n \le 100$ for which f_n defines a field of 2^n elements.

32. [*M20*] True or false: Walsh functions satisfy $w_k(-x) = (-1)^k w_k(x)$.

▶ **33.** [*M20*] Prove the Rademacher-to-Walsh law (17).

34. [*M21*] The *Paley functions* $p_k(x)$ are defined by

$$p_0(x) = 1 \qquad \text{and} \qquad p_k(x) = (-1)^{\lfloor 2x \rfloor k} p_{\lfloor k/2 \rfloor}(2x).$$

Show that $p_k(x)$ has a simple expression in terms of Rademacher functions, analogous to (17), and relate Paley functions to Walsh functions.

35. [*HM23*] The $2^n \times 2^n$ Paley matrix P_n is obtained from Paley functions just as the Walsh matrix W_n is obtained from Walsh functions. (See (20).) Find interesting relations between P_n, W_n, and the Hadamard matrix H_n. Prove that all three matrices are symmetric.

36. [*21*] Spell out the details of an efficient algorithm to compute the Walsh transform $(x_0, \ldots, x_{2^n-1})$ of a given vector $(X_0, \ldots, X_{2^n-1})$.

37. [*HM23*] Let z_{kl} be the location of the lth sign change in $w_k(x)$, for $1 \le l \le k$ and $0 < z_{kl} < 1$. Prove that $|z_{kl} - l/(k+1)| = O\big((\log k)/k\big)$.

▶ **38.** [*M25*] Devise a ternary generalization of Walsh functions.

▶ **39.** [*HM30*] (J. J. Sylvester.) The rows of $\left(\begin{smallmatrix} a & b \\ b & -a \end{smallmatrix}\right)$ are orthogonal to each other and have the same magnitude; therefore the matrix identity

$$\begin{aligned}(\,A\;B\,)\begin{pmatrix} a^2+b^2 & 0 \\ 0 & a^2+b^2 \end{pmatrix}\begin{pmatrix} A \\ B \end{pmatrix} &= (\,A\;B\,)\begin{pmatrix} a & b \\ b & -a \end{pmatrix}\begin{pmatrix} a & b \\ b & -a \end{pmatrix}\begin{pmatrix} A \\ B \end{pmatrix} \\ &= (\,Aa+Bb\;\; Ab-Ba\,)\begin{pmatrix} aA+bB \\ bA-aB \end{pmatrix}\end{aligned}$$

implies the sum-of-two-squares identity $(a^2+b^2)(A^2+B^2) = (aA+bB)^2 + (bA-aB)^2$. Similarly, the matrix

$$\begin{pmatrix} a & b & c & d \\ b & -a & d & -c \\ d & c & -b & -a \\ c & -d & -a & b \end{pmatrix}$$

leads to the sum-of-four-squares identity

$$\begin{aligned}(a^2+b^2+c^2+d^2)(A^2+B^2+C^2+D^2) = {}& (aA+bB+cC+dD)^2 + (bA-aB+dC-cD)^2 \\ & + (dA+cB-bC-aD)^2 + (cA-dB-aC+bD)^2.\end{aligned}$$

a) Attach the signs of the matrix H_3 in (21) to the symbols $\{a, b, c, d, e, f, g, h\}$, obtaining a matrix with orthogonal rows and a sum-of-eight-squares identity.

b) Generalize to H_4 and higher-order matrices.

▶ **40.** [*21*] Would the text's five-letter word computation scheme produce correct answers also if the masks in step W2 were computed as $m_j = x \wedge (2^{5^j} - 1)$ for $0 \le j < 5$?

41. [*25*] If we restrict the five-letter word problem to the most common 3000 words — thereby eliminating `ducky`, `duces`, `dunks`, `dinks`, `dinky`, `dices`, `dicey`, `dicky`, `dicks`, `picky`, `pinky`, `punky`, and `pucks` from (23) — how many valid words can still be generated from a single pair?

42. [*35*] (M. L. Fredman.) Algorithm L uses $\Theta(n \log n)$ bits of auxiliary memory for focus pointers as it decides what Gray binary bit a_j should be complemented next. On each step L3 it examines $\Theta(\log n)$ of the auxiliary bits, and it occasionally changes $\Omega(\log n)$ of them.

Show that, from a theoretical standpoint, we can do better: The n-bit Gray binary code can be generated by changing at most 2 auxiliary bits between visits. (We still allow ourselves to examine $O(\log n)$ of the auxiliary bits on each step, so that we know which of them should be changed.)

43. [*47*] Determine $d(6)$, the number of 6-bit Gray codes.

44. [*M35*] How many of the delta sequences for n-bit Gray codes have the property that exactly (a) one or (b) two of the coordinate names occur only twice? Express your answers in terms of $d(n-1)$ and $d(n-2)$.

45. [*M25*] Show that the sequence $d(n)$ has doubly exponential growth: There is a constant $A > 1$ such that $d(n) = \Omega(A^{2^n})$.

46. [*HM48*] Determine the asymptotic behavior of $d(n)^{1/2^n}$ as $n \to \infty$.

47. [*M46*] (Silverman, Vickers, and Sampson.) Let $S_k = \{g(0), \ldots, g(k-1)\}$ be the first k elements of the standard Gray binary code, and let $H(k, v)$ be the number of Hamiltonian paths in S_k that begin with 0 and end with v. Prove or disprove: $H(k, v) \le H(k, g(k-1))$ for all $v \in S_k$ that are adjacent to $g(k)$.

48. [*36*] Prove that $d(n) \le 4(n/2)^{2^n}$ if the conjecture in the previous exercise is true. [*Hint:* Let $d(n, k)$ be the number of n-bit Gray codes that begin with $g(0) \ldots g(k-1)$; the conjecture implies that $d(n) \le c_{n1} \ldots c_{n(k-1)} d(n, k)$, where c_{nk} is the number of vertices adjacent to $g(k-1)$ in the n-cube but not in S_k.]

49. [*20*] Prove that for all $n \ge 1$ there is a $2n$-bit Gray code in which $v_{k+2^{2n-1}}$ is the complement of v_k, for all $k \ge 0$.

▸ **50.** [*21*] Find a construction like that of Theorem D but with l even.

51. [*M24*] Complete the proof of Corollary B to Theorem D.

52. [*M20*] Prove that if the transition counts of an n-bit Gray code satisfy $c_0 \le c_1 \le \cdots \le c_{n-1}$, we must have $c_0 + \cdots + c_{j-1} \ge 2^j$, with equality when $j = n$.

53. [*M46*] If the numbers $(c_0, \ldots, c_{n-1})$ are even and satisfy the condition of the previous exercise, is there always an n-bit Gray code with these transition counts?

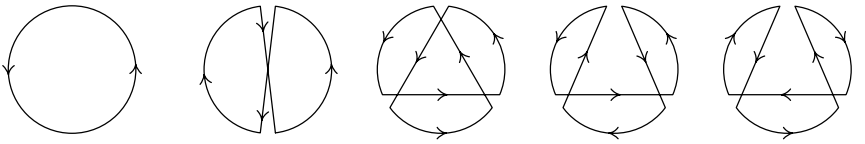
54. [*M20*] (H. S. Shapiro, 1953.) Show that if a sequence of integers $(a_1, \ldots, a_{2^n})$ contains only n distinct values, then there is a subsequence whose product $a_{k+1} a_{k+2} \ldots a_l$ is a perfect square, for some $0 \le k < l \le 2^n$. However, this conclusion might not be true if we disallow the case $l = 2^n$.

55. [*47*] (F. Ruskey and C. Savage, 1993.) If $(v_0, \ldots, v_{2^n-1})$ is an n-bit Gray code, the pairs $\{\, \{v_{2k}, v_{2k+1}\} \mid 0 \le k < 2^{n-1} \,\}$ form a perfect matching between the vertices of even and odd parity in the n-cube. Conversely, does every such perfect matching arise as "half" of some n-bit Gray code?

56. [*M30*] (E. N. Gilbert, 1958.) Say that two Gray codes are equivalent if their delta sequences can be made equal by permuting the coordinate names, or by reversing the

cycle and/or starting the cycle at a different place. Show that the 2688 different 4-bit Gray codes fall into just 9 equivalence classes.

57. [*32*] Consider a graph whose vertices are the 2688 possible 4-bit Gray codes, where two such Gray codes are adjacent if they are related by one of the following simple transformations:



Before After Type 1 After Type 2 After Type 3 After Type 4

(Type 1 changes arise when the cycle can be broken into two parts and reassembled with one part reversed. Types 2, 3, and 4 arise when the cycle can be broken into three parts and reassembled after reversing 0, 1, or 2 of the parts. The parts need not have equal size. Such transformations of Hamiltonian circuits are often possible.)

Write a program to discover which 4-bit Gray codes are transformable into each other, by finding the connected components of the graph; restrict consideration to only one of the four types at a time.

▸ **58.** [*21*] Let α be the delta sequence of an n-bit Gray code, and obtain β from α by changing q occurrences of 0 to n, where q is odd. Prove that $\beta\beta$ is the delta sequence of an $(n+1)$-bit Gray code.

59. [*22*] The 5-bit Gray code of (30) is *nonlocal* in the sense that no 2^t consecutive elements belong to a single t-subcube, for $1 < t < n$. Prove that nonlocal n-bit Gray codes exist for all $n \geq 5$. [*Hint:* See the previous exercise.]

60. [*20*] Show that the run-length-bound function satisfies $r(n+1) \geq r(n)$.

61. [*M30*] Show that $r(m+n) \geq r(m) + r(n) - 1$ if (a) $m = 2$ and $2 < r(n) < 8$; or (b) $m \leq n$ and $r(n) \leq 2^{m-3}$.

62. [*46*] Does $r(8) = 6$?

63. [*30*] (Luis Goddyn.) Prove that $r(10) \geq 8$.

▸ **64.** [*HM35*] (L. Goddyn and P. Gvozdjak.) An n-bit *Gray stream* is a sequence of permutations $(\sigma_0, \sigma_1, \ldots, \sigma_{l-1})$ where each σ_k is a permutation of the vertices of the n-cube, taking every vertex to one of its neighbors.

a) Suppose $(u_0, \ldots, u_{2^m-1})$ is an m-bit Gray code and $(\sigma_0, \sigma_1, \ldots, \sigma_{2^m-1})$ is an n-bit Gray stream. Let $v_0 = 0 \ldots 0$ and $v_{k+1} = v_k\sigma_k$, where $\sigma_k = \sigma_{k \bmod 2^m}$ if $k \geq 2^m$. Under what conditions is the sequence

$$W = (u_0v_0,\ u_0v_1,\ u_1v_1,\ u_1v_2,\ \ldots,\ u_{2^{m+n-1}-1}v_{2^{m+n-1}-1},\ u_{2^{m+n-1}-1}v_{2^{m+n-1}})$$

an $(m+n)$-bit Gray code?

b) Show that if m is sufficiently large, there is an n-bit Gray stream satisfying the conditions of (a) for which all run lengths of the sequence $(v_0, v_1, \ldots)$ are $\geq n-2$.

c) Apply these results to prove that $r(n) \geq n - O(\log n)$.

65. [*30*] (Brett Stevens.) In Samuel Beckett's play *Quad*, the stage begins and ends empty; n actors enter and exit one at a time, running through all 2^n possible subsets, and the actor who leaves is always the one whose previous entrance was earliest. When $n = 4$, as in the actual play, some subsets are necessarily repeated. Show, however, that there is a perfect pattern with exactly 2^n entrances and exits when $n = 5$.

66. [*40*] Is there a perfect Beckett–Gray pattern for 8 actors?

67. [*20*] Sometimes it is desirable to run through all n-bit binary strings by changing as *many* bits as possible from one step to the next, for example when testing a physical circuit for reliable behavior in worst-case conditions. Explain how to traverse all binary n-tuples in such a way that each step changes n or $n-1$ bits, alternately.

68. [*21*] Peter Q. Perverse decided to construct an *anti-Gray* ternary code, in which each n-trit number differs from its neighbors in *every* digit position. Is such a code possible for all n?

▸ **69.** [*M25*] Modify the definition of Gray binary code (7) by letting

$$h(k) = \big(\dots(b_6 \oplus b_5)(b_5 \oplus b_4)(b_4 \oplus b_3 \oplus b_2 \oplus b_0)(b_3 \oplus b_0)(b_2 \oplus b_1 \oplus b_0)b_1\big)_2,$$

when $k = (\dots b_5 b_4 b_3 b_2 b_1 b_0)_2$.

a) Show that the sequence $h(0)$, $h(1)$, $\dots$, $h(2^n-1)$ runs through all n-bit numbers in such a way that exactly 3 bits change each time, when $n > 3$.

b) Generalize this rule to obtain sequences in which exactly t bits change at each step, when t is odd and $n > t$.

70. [*21*] How many monotonic n-bit Gray paths exist for $n = 5$ and $n = 6$?

71. [*M22*] Derive (45), the recurrence that defines the Savage–Winkler permutations.

72. [*20*] What is the Savage–Winkler path from 00000 to 11111?

▸ **73.** [*32*] Design an efficient algorithm to construct the delta sequence of an n-bit monotonic Gray path.

74. [*M25*] (Savage and Winkler.) How far apart can adjacent vertices of the n-cube be, in a monotonic Gray path?

75. [*32*] Find all 5-bit Gray paths v_0, $\dots$, v_{31} that are *trend-free*, in the sense that $\sum_{k=0}^{31} k(-1)^{v_{kj}} = 0$ in each coordinate position j.

76. [*M25*] Prove that trend-free n-bit Gray paths exist for all $n \geq 5$.

77. [*21*] Modify Algorithm H in order to visit mixed-radix n-tuples in *modular* Gray order.

78. [*M26*] Prove the conversion formulas (50) and (51) for reflected mixed-radix Gray paths, and derive analogous formulas for the modular case.

▸ **79.** [*M22*] When is the last n-tuple of the (a) reflected (b) modular mixed-radix Gray path adjacent to the first?

80. [*M20*] Explain how to run through all divisors of a number, given its prime factorization $p_1^{e_1} \dots p_t^{e_t}$, repeatedly multiplying or dividing by a single prime at each step.

81. [*M21*] Let (a_0, b_0), (a_1, b_1), $\dots$, (a_{m^2-1}, b_{m^2-1}) be the 2-digit m-ary modular Gray code. Show that, if $m > 2$, every edge (x, y) — $\big(x, (y+1) \bmod m\big)$ and (x, y) — $\big((x+1) \bmod m, y\big)$ occurs in one of the two cycles

$$(a_0, b_0) — (a_1, b_1) — \cdots — (a_{m^2-1}, b_{m^2-1}) — (a_0, b_0),$$
$$(b_0, a_0) — (b_1, a_1) — \cdots — (b_{m^2-1}, a_{m^2-1}) — (b_0, a_0).$$

▸ **82.** [*M25*] (G. Ringel, 1956.) Use the previous exercise to deduce that there exist four 8-bit Gray codes that, together, cover all edges of the 8-cube.

83. [*M46*] Can four *balanced* 8-bit Gray codes cover all edges of the 8-cube?

▸ **84.** [*25*] (Howard L. Dyckman.) Fig. 17 shows a fascinating puzzle called Loony Loop or the Gordian Knot, in which the object is to remove a flexible cord from the rigid loops that surround it. Show that the solution to this puzzle is inherently related to the reflected Gray ternary path.

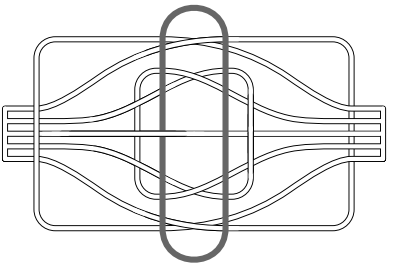


Fig. 17. The Loony Loop puzzle.

▸ **85.** [*M25*] (Dana Richards.) If $\Gamma = (\alpha_0, \ldots, \alpha_{t-1})$ is a sequence of t strings of length n and $\Gamma' = (\alpha'_0, \ldots, \alpha'_{t'-1})$ is a sequence of t' strings of length n', the *boustrophedon product* $\Gamma \boxtimes \Gamma'$ is the sequence of tt' strings of length $n+n'$ that begins

$$(\alpha_0\alpha'_0, \ldots, \alpha_0\alpha'_{t'-1}, \alpha_1\alpha'_{t'-1}, \ldots, \alpha_1\alpha'_0, \alpha_2\alpha'_0, \ldots, \alpha_2\alpha'_{t'-1}, \alpha_3\alpha'_{t'-1}, \ldots)$$

and ends with $\alpha_{t-1}\alpha'_0$ if t is even, $\alpha_{t-1}\alpha'_{t'-1}$ if t is odd. For example, the basic definition of Gray binary code in (5) can be expressed in this notation as $\Gamma_n = (0,1) \boxtimes \Gamma_{n-1}$ when $n > 0$. Prove that the operation $\boxtimes$ is associative, hence $\Gamma_{m+n} = \Gamma_m \boxtimes \Gamma_n$.

▸ **86.** [*26*] Define an infinite Gray path that runs through all possible nonnegative integer n-tuples $(a_1, \ldots, a_n)$ in such a way that $\max(a_1, \ldots, a_n) \le \max(a'_1, \ldots, a'_n)$ when $(a_1, \ldots, a_n)$ is followed by $(a'_1, \ldots, a'_n)$.

87. [*27*] Continuing the previous exercise, define an infinite Gray path that runs through *all* integer n-tuples $(a_1, \ldots, a_n)$, in such a way that $\max(|a_1|, \ldots, |a_n|) \le \max(|a'_1|, \ldots, |a'_n|)$ when $(a_1, \ldots, a_n)$ is followed by $(a'_1, \ldots, a'_n)$.

▸ **88.** [*25*] After Algorithm K has terminated in step K4, what would happen if we immediately restarted it in step K2?

▸ **89.** [*25*] (*Gray code for Morse code.*) The Morse code words of length n (exercise 4.5.3–32) are strings of dots and dashes, where n is the number of dots plus twice the number of dashes.

a) Show that it is possible to generate all Morse code words of length n by successively changing a dash to two dots or vice versa. For example, the path for $n = 3$ must be •—, •••, —• or its reverse.

b) What string follows •— —•• — — •— • in your sequence for $n = 15$?

90. [*26*] For what values of n can the Morse code words be arranged in a *cycle*, under the ground rules of exercise 89? [*Hint:* The number of code words is F_{n+1}.]

▸ **91.** [*34*] Design a loopless algorithm to visit all binary n-tuples $(a_1, \ldots, a_n)$ such that $a_1 \le a_2 \ge a_3 \le a_4 \ge \cdots$. [The number of such n-tuples is F_{n+2}.]

92. [*M30*] Is there an infinite sequence Φ_n whose first m^n elements form an m-ary de Bruijn cycle, for all m? [The case $n = 2$ is solved in (54).]

▸ **93.** [*M28*] Prove that Algorithm R outputs a de Bruijn cycle as advertised.

94. [*22*] What is the output of Algorithm D when $m = 5$, $n = 1$, $r = 3$, and both $f()$ and $g()$ are the trivial cycles 01234 01234 01...?

▸ **95.** [*M23*] Suppose an infinite sequence $a_0a_1a_2\ldots$ of period p is interleaved with an infinite sequence $b_0b_1b_2\ldots$ of period q to form the infinite cyclic sequence

$$c_0\,c_1c_2\,c_3c_4\,c_5\ldots = a_0\,b_0\,a_1b_1a_2\,b_2\ldots.$$

a) Under what circumstances does $c_0\,c_1c_2\ldots$ have period pq? (The "period" of a sequence $a_0a_1a_2\ldots$, for the purposes of this exercise, is the smallest integer $p>0$ such that $a_k=a_{k+p}$ for all $k\ge 0$.)

b) Which $2n$-tuples would occur as consecutive outputs of Algorithm D if step D6 were changed to say simply "If $t'=n$ and $x'<r$, go to D4"?

c) Prove that Algorithm D outputs a de Bruijn cycle as advertised.

▸ **96.** [*M23*] Suppose a family of coroutines has been set up to generate a de Bruijn cycle of length m^n using Algorithms R and D, based recursively on simple coroutines for the base case $n=2$.

a) How many coroutines of each type will there be?

b) What is the maximum number of coroutine activations needed to get one top-level digit of output?

97. [*M29*] The purpose of this exercise is to analyze the de Bruijn cycles constructed by Algorithms R and D in the important special case $m=2$. Let $f_n(k)$ be the $(k+1)$st bit of the 2^n-cycle, so that $f_n(k)=0$ for $0\le k<n$. Also let j_n be the index such that $0\le j_n<2^n$ and $f_n(k)=1$ for $j_n\le k<j_n+n$.

a) Write out the cycles $\big(f_n(0)\ldots f_n(2^n-1)\big)$ for $n=2$, 3, 4, and 5.

b) Prove that, for all even values of n, there is a number $\delta_n=\pm1$ such that we have

$$f_{n+1}(k)\equiv\begin{cases}\Sigma f_n(k), & \text{if } 0<k\le j_n \text{ or } 2^n+j_n<k\le 2^{n+1},\\ 1+\Sigma f_n(k+\delta_n), & \text{if } j_n<k\le 2^n+j_n,\end{cases}$$

where the congruence is modulo 2. (In this formula Σf stands for the summation function $\Sigma f(k)=\sum_{j=0}^{k-1}f(j)$.) Hence $j_{n+1}=2^n-\delta_n$ when n is even.

c) Let $\big(c_n(0)\,c_n(1)\ldots c_n(2^{2n}-5)\big)$ be the cycle produced when the simplified version of Algorithm D in exercise 95(b) is applied to $f_n()$. Where do the $(2n-1)$-tuples 1^{2n-1} and $(01)^{n-1}0$ occur in this cycle?

d) Use the results of (c) to express $f_{2n}(k)$ in terms of $f_n()$.

e) Find a (somewhat) simple formula for j_n as a function of n.

98. [*M34*] Continuing the previous exercise, design an efficient algorithm to compute $f_n(k)$, given $n\ge 2$ and $k\ge 0$.

▸ **99.** [*M23*] Exploit the technology of the previous exercises to design an efficient algorithm that locates any given n-bit string in the cycle $\big(f_n(0)f_n(1)\ldots f_n(2^n-1)\big)$.

100. [*40*] Do the de Bruijn cycles of exercise 97 provide a useful source of pseudo-random bits when n is large?

▸ **101.** [*M30*] (*Unique factorization of strings into nonincreasing primes.*)

a) Prove that if λ and λ' are prime, then $\lambda\lambda'$ is prime if $\lambda<\lambda'$.

b) Consequently every string α can be written in the form

$$\alpha=\lambda_1\lambda_2\ldots\lambda_t,\qquad \lambda_1\ge\lambda_2\ge\cdots\ge\lambda_t,\qquad \text{where each } \lambda_j \text{ is prime.}$$

c) In fact, only one such factorization is possible. *Hint:* Show that λ_t must be the lexicographically smallest nonempty suffix of α.

d) True or false: λ_1 is the longest prime prefix of α.

e) What are the prime factors of 31415926535897932384626433832795028841971?

102. [*HM28*] Deduce the number of m-ary primes of length n from the unique factorization theorem in the previous exercise.

103. [*M20*] Use Eq. (59) to prove Fermat's theorem that $m^p \equiv m$ (modulo p).

104. [*17*] According to formula (6o), about $1/n$ of all n-letter words are prime. How many of the 5757 five-letter GraphBase words are prime? Which of them is the smallest nonprime? The largest prime?

105. [*M31*] Let α be a preprime of length n.

a) Show that if the final letter of α is increased, the resulting string is prime.

b) If α has been factored as in exercise 101, show that it is the n-extension of λ_1.

c) Furthermore α cannot be the n-extension of two different primes.

▸ **106.** [*M30*] By reverse-engineering Algorithm F, design an algorithm that visits all m-ary primes and preprimes in *decreasing* order.

107. [*HM30*] Analyze the running time of Algorithm F.

108. [*M35*] Let $\lambda_1 < \cdots < \lambda_t$ be the m-ary prime strings whose lengths divide n, and let $a_1 \ldots a_n$ be any m-ary string. The object of this exercise is to prove that $a_1 \ldots a_n$ appears in $\lambda_1 \ldots \lambda_t \lambda_1 \lambda_2$; hence $\lambda_1 \ldots \lambda_t$ is a de Bruijn cycle (since it has length m^n). For convenience we may assume that $m = 10$ and that strings correspond to decimal numbers; the same arguments will apply for arbitrary $m \geq 2$.

a) Show that if $a_1 \ldots a_n = \alpha\beta$ is distinct from all its cyclic shifts, and if $\beta\alpha = \lambda_k$ is prime, then $\alpha\beta$ is a substring of $\lambda_k\lambda_{k+1}$, unless $\alpha = 9^j$ for some $j \geq 1$.

b) Where does $\alpha\beta$ appear in $\lambda_1 \ldots \lambda_t$ if $\beta\alpha$ is prime and α consists of all 9s? *Hint:* Show that if $a_{n+1-j} \ldots a_n = 9^l$ in step F2 for some $l > 0$, and if j is not a divisor of n, the previous step F2 had $a_{n-l} \ldots a_n = 9^{l+1}$.

c) Now consider n-tuples of the form $(\alpha\beta)^d$, where $d > 1$ is a divisor of n and $\beta\alpha = \lambda_k$ is prime.

d) Identify the positions of 899135, 997879, 913131, 090909, 909090, and 911911 when $n = 6$.

109. [*M22*] An m-ary de Bruijn torus of size $m^2 \times m^2$ for 2×2 windows is a matrix of m-ary digits a_{ij} such that each of the m^4 submatrices

$$\begin{pmatrix} a_{ij} & a_{i(j+1)} \\ a_{(i+1)j} & a_{(i+1)(j+1)} \end{pmatrix}, \qquad 0 \leq i, j < m^2$$

is different, where subscripts wrap around modulo m^2. Thus every possible m-ary 2×2 submatrix occurs exactly once; Ian Stewart [*Game, Set, and Math* (Oxford: Blackwell, 1989), Chapter 4] has therefore called it an m-ary *ourotorus*. For example,

$$\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

is a binary ourotorus; indeed, it is essentially the only such matrix when $m = 2$, except for shifting and/or transposition.

Consider the infinite matrix A whose entry in row $i = (\ldots a_2 a_1 a_0)_2$ and column $j = (\ldots b_2 b_1 b_0)_2$ is $a_{ij} = (\ldots c_2 c_1 c_0)_2$, where

$$c_0 = (a_0 \oplus b_0)(a_1 \oplus b_1) \oplus b_1;$$
$$c_k = (a_{2k} a_0 \oplus b_{2k})b_0 \oplus (a_{2k+1} a_0 \oplus b_{2k+1})(b_0 \oplus 1), \quad \text{for } k > 0.$$

Show that the upper left $2^{2n} \times 2^{2n}$ submatrix of A is a 2^n-ary ourotorus for all $n \geq 0$.

110. [*M25*] Continuing the previous exercise, construct m-ary ourotoruses for all m.

111. [*20*] We can obtain the number 100 in twelve ways by inserting + and − signs into the sequence 123456789; for example, $100 = 1 + 23 - 4 + 5 + 6 + 78 - 9 = 123 - 45 - 67 + 89 = -1 + 2 - 3 + 4 + 5 + 6 + 78 + 9$.

a) What is the smallest positive integer that cannot be represented in such a way?

b) Consider also inserting signs into the 10-digit sequence 9876543210.

▸ **112.** [*25*] Continuing the previous exercise, how far can we go by inserting signs into 123456789876543 21? For example, $100 = -1234 - 5 - 6 + 7898 - 7 - 6543 - 2 - 1$.

ANSWERS TO EXERCISES

All that heard him were astonished
at his understanding and answers.
— Luke 2:47

SECTION 7.2.1.1

1. Let $m_j = u_j - l_j + 1$, and visit $(a_1 + l_1, \ldots, a_n + l_n)$ instead of visiting $(a_1, \ldots, a_n)$ in Algorithm M. Or, change '$a_j \leftarrow 0$' to '$a_j \leftarrow l_j$' and '$a_j = m_j - 1$' to '$a_j = u_j$' in that algorithm, and set $l_0 \leftarrow 0$, $u_0 \leftarrow 1$ in step M1.

2. $(0, 0, 1, 2, 3, 0, 2, 7, 0, 9)$.

3. Step M4 is performed $m_1 m_2 \ldots m_k$ times when $j = k$; therefore the total is $\sum_{k=0}^{n} \prod_{j=1}^{k} m_j = m_1 \ldots m_n (1 + 1/m_n + 1/m_n m_{n-1} + \cdots + 1/m_n \ldots m_1)$. If all m_j are 2 or more, this is less than $2m_1 \ldots m_n$. [Thus, we should keep in mind that fancy Gray-code methods, which change only one digit per visit, actually reduce the total number of digit changes by at most a factor of 2.]

4. **N1.** [Initialize.] Set $a_j \leftarrow m_j - 1$ for $0 \le j \le n$, where $m_0 = 2$.

N2. [Visit.] Visit the n-tuple $(a_1, \ldots, a_n)$.

N3. [Prepare to subtract one.] Set $j \leftarrow n$.

N4. [Borrow if necessary.] If $a_j = 0$, set $a_j \leftarrow m_j - 1$, $j \leftarrow j - 1$, and repeat this step.

N5. [Decrease, unless done.] If $j = 0$, terminate the algorithm. Otherwise set $a_j \leftarrow a_j - 1$ and go back to step N2. ∎

5. Bit reflection is easy on a machine like `MMIX`, but on other computers we can proceed as follows:

R1. [Initialize.] Set $j \leftarrow k \leftarrow 0$.

R2. [Swap.] Interchange $A[j + 1] \leftrightarrow A[k + 2^{n-1}]$. Also, if $j > k$, interchange $A[j] \leftrightarrow A[k]$ and $A[j + 2^{n-1} + 1] \leftrightarrow A[k + 2^{n-1} + 1]$.

R3. [Advance k.] Set $k \leftarrow k + 2$, and terminate if $k \ge 2^{n-1}$.

R4. [Advance j.] Set $h \leftarrow 2^{n-2}$. If $j \ge h$, repeatedly set $j \leftarrow j - h$ and $h \leftarrow h/2$ until $j < h$. Then set $j \leftarrow j + h$. (Now $j = (b_0 \ldots b_{n-1})_2$ if $k = (b_{n-1} \ldots b_0)_2$.) Return to R2. ∎

6. If $g\big((0b_{n-1} \ldots b_1 b_0)_2\big) = \big(0(b_{n-1}) \ldots (b_2 \oplus b_1)(b_1 \oplus b_0)\big)_2$ then $g\big((1b_{n-1} \ldots b_1 b_0)_2\big) = 2^n + g\big((0\overline{b}_{n-1} \ldots \overline{b}_1 \overline{b}_0)_2\big) = \big(1(\overline{b}_{n-1}) \ldots (\overline{b}_2 \oplus \overline{b}_1)(\overline{b}_1 \oplus \overline{b}_0)\big)_2$, where $\overline{b} = b \oplus 1$.

7. To accommodate $2r$ sectors one can use $g(k)$ for $2^n - r \le k < 2^n + r$, where $n = \lceil \lg r \rceil$, because $g(2^n - r) \oplus g(2^n + r - 1) = 2^n$ by (5). [G. C. Tootill, *Proc. IEE* **103**, Part B Supplement (1956), 434.] See also exercise 26.

8. Use Algorithm G with $n \leftarrow n-1$ and include the parity bit a_∞ at the right. (This yields $g(0)$, $g(2)$, $g(4)$, ...)

9. Replace the rightmost ring, since $\nu(1011000)$ is odd.

10. $A_n + B_n = g^{[-1]}(2^n - 1) = \lfloor 2^{n+1}/3 \rfloor$ and $A_n = B_n + n$. Hence $A_n = \lfloor 2^n/3 + n/2 \rfloor$ and $B_n = \lfloor 2^n/3 - n/2 \rfloor$.

Historical notes: The early Japanese mathematician Yoriyuki Arima (1714–1783) treated this problem in his *Shūki Sanpō* (1769), Problem 44, observing that the n-ring puzzle reduces to an $(n-1)$-ring puzzle after a certain number of steps. Let $C_n = A_n - A_{n-1} = B_n - B_{n-1} + 1$ be the number of rings removed during this reduction. Arima noticed that $C_n = 2C_{n-1} - [n \text{ even}]$; thus he could compute $A_n = C_1 + C_2 + \cdots + C_n$ for $n = 9$ without actually knowing the formula $C_n = \lceil 2^{n-1}/3 \rceil$.

More than two centuries earlier, Cardano had already mentioned the "complicati annuli" in his *De Subtilitate Libri XXI* (Nuremberg: 1550), Book 15. He wrote that they are "useless yet admirably subtle," stating erroneously that 95 moves are needed to remove seven rings and 95 more to put them back. John Wallis devoted seven pages to this puzzle in the Latin edition of his *Algebra* **2** (Oxford: 1693), Chapter 111, presenting detailed but nonoptimum methods for the nine-ring case. He included the operation of sliding a ring through the bar as well as putting it on or off, and he hinted that shortcuts were available, but he did not attempt to find a shortest solution.

11. The solution to $S_n = S_{n-2} + 1 + S_{n-2} + S_{n-1}$ when $S_1 = S_2 = 1$ is $S_n = 2^{n-1} - [n \text{ even}]$. [*Math. Quest. Educational Times* **3** (1865), 66–67.]

12. (a) The theory of $n-1$ Chinese rings proves that Gray binary code yields the compositions in a convenient order (4, 31, 211, 22, 112, 1111, 121, 13):

A1. [Initialize.] Set $t \leftarrow 0$, $j \leftarrow 1$, $s_1 \leftarrow n$. (We assume that $n > 1$.)

A2. [Visit.] Visit $s_1 \ldots s_j$. Then set $t \leftarrow 1 - t$, and go to A4 if $t = 0$.

A3. [Odd step.] If $s_j > 1$, set $s_j \leftarrow s_j - 1$, $s_{j+1} \leftarrow 1$, $j \leftarrow j + 1$; otherwise set $j \leftarrow j - 1$ and $s_j \leftarrow s_j + 1$. Return to A2.

A4. [Even step.] If $s_{j-1} > 1$, set $s_{j-1} \leftarrow s_{j-1} - 1$, $s_{j+1} \leftarrow s_j$, $s_j \leftarrow 1$, $j \leftarrow j + 1$; otherwise set $j \leftarrow j - 1$, $s_j \leftarrow s_{j+1}$, $s_{j-1} \leftarrow s_{j-1} + 1$ (but terminate if $j - 1 = 0$). Return to A2. ∎

(b) Now $q_1, \ldots, q_{t-1}$ represent rings on the bar:

B1. [Initialize.] Set $t \leftarrow 1$, $q_0 \leftarrow n$. (We assume that $n > 1$.)

B2. [Visit.] Set $q_t \leftarrow 0$ and visit $(q_0 - q_1) \ldots (q_{t-1} - q_t)$. Go to B4 if t is even.

B3. [Odd step.] If $q_{t-1} = 1$, set $t \leftarrow t - 1$; otherwise set $q_t \leftarrow 1$ and $t \leftarrow t + 1$. Return to step B2.

B4. [Even step.] If $q_{t-2} = q_{t-1} + 1$, set $q_{t-2} \leftarrow q_{t-1}$ and $t \leftarrow t - 1$ (but terminate if $t = 2$); otherwise set $q_t \leftarrow q_{t-1}$, $q_{t-1} \leftarrow q_t + 1$, $t \leftarrow t + 1$. Return to B2. ∎

These algorithms [see J. Misra, *ACM Trans. Math. Software* **1** (1975), 285] are loopless even in their initialization steps.

13. In step A1, also set $C \leftarrow 1$. In step A3, set $C \leftarrow s_j C$ if $s_j > 1$, otherwise $C \leftarrow C/(s_{j-1}+1)$. In step A4, set $C \leftarrow s_{j-1} C$ if $s_{j-1} > 1$, otherwise $C \leftarrow C/(s_{j-2}+1)$.

Similar modifications apply to steps B1, B3, B4. Sufficient precision is needed to accommodate the value $C = n!$ for the composition $1 \ldots 1$; we are stretching the definition of looplessness by assuming that arithmetic operations take unit time.

14. **S1.** [Initialize.] Set $j \leftarrow 0$.

S2. [Visit.] Visit the string $a_1 \ldots a_j$.

S3. [Lengthen.] If $j < n$, set $j \leftarrow j + 1$, $a_j \leftarrow 0$, and return to S2.

S4. [Increase.] If $a_j < m_j - 1$, set $a_j \leftarrow a_j + 1$ and return to S2.

S5. [Shorten.] Set $j \leftarrow j - 1$, and return to S4 if $j > 0$. ∎

15. **T1.** [Initialize.] Set $j \leftarrow 0$.

T2. [Even visit.] If j is even, visit the string $a_1 \ldots a_j$.

T3. [Lengthen.] If $j < n$, set $j \leftarrow j + 1$, $a_j \leftarrow 0$, and return to T2.

T4. [Odd visit.] If j is odd, visit the string $a_1 \ldots a_j$.

T5. [Increase.] If $a_j < m_j - 1$, set $a_j \leftarrow a_j + 1$ and return to T2.

T6. [Shorten.] Set $j \leftarrow j - 1$, and return to T4 if $j > 0$. ∎

This algorithm is loopless, although it may appear at first glance to contain loops, because four steps separate consecutive visits. The basic idea is related to exercise 2.3.1–5 and to "prepostorder" traversal (exercise 7.2.1.3–00).

16. Suppose $\texttt{LINK}(j-1{:}b) = j{:}(b \oplus a_j)$ for $1 \le j \le n$, and $\texttt{LINK}(n{:}b) = \Lambda$. These links represent $(b_1, \ldots, b_n)$ if and only if $g(b_1 \ldots b_n) = a_1 \ldots a_n$, so we can use a loopless Gray binary generator to achieve the desired result.

17. Put the concatenation of 3-bit codes $\big(g(j), g(k)\big)$ in row j and column k, for $0 \le j, k < 8$. [It is not difficult to prove that this is essentially the *only* solution, except for permuting and/or complementing coordinates and/or rotating rows, because the coordinate that changes when moving north or south depends only on the row, and a similar statement applies to columns. Karnaugh's isomorphism between the 4-cube and the 4×4 torus can be traced back to *The Design of Switching Circuits* by W. Keister, A. E. Ritchie, and S. H. Washburn (1951), page 174. Incidentally, Keister went on to design an ingenious variant of Chinese rings called SpinOut, and a generalization called The Hexadecimal Puzzle, *U.S. Patents 3637215–3637216* (1972).]

18. Use 2-bit Gray code to represent the digits $u_j = (0, 1, 2, 3)$ respectively as the bit pairs $u'_{2j-1}u'_{2j} = (00, 01, 11, 10)$. [C. Y. Lee introduced his metric in *IEEE Trans.* **IT-4** (1958), 77–82. A similar $m/2$-bit encoding works for even values of m; for example, when $m = 8$ we can represent $(0, 1, 2, 3, 4, 5, 6, 7)$ by $(0000, 0001, 0011, 0111, 1111, 1110, 1100, 1000)$. But such a scheme leaves out some of the binary patterns when $m > 4$.]

19. (a) A modular Gray quaternary algorithm needs slightly less computation than Algorithm M, but it doesn't matter because 256 is so small. The result is $z_0^8 + z_1^8 + z_2^8 + z_3^8 + 14(z_0^4 z_2^4 + z_1^4 z_3^4) + 56 z_0 z_1 z_2 z_3 (z_0^2 + z_2^2)(z_1^2 + z_3^2)$.

(b) Replacing (z_0, z_1, z_2, z_3) by $(1, z, z^2, z)$ gives $1 + 112z^6 + 30z^8 + 112z^{10} + z^{16}$; thus all of the nonzero Lee weights are ≥ 6. Now use the construction in the previous exercise to convert each $(u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_\infty)$ into a 16-bit number.

20. Recover the quaternary vector $(u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_\infty)$ from u', and use Algorithm 4.6.1D to find the remainder of $u_0 + u_1 x + \cdots + u_6 x^6$ divided by $g(x)$, mod 4; that algorithm can be used in spite of the fact that the coefficients do not belong to a field, because $g(x)$ is monic. Express the remainder as $x^j + 2x^k$ (modulo $g(x)$ and 4), and let $d = (k - j) \bmod 4$, $s = (u_0 + \cdots + u_6 + u_\infty) \bmod 4$.

Case 1, $s = 1$: If $k = \infty$, the error was x^j (in other words, the correct vector has $u_j \leftarrow (u_j - 1) \bmod 4$); otherwise there were three or more errors.

Case 2, $s = 3$: If $j = k$ the error was x^j; otherwise ≥ 3 errors occurred.

Case 3, $s = 0$: If $j = k = \infty$, no errors were made; if $j = \infty$ and $k < \infty$, at least four errors were made. Otherwise the errors were $x^a - x^b$, where $a = \big(j + (\infty, 3, 6, 1, 5, 4, 2, 0)\big) \bmod 7$ according as $d = (0, 1, 2, 3, 4, 5, 6, \infty)$, and $b = (j+2d) \bmod 7$.

Case 4, $s = 2$: If $j = \infty$ the errors were $2x^k$. Otherwise the errors were

$$\begin{aligned} x^j + x^\infty&, \text{ if } k = \infty; \\ -x^j - x^\infty&, \text{ if } d = 0; \\ x^a + x^b&, \text{ if } d \in \{1, 2, 4\},\ a = (j - 3d) \bmod 7,\ b = (j - 2d) \bmod 7; \\ -x^a - x^b&, \text{ if } d \in \{3, 5, 6\},\ a = (j - 3d) \bmod 7,\ b = (j - d) \bmod 7. \end{aligned}$$

Given $u' = (1100100100001111)_2$, we have $u = (2, 0, 3, 1, 0, 0, 2, 2)$ and $2 + 3x^2 + x^3 + 2x^6 \equiv 1 + 3x + 3x^2 \equiv x^5 + 2x^6$; also $s = 2$. Thus the errors are $x^2 + x^3$, and the nearest errorfree codeword is $(2, 0, 2, 0, 0, 0, 2, 2)$. Algorithm 4.6.1D tells us that $2 + 2x^2 + 2x^6 \equiv (2 + 2x + 2x^3)g(x)$ (modulo 4); so the eight information bits are $(11110011)_2$, corresponding to $(v_0, v_1, v_2, v_3) = (2, 2, 0, 2)$. [A more intelligent algorithm would also say, "Aha: The first 16 bits of π."]

For generalizations to other efficient coding schemes based on quaternary vectors, see the classic paper by Hammons, Kumar, Calderbank, Sloane, and Solé, *IEEE Trans.* **IT-40** (1994), 301–319.

21. (a) $C(\epsilon) = 1$, $C(0\alpha) = C(1\alpha) = C(\alpha)$, and $C(*\alpha) = 2C(\alpha) - [10\ldots0 \in \alpha]$. Iterating this recurrence gives $C(\alpha) = 2^t - 2^{t-1}e_t - 2^{t-2}e_{t-1} - \cdots - 2^0 e_1$, where $e_j = [10\ldots0 \in \alpha_j]$ and α_j is the suffix of α following the jth asterisk. In the example we have $\alpha_1 = *10{*}{*}0*$, $\alpha_2 = 10{*}{*}0*$, ..., $\alpha_5 = \epsilon$; thus $e_1 = 0$, $e_2 = 1$, $e_3 = 1$, $e_4 = 0$, and $e_5 = 1$ (by convention), hence $C({*}{*}10{*}{*}0*) = 2^5 - 2^4 - 2^2 - 2^1 = 10$.

(b) We may remove trailing asterisks so that $t = t'$. Then $e_t = 1$ implies $e_{t-1} = \cdots = e_1 = 0$. [The case $C(\alpha) = 2^{t'-1}$ occurs if and only if α ends in 10^j*^k.]

(c) To compute the sum of $C(\alpha)$ over all t-subcubes, note that $\binom{n}{t}$ clusters begin at the n-tuple $0\ldots0$, and $\binom{n-1}{t}$ begin at each succeeding n-tuple (namely one cluster for each t-subcube containing that n-tuple and specifying the bit that changed). Thus the average is $\big(\binom{n}{t} + (2^n - 1)\binom{n-1}{t}\big)/2^{n-t}\binom{n}{t} = 2^t(1 - t/n) + 2^{t-n}(t/n)$. [The formula in (c) holds for *any* n-bit Gray path, but (a) and (b) are specific to the reflected Gray binary code. These results are due to C. Faloutsos, *IEEE Trans.* **SE-14** (1988), 1381–1393.]

22. Let $\alpha*^j$ and $\beta*^k$ be consecutive lieves of a Gray binary trie, where α and β are binary strings and $j \leq k$. Then the last $k - j$ bits of α are a string α' such that α and $\beta\alpha'$ are consecutive elements of Gray binary code, hence adjacent. [Interesting applications of this property to cube-connected message-passing concurrent computers are discussed in *A VLSI Architecture for Concurrent Data Structures* by William J. Dally (Kluwer, 1987), Chapter 3.]

23. $2^j = g(k) \oplus g(l) = g(k \oplus l)$ implies that $l = k \oplus g^{[-1]}(2^j) = k \oplus (2^{j+1} - 1)$. In other words, if $k = (b_{n-1}\ldots b_0)_2$ we have $l = (b_{n-1}\ldots b_{j+1}\overline{b}_j\ldots\overline{b}_0)_2$.

24. Defining $g(k) = k \oplus \lfloor k/2 \rfloor$ as usual, we find $g(k) = g(-1-k)$; hence there are *two* 2-adic integers k such that $g(k)$ has a given 2-adic value l. One of them is even, the other is odd. We can conveniently define $g^{[-1]}$ to be the solution that is even; then (8) is replaced by $b_j = a_{j-1} \oplus \cdots \oplus a_0$, for $j \geq 0$. For example, $g^{[-1]}(1) = -2$ by this definition; when l is a normal integer, the "sign" of $g^{[-1]}(l)$ is the parity of l.

25. Let $p = k \oplus l$; exercise 7.1–00 tells us that $2^{\lfloor \lg p \rfloor + 1} - p \le |k - l| \le p$. We have $\nu(g(p)) = \nu(g(k) \oplus g(l)) = t$ if and only if there are positive integers $j_1, \ldots, j_t$ such that $p = (1^{j_1}0^{j_2}1^{j_3}\ldots(0 \text{ or } 1)^{j_t})_2$. The largest possible $p < 2^n$ occurs when $j_1 = n + 1 - t$ and $j_2 = \cdots = j_t = 1$, yielding $p = 2^n - \lceil 2^t/3 \rceil$. The smallest possible $2^{\lfloor \lg p \rfloor + 1} - p = (1^{j_2}0^{j_3}\ldots(1 \text{ or } 0)^{j_t})_2 + 1$ occurs when $j_2 = \cdots = j_t = 1$, yielding $p = \lceil 2^t/3 \rceil$. [C. K. Yuen, *IEEE Trans.* **IT-20** (1974), 668; S. R. Cavior, *IEEE Trans.* **IT-21** (1975), 596.]

26. Let $N = 2^{n_t} + \cdots + 2^{n_1}$ where $n_t > \cdots > n_1 \ge 0$; also, let Γ_n be any Gray code for $\{0, 1, \ldots, 2^n - 1\}$ that begins at 0 and ends at 1, except that Γ_0 is simply 0. Use

$$\Gamma^R_{n_t},\ 2^{n_t}+\Gamma_{n_{t-1}},\ \ldots,\ 2^{n_t}+\cdots+2^{n_3}+\Gamma^R_{n_2},\ 2^{n_t}+\cdots+2^{n_2}+\Gamma_{n_1},\ \text{if } t \text{ is even;}$$

$$\Gamma_{n_t},\ 2^{n_t}+\Gamma^R_{n_{t-1}},\ \ldots,\ 2^{n_t}+\cdots+2^{n_3}+\Gamma^R_{n_2},\ 2^{n_t}+\cdots+2^{n_2}+\Gamma_{n_1},\ \text{if } t \text{ is odd.}$$

27. In general, if $k = (b_{n-1}\ldots b_0)_2$, the $(k+1)$st largest element of S_n is equal to

$$1/\big(2 - (-1)^{a_{n-1}}/\big(2 - \cdots/\big(2 - (-1)^{a_1}/\big(2 - (-1)^{a_0}\big)\big)\ldots\big)\big),$$

corresponding to the sign pattern $g(k) = (a_{n-1}\ldots a_0)_2$. Thus we can compute any element of S_n in $O(n)$ steps, given its rank. Setting $k = 2^{100} - 10^{10}$ and $n = 100$ yields the answer 373065177/1113604409. [Whenever $f(x)$ is a positive and monotonic function, the 2^n elements $f(\pm f(\ldots \pm f(\pm x)\ldots))$ are ordered according to Gray binary code, as observed by H. E. Salzer, *CACM* **16** (1973), 180. In this particular case there is, however, another way to get the answer, because we also have $S_n = //2, \pm 2, \ldots, \pm 2, \pm 1//$ using the notation of Section 4.5.3; continued fractions in this form are ordered by complementing alternate bits of k.]

28. (a) As $t = 1, 2, \ldots$, bit a_j of median(G_t) runs through the periodic sequence

$$0, \ldots, 0, *, 1, \ldots, 1, *, 0, \ldots, 0, *, \ldots$$

with asterisks at every 2^{1+j}th step. Thus the strings that correspond to the binary representations of $\lfloor (t-1)/2 \rfloor$ and $\lfloor t/2 \rfloor$ are medians. And those strings are in fact "extreme" cases, in the sense that all medians agree with the common bits of $\lfloor (t-1)/2 \rfloor$ and $\lfloor t/2 \rfloor$, hence asterisks appear where they disagree. For example, when $t = 100 = (01100100)_2$ and $n = 8$, we have median$(G_{100}) = 001100{*}{*}$.

(b) If α is $g(p)$ and β is $g(q)$ in Gray binary, we have $p = (p_{n-1}\ldots p_0)_2$ and $q = (p_{n-1}\ldots p_{j+1}\overline{p}_j\ldots\overline{p}_0)_2$, where $p_{j+1} \oplus p_j = a'_j \ne a_j$. By (a), either t or $t-1$ lies in $(a_{n-1}\ldots a_0{*})_2$. Now if $p_j = 1$, clearly $q < p < t$. And if $p_j = 0$ we have $p_{j+1} = \overline{a}_j$; hence $p < t$ implies $q < t$. [See A. J. Bernstein, K. Steiglitz, and J. E. Hopcroft, *IEEE Trans.* **IT-12** (1966), 425–430.]

29. Assuming that $p \ne 0$, let $l = \lfloor \lg p \rfloor$ and $S_a = \{s \mid 2^l a \le s < 2^l(a+1)\}$ for $0 \le a < 2^{n-l}$. Then $(k \oplus p) - k$ has a constant sign for all $k \in S_a$, and

$$\sum_{k \in S_a} \Big|(k \oplus p) - k\Big| = 2^l \|S_a\| = 2^{2l}.$$

Also $g^{[-1]}(g(k) \oplus p) = k \oplus g^{[-1]}(p)$, and $\lfloor \lg g^{[-1]}(p) \rfloor = \lfloor \lg p \rfloor$. Therefore

$$\frac{1}{2^n}\sum_{k=0}^{2^n-1}\Big|g^{[-1]}(g(k)\oplus p) - k\Big| = \frac{1}{2^n}\sum_{a=0}^{2^{n-l}-1}\sum_{k\in S_a}\Big|(k \oplus g^{[-1]}(p)) - k\Big| = \frac{1}{2^n}\sum_{a=0}^{2^{n-l}-1} 2^{2l} = 2^l.$$

[See Morgan M. Buchner, Jr., *Bell System Tech. J.* **48** (1969), 3113–3130.]

30. The cycle containing $k > 1$ has length $2^{\lfloor \lg \lg k \rfloor + 1}$, because it is easy to show from Eq. (7) that if $k = (b_{n-1} \dots b_0)_2$ we have

$$g^{[2^l]}(k) = (c_{n-1} \dots c_0)_2, \qquad \text{where } c_j = b_j \oplus b_{j+l+1}.$$

To permute all elements k such that $\lfloor \lg k \rfloor = t$, there are two cases: If t is a power of 2, the cycle containing $2\lfloor k/2 \rfloor$ also contains $2\lfloor k/2 \rfloor + 1$, so we must double the cycle leaders for $t-1$. Otherwise the cycle containing $2\lfloor k/2 \rfloor$ is disjoint from the cycle containing $2\lfloor k/2 \rfloor + 1$, so $L_t = (2L_{t-1}) \cup (2L_{t-1}+1) = (L_{t-1}*)_2$. This argument, discovered by Jörg Arndt in 2001, establishes the hint and yields the following algorithm:

P1. [Initialize.] Set $t \leftarrow 1$, $m \leftarrow 0$. (We may assume that $n \geq 2$.)

P2. [Loop through leaders.] Set $r \leftarrow m$. Perform Algorithm Q with $k = 2^t + r$; then if $r > 0$, set $r \leftarrow (r-1) \wedge m$ and repeat until $r = 0$. [See exercise 7.1–00.]

P3. [Increase $\lg k$.] Set $t \leftarrow t + 1$. Terminate if t is now equal to n; otherwise set $m \leftarrow 2m + [t \wedge (t-1) \neq 0]$ and return to P2. ▮

Q1. [Begin a cycle.] Set $s \leftarrow X_k$, $l \leftarrow k$, $j \leftarrow l \oplus \lfloor l/2 \rfloor$.

Q2. [Follow the cycle.] If $j \neq k$ set $X_l \leftarrow X_j$, $l \leftarrow j$, $j \leftarrow l \oplus \lfloor l/2 \rfloor$, and repeat until $j = k$. Then set $X_l \leftarrow s$. ▮

31. We get a field from f_n if and only if we get one from $f_n^{[2]}$, which takes $(a_{n-1} \dots a_0)_2$ to $\big((a_{n-1} \oplus a_{n-2})(a_{n-1} \oplus a_{n-3})(a_{n-2} \oplus a_{n-4}) \dots (a_2 \oplus a_0)(a_1)\big)_2$. Let $c_n(x)$ be the characteristic polynomial of the matrix A defining this transformation, mod 2; then $c_1(x) = x + 1$, $c_2(x) = x^2 + x + 1$, and $c_{n+1}(x) = xc_n(x) + c_{n-1}(x)$. Since $c_n(A)$ is the zero matrix, by the Cayley–Hamilton theorem, a field is obtained if and only if $c_n(x)$ is a primitive polynomial, and this condition can be tested as in Section 3.2.2. The first such values of n are 1, 2, 3, 5, 6, 9, 11, 14, 23, 29, 30, 33, 35, 39, 41, 51, 53, 65, 69, 74, 81, 83, 86, 89, 90, 95.

[Running the recurrence backwards shows that $c_{-j-2}(x) = c_j(x)$, hence $c_j(x)$ divides $c_{(2j+1)k+j}(x)$; for example, $c_{3k+1}(x)$ is always a multiple of $x+1$. All numbers n of the form $2jk + j + k$ are therefore excluded when $j > 0$ and $k > 0$. The polynomials $c_{18}(x)$, $c_{50}(x)$, $c_{98}(x)$, and $c_{99}(x)$ are irreducible but not primitive.]

32. Mostly true, but false at the points where x changes sign. (Walsh originally suggested that $w_k(x)$ should be zero at such points; but the convention adopted here is better, because it makes simple formulas like (15)–(19) valid for all x.)

33. By induction on k, we have

$$w_k(x) = w_{\lfloor k/2 \rfloor}(2x) = r_1(2x)^{b_1+b_2} r_2(2x)^{b_2+b_3} \dots = r_1(x)^{b_0+b_1} r_2(x)^{b_1+b_2} r_3(x)^{b_2+b_3} \dots$$

for $0 \leq x < \frac{1}{2}$, because $r_j(2x) = r_{j+1}(x)$ and $r_1(x) = 1$ in this range. And when $\frac{1}{2} \leq x < 1$,

$$\begin{aligned} w_k(x) = (-1)^{\lceil k/2 \rceil} w_{\lfloor k/2 \rfloor}(2x-1) &= r_1(x)^{b_0+b_1} r_1(2x-1)^{b_1+b_2} r_2(2x-1)^{b_2+b_3} \dots \\ &= r_1(x)^{b_0+b_1} r_2(x)^{b_1+b_2} r_3(x)^{b_2+b_3} \dots \end{aligned}$$

because $\lceil k/2 \rceil \equiv b_0 + b_1$ (modulo 2) and $r_j(2x-1) = r_{j+1}(x - \frac{1}{2}) = r_{j+1}(x)$ for $j \geq 1$.

34. $p_k(x) = \prod_{j \geq 0} r_{j+1}^{b_j}$; hence $w_k(x) = p_k(x) p_{\lfloor k/2 \rfloor}(x) = p_{g(k)}(x)$. [R. E. A. C. Paley, *Proc. London Math. Soc.* (2) **34** (1932), 241–279.]

35. If $j = (a_{n-1}\dots a_0)_2$ and $k = (b_{n-1}\dots b_0)_2$, the element in row j and column k is $(-1)^{f(j,k)}$, where $f(j,k)$ is the sum of all $a_r b_s$ such that: $r = s$ (Hadamard); $r+s = n-1$ (Paley); $r + s = n$ or $n-1$ (Walsh).

Let R_n, F_n, and G_n be permutation matrices for the permutations that take $j = (a_{n-1}\dots a_0)_2$ to $k = (a_0\dots a_{n-1})_2$, $k = 2^n - 1 - j = (\overline{a}_{n-1}\dots\overline{a}_0)_2$, and $k = g^{[-1]}(j) = \big((a_{n-1})\dots(a_{n-1}\oplus\cdots\oplus a_0)\big)_2$, respectively. Then, using the Kronecker product of matrices, we have the recursive formulas

$$R_{n+1} = \begin{pmatrix} R_n\otimes(1\ 0)\\ R_n\otimes(0\ 1)\end{pmatrix},\qquad F_{n+1} = F_n\otimes\begin{pmatrix}0&1\\1&0\end{pmatrix},\qquad G_{n+1} = \begin{pmatrix}G_n&0\\0&G_nF_n\end{pmatrix},$$

$$H_{n+1} = H_n\otimes\begin{pmatrix}1&1\\1&\overline{1}\end{pmatrix},\qquad P_{n+1} = \begin{pmatrix}P_n\otimes(1\ 1)\\ P_n\otimes(1\ \overline{1})\end{pmatrix},\qquad W_{n+1} = \begin{pmatrix}W_n\otimes(1\ 1)\\ F_nW_n\otimes(1\ \overline{1})\end{pmatrix}.$$

Thus $W_n = G_n^TP_n = P_nG_n$; $H_n = P_nR_n = R_nP_n$; and $P_n = W_nG_n^T = G_nW_n = H_nR_n = R_nH_n$.

36. **W1.** [Hadamard transform.] For $k = 0, 1, \dots, n-1$, replace the pair (X_j, X_{j+2^k}) by $(X_j + X_{j+2^k}, X_j - X_{j+2^k})$ for all j with $\lfloor j/2^k\rfloor$ even, $0 \le j < 2^n$. (These operations effectively set $X^T \leftarrow H_nX^T$.)

W2. [Bit reversal.] Apply the algorithm of exercise 5 to the vector X. (These operations effectively set $X^T \leftarrow R_nX^T$, in the notation of exercise 35.)

W3. [Gray binary permutation.] Apply the algorithm of exercise 30 to the vector X. (These operations effectively set $X^T \leftarrow G_n^TX^T$.) ∎

If n has one of the special values in exercise 31, it may be faster to combine steps W2 and W3 into a single permutation step.

37. If $k = 2^{e_1}+\cdots+2^{e_t}$ with $e_1 > \cdots > e_t \ge 0$, the sign changes occur at $S_{e_1}\cup\cdots\cup S_{e_t}$, where

$$S_0 = \Big\{\frac12\Big\},\quad S_1 = \Big\{\frac14,\frac34\Big\},\quad \dots,\quad S_e = \Big\{\frac{2j+1}{2^e}\ \Big|\ 0\le j<2^e\Big\}.$$

Therefore the number of sign changes in $(0\,..\,x)$ is $\sum_{j=1}^t\lfloor 2^{e_j}x+\frac12\rfloor$. Setting $x = l/(k+1)$ gives $l+O(t)$ changes; so the lth is at a distance of at most $O(\nu(k))/2^{\lfloor\lg k\rfloor}$ from $l/(k+1)$.

[This argument makes it plausible that infinitely many pairs (k,l) exist with $|z_{kl} - l/(k+1)| = \Omega\big((\log k)/k\big)$. But no explicit construction of such "bad" pairs is immediately apparent.]

38. Let $t_0(x) = 1$ and $t_k(x) = \omega^{\lfloor 3x\rfloor\lceil 2k/3\rceil}t_{\lfloor k/3\rfloor}(3x)$, where $\omega = e^{2\pi i/3}$. Then $t_k(x)$ winds around the origin $\frac23k$ times as x increases from 0 to 1. If $s_k(x) = \omega^{\lfloor 3^kx\rfloor}$ is the ternary analog of the Rademacher function $r_k(x)$, we have $t_k(x) = \prod_{j\ge0}s_{j+1}(x)^{b_j-b_{j+1}}$ when $k = (b_{n-1}\dots b_0)_3$, as in the modular ternary Gray code.

39. Let's call the symbols $\{x_0, x_1, \dots, x_7\}$ instead of $\{a,b,c,d,e,f,g,h\}$. We want to find a permutation p of $\{0,1,\dots,7\}$ such that the matrix with $(-1)^{j\cdot k}x_{p(j)\oplus k}$ in row j and column k has orthogonal rows; this condition is equivalent to requiring that

$$(j+j')\cdot\big(p(j)+p(j')\big)\equiv 1\ (\text{modulo }2),\qquad \text{for } 0\le j<j'<8.$$

One solution is $p(0)\dots p(7) = 0\,1\,7\,2\,5\,6\,3\,4$, yielding the identity $(a^2+b^2+c^2+d^2+e^2+f^2+g^2+h^2)(A^2+B^2+C^2+D^2+E^2+F^2+G^2+H^2) = \mathcal{A}^2+\mathcal{B}^2+\mathcal{C}^2+\mathcal{D}^2+$

$\mathcal{E}^2+\mathcal{F}^2+\mathcal{G}^2+\mathcal{H}^2$, where

$$\begin{pmatrix}\mathcal{A}\\ \mathcal{B}\\ \mathcal{C}\\ \mathcal{D}\\ \mathcal{E}\\ \mathcal{F}\\ \mathcal{G}\\ \mathcal{H}\end{pmatrix} = \begin{pmatrix} a & b & c & d & e & f & g & h\\ b & -a & d & -c & f & -e & h & -g\\ h & g & -f & -e & d & c & -b & -a\\ c & -d & -a & b & g & -h & -e & f\\ f & e & h & g & -b & -a & -d & -c\\ g & -h & e & -f & -c & d & -a & b\\ d & c & -b & -a & -h & -g & f & e\\ e & -f & -g & h & -a & b & c & -d\end{pmatrix}\begin{pmatrix}A\\ B\\ C\\ D\\ E\\ F\\ G\\ H\end{pmatrix}.$$

(b) There *is* no 16×16 solution. The closest one can come is

$$p(0)\ldots p(15) \;=\; 0\ 1\ 11\ 2\ 14\ 15\ 13\ 4\ 9\ 10\ 7\ 12\ 5\ 6\ 3\ 8,$$

which fails if and only if $j\oplus j'=5$. [See *Philos. Mag.* **34** (1867), 461–475. In §9, §10, §11, and §13 of this paper, Sylvester stated and proved the basic results about what has somehow come to be known as the Hadamard transform — although Hadamard himself gave credit to Sylvester [*Bull. des Sciences Mathématiques* (2) **17** (1893), 240–246]. Moreover, Sylvester introduced transforms of m^n elements in §14, using mth roots of unity.]

40. Yes; this change would in fact run through the swapped subsets in lexicographic binary order rather than in Gray binary order. (Any 5×5 matrix of 0s and 1s that is nonsingular mod 2 will generate all 32 possibilities when we run through all linear combinations of its rows.) The most important thing is the appearance of the ruler function, or some other Gray code delta sequence, not the fact that only one a_j changes per step, in cases like this where any number of the a_j can be changed simultaneously at the same cost.

41. At most 16; for example, `fired`, `fires`, `finds`, `fines`, `fined`, `fares`, `fared`, `wares`, `wards`, `wands`, `wanes`, `waned`, `wines`, `winds`, `wires`, `wired`. We also get 16 from `paced`/`links` and `paled`/`mints`; perhaps also from a word mixed with an antipodal nonword.

42. Suppose $n\le 2^{2^r}+r+1$, and let $s=2^r$. We use an auxiliary table of 2^{r+s} bits f_{jk} for $0\le j<2^s$ and $0\le k<s$, representing focus pointers as in Algorithm L, together with an auxiliary s-bit "register" $j=(j_{s-1}\ldots j_0)_2$ and an $(r+2)$-bit "program counter" $p=(p_{r+1}\ldots p_0)_2$. At each step we examine the program counter and possibly the j register and one of the f bits; then, based on the bits seen, we complement a bit of the Gray code, complement a bit of the program counter, and possibly change a j or f bit, thereby emulating step L3 with respect to the most significant $n-r-2$ bits.

For example, here is the construction when $r=1$:

$p_2p_1p_0$	Change	Set		$p_2p_1p_0$	Change	Set	
0 0 0	a_0,p_0	$j_0\leftarrow f_{00}$	$\}\ j\leftarrow f_0$	1 1 0	a_0,p_0	$f_{j0}\leftarrow f_{(j+1)0}$	$\}\ f_j\leftarrow f_{j+1}$
0 0 1	a_1,p_1	$j_1\leftarrow f_{01}$		1 1 1	a_1,p_1	$f_{j1}\leftarrow f_{(j+1)1}$	
0 1 1	a_0,p_0	$f_{00}\leftarrow 0$	$\}\ f_0\leftarrow 0$	1 0 1	a_0,p_0	$f_{(j+1)0}\leftarrow (j+1)_0$	$\}\ f_{j+1}\leftarrow j+1$
0 1 0	a_2,p_2	$f_{01}\leftarrow 0$		1 0 0	a_{j+3},p_2	$f_{(j+1)1}\leftarrow (j+1)_1$	

The process stops when it attempts to change bit a_n.

[In fact, we need change only *one* auxiliary bit per step if we allow ourselves to examine some Gray binary bits as well as the auxiliary bits, because $p_r\ldots p_0=a_r\ldots a_0$, and we can set $f_0\leftarrow0$ in a more clever way when j doesn't have its final value 2^s-1. This construction, suggested by Fredman in 2001, improves on another that he had

published in *SICOMP* **7** (1978), 134–146. With a more elaborate construction it is possible to reduce the number of auxiliary bits to $O(n)$.]

43. This number was estimated by Silverman, Vickers, and Sampson [*IEEE Trans.* **IT-29** (1983), 894–901] to be about 7×10^{22}. Exact calculation might be feasible because every 6-bit Gray code has only five or fewer segments that lie in a 5-cube corresponding to at least one of the six coordinates. (In unpublished work, Steve Winker had used a similar idea to evaluate $d(5)$ in less than 15 minutes on a "generic" computer in 1972.)

44. All $(n+1)$-bit delta sequences with just two occurrences of the coordinate j are produced by the following construction: Take n-bit delta sequences $\delta_0 \ldots \delta_{2^n-1}$ and $\varepsilon_0 \ldots \varepsilon_{2^n-1}$ and find an index k with $\delta_k = \varepsilon_0$. Form the cycle

$$\delta_0 \ldots \delta_{k-1}\, n\, \varepsilon_1 \ldots \varepsilon_{2^n-1}\, n\, \delta_{k+1} \ldots \delta_{2^n-1}$$

and then interchange $n \leftrightarrow j$.

All $(n-2)$-bit delta sequences with just two occurrences of coordinates h and j (with h before j) are, similarly, produced from four n-bit sequences $\delta_0 \ldots \delta_{2^n-1}$, $\ldots$, $\eta_0 \ldots \eta_{2^n-1}$ and an index k with $\delta_k = \varepsilon_0 = \zeta_0 = \eta_0$, by interchanging $n \leftrightarrow h$ and $n+1 \leftrightarrow j$ in

$$\delta_0 \ldots \delta_{k-1}\, n\, \varepsilon_1 \ldots \varepsilon_{2^n-1} (n{+}1)\, \zeta_1 \ldots \zeta_{2^n-1}\, n\, \eta_1 \ldots \eta_{2^n-1} (n{+}1)\, \delta_{k+1} \ldots \delta_{2^n-1}.$$

Let $a(n)$ and $b(n)$ be the answers to (a) and (b), for $n \geq 1$. The first construction shows that $a(n+1) + 2b(n+1) = 2^n(n+1)\,d(n)^2/n$, because it produces the delta sequences enumerated by $b(n+1)$ in two ways. The second construction shows that $b(n+2) = 2^n(n+2)(n+1)\,d(n)^4/n^3$.

45. We have $d(n+1) \geq 2^n d(n)^2/n$, because $2^n d(n)^2/n$ is the number of $(n+1)$-bit delta sequences with exactly two appearances of 0. Hence $d(n) > \frac{5}{32}2^{2^n}$ for $n \geq 5$, by induction on n.

Indeed, we can establish even faster growth by using the previous exercise, because $d(n+1) \geq a(n+1) + b(n+1)$ and $b(n+1) \leq \frac{25}{64}(n+1)\,d(n)^2/n$ for $n \geq 5$. Hence $d(n+1) \geq \left(2^n - \frac{25}{64}\right)(n+1)\,d(n)^2/n$ for $n \geq 5$, and iteration of this relation shows that

$$\lim_{n\to\infty} d(n)^{1/2^n} \geq d(5)^{1/32} \prod_{n=5}^{\infty} \left(2^n - \frac{25}{64}\right)^{1/2^{n+1}} \left(\frac{n+1}{n}\right)^{1/2^{n+1}} \approx 2.3606.$$

[See R. J. Douglas, *Disc. Math.* **17** (1977), 143–146; M. Mollard, *European J. Comb.* **9** (1988), 49–52.] But the true value of this limit is probably ∞.

46. Leo Moser (unpublished) has conjectured that it is $\sim n/e$. So far only an upper bound of about $n/\sqrt{2}$ has been established; see the references in the previous answer.

48. If $d(n,k,v)$ of the codes begin with $g(0) \ldots g(k-1)\,v$, the conjecture implies that $d(n,k,v) \leq d(n,k,g(k))$, because the reverse of a Gray code is a Gray code. Thus the hint follows from $d(n) = d(n,1)$ and

$$d(n,k) = \sum_v \{\, d(n,k,v) \mid v \mathrel{-\!\!-} g(k-1), v \notin S_k \,\} \leq c_{nk} d(n,k,g(k)) = d(n,k+1).$$

Finally, $d(n,2^n) = 1$, hence $d(n) \leq \prod_{k=1}^{2^n-1} c_{nk} = \prod_{k=1}^{n} k^{\binom{n}{k}} = n \prod_{k=1}^{n-1} \big(k(n-k)\big)^{\binom{n}{k}/2} \leq n \prod_{k=1}^{n-1} (n/2)^{\binom{n}{k}} = n(n/2)^{2^n-2}$. [*IEEE Trans.* **IT-29** (1983), 894–901.]

49. Take any Hamiltonian path P from $0 \ldots 0$ to $1 \ldots 1$ in the $(2n-1)$-cube, such as the Savage–Winkler path, and use $0P$, $1P^R$. (All such codes are obtained by this construction when $n = 1$ or $n = 2$, but many more possibilities exist when $n > 2$.)

50. $\alpha_1(n+1)\alpha_1^R n\,\alpha_1 j_1\alpha_2 n\alpha_2^R(n+1)\alpha_2\ldots j_{l-1}\alpha_l n\,\alpha_l^R(n+1)\alpha_l n\,\alpha_l^R j_{l-1}\ldots j_1\alpha_1^R n$.

51. We can assume that $n>3$ and that we have an n-bit Gray code with transition counts $c_j=2\lfloor(2^{n-1}+j)/n\rfloor$; we want to construct an $(n+2)$-bit code with transition counts $c'_j=2\lfloor(2^{n+1}+j)/(n+2)\rfloor$. If $2^{n+1}\bmod(n+2)\ge 2$, we can use Theorem D with $l=2\lfloor 2^{n+1}/(n+2)\rfloor+1$, underlining b_j copies of j where $b_j=4\lfloor(2^{n-1}+j)/n\rfloor-\lfloor(2^{n+1}+j)/(n+2)\rfloor-[j=0]$ and putting an underlined 0 last. This is always easy to do because $\left|b_j-2^{n+2}/n(n+2)\right|<5$. A similar construction works if $2^{n+1}\bmod(n+2)\le n$, with $l=2\lfloor 2^{n+1}/(n+2)\rfloor-1$ and $b_j=4\lfloor(2^{n-1}+j)/n\rfloor-\lfloor(2^{n+1}+j+2)/(n+2)\rfloor-[j=0]$. In fact, $2^{n+1}\bmod(n+2)$ is always $\le n$ [see K. Kedlaya, *Electronic J. Combinatorics* **3** (1996), comment on #R25 (9 April 1997)]. The basic idea of this proof is due to J. P. Robinson and M. Cohn [*IEEE Trans.* **C-30** (1981), 17–23].

52. The number of different code patterns in the smallest j coordinate positions is at most $c_0+\cdots+c_{j-1}$.

53. Notice that Theorem D produces only codes with $c_j=c_{j+1}$ for some j, so it cannot produce the counts $(2,4,6,8,12)$. The extension in exercise 50 gives also $c_j=c_{j+1}-2$, but it cannot produce $(6,10,14,18,22,26,32)$. The sets of numbers satisfying the conditions of exercise 52 are precisely those obtainable by starting with $\{2,2,4,\ldots,2^{n-1}\}$ and repeatedly replacing some pair $\{c_j,c_k\}$ for which $c_j<c_k$ by the pair $\{c_j+2,c_k-2\}$.

54. Suppose the values are $\{p_1,\ldots,p_n\}$, and let x_{jk} be the number of times p_j occurs in $(a_1,\ldots,a_k)$. We must have $(x_{1k},\ldots,x_{nk})\equiv(x_{1l},\ldots,x_{nl})$ (modulo 2) for some $k<l$. But if the p's are prime numbers, varying as the delta sequence of an n-bit Gray code, the only solution is $k=0$ and $l=2^n$. [*AMM* **60** (1953), 418; **83** (1976), 54.]

56. [*Bell System Tech. J.* **37** (1958), 815–826.] The 112 canonical delta sequences yield

Class	Example	t	Class	Example	t	Class	Example	t
A	0102101302012023	2	D	0102013201020132	4	G	0102030201020302	8
B	0102303132101232	2	E	0102032021202302	4	H	0102101301021013	8
C	0102030130321013	2	F	0102013102010232	4	I	0102013121012132	1

Here B is the balanced code (Fig. 13(b)), G is standard Gray binary (Fig. 10(b)), and H is the complementary code (Fig. 13(a)). Class H is also equivalent to the modular $(4,4)$ Gray code under the correspondence of exercise 18. A class with t automorphisms corresponds to $32\times 24/t$ of the 2688 different delta sequences $\delta_0\delta_1\ldots\delta_{15}$.

Similarly (see exercise 7.2.3–00), the 5-bit Gray codes fall into 237,675 different equivalence classes.

57. With Type 1 only, 480 vertices are isolated, namely those of classes D, F, G in the previous answer. With Type 2 only, the graph has 384 components, 288 of which are isolated vertices of classes F and G. There are 64 components of size 9, each containing 3 vertices from E and 6 from A; 16 components of size 30, each with 6 from H and 24 from C; and 16 components of size 84, each with 12 from D, 24 from B, 48 from I. With Type 3 (or Type 4) only, the entire graph is connected. [Similarly, all 91,392 of the 4-bit Gray *paths* are connected if path $\alpha\beta$ is considered adjacent to path $\alpha^R\beta$. Vickers and Silverman, *IEEE Trans.* **C-29** (1980), 329–331, have conjectured that Type 3 changes will suffice to connect the graph of n-bit Gray codes for all $n\ge 3$.]

58. If some nonempty substring of $\beta\beta$ involves each coordinate an even number of times, that substring cannot have length $|\beta|$, so some cyclic shift of β has a prefix γ

with the same evenness property. But then α doesn't define a Gray code, because we could change each n of γ back to 0.

59. If α is nonlocal in exercise 58, so is $\beta\beta$, provided that $q > 1$ and that 0 occurs more than $q+1$ times in α. Therefore, starting with the α of (30) but with 0 and 1 interchanged, we obtain nonlocal codes for $n \ge 5$ in which coordinate 0 changes exactly 6 times. [Mark Ramras, *Discrete Math.* **85** (1990), 329–331.] On the other hand, a 4-bit Gray code cannot be nonlocal because it always has a run of length 2; if $\delta_k = \delta_{k+2}$, elements $\{v_{k-1}, v_k, v_{k+1}, v_{k+2}\}$ form a 2-subcube.

60. Use the construction of exercise 58 with $q = 1$.

61. The idea is to interleave an m-bit code $U = (u_0, u_1, u_2, \dots)$ with an n-bit code $V = (v_0, v_1, v_2, \dots)$, by forming concatenations

$$W = (u_{i_0}v_{j_0},\ u_{i_1}v_{j_1},\ u_{i_2}v_{j_2},\ \dots), \qquad i_k = \overline{a}_0 + \cdots + \overline{a}_{k-1}, \quad j_k = a_0 + \cdots + a_{k-1},$$

where $a_0a_1a_2\dots$ is a periodic string of control bits $\alpha\alpha\alpha\dots$; we advance to the next element of U when $a_k = 0$, otherwise to the next element of V.

If α is any string of length $2^m \le 2^n$, containing s bits that are 0 and $t = 2^m - s$ bits that are 1, W will be an $(m+n)$-bit Gray code if s and t are odd. For we have $i_{k+l} \equiv i_k$ (modulo 2^m) and $j_{k+l} \equiv j_k$ (modulo 2^n) only if l is a multiple of 2^m, since $i_k + j_k = k$. Suppose $l = 2^m c$; then $j_{k+l} = j_k + tc$, so c is a multiple of 2^n.

(a) Let $\alpha = 0111$; then runs of length 8 occur in the left 2 bits and runs of length $\ge \lfloor \frac{4}{3} r(n) \rfloor$ occur in the right n bits.

(b) Let s be the largest odd number $\le 2^m r(m)/(r(m)+r(n))$. Also let $t = 2^m - s$ and $a_k = \lfloor (k+1)t/2^m \rfloor - \lfloor kt/2^m \rfloor$, so that $i_k = \lceil ks/2^m \rceil$ and $j_k = \lfloor kt/2^m \rfloor$. If a run of length l occurs in the left m bits, we have $i_{k+l+1} \ge i_k + r(m) + 1$, hence $l + 1 > 2^m r(m)/s \ge r(m) + r(n)$. And if it occurs in the right n bits we have $j_{k+l+1} \ge j_k + r(n) + 1$, hence

$$\begin{aligned} l + 1 &> 2^m r(n)/t > 2^m r(n)/\big(2^m r(n)/(r(m)+r(n)) + 2\big) \\ &= r(m) + r(n) - \frac{2(r(m)+r(n))^2}{2^m r(n) + 2(r(m)+r(n))} > r(m) + r(n) - 1 \end{aligned}$$

because $r(m) \le r(n)$.

The construction often works also in less restricted cases. See the paper that introduced the study of Gray code runs: L. Goddyn, G. M. Lawrence, and E. Nemeth, *Utilitas Math.* **34** (1988), 179–192.

63. Set $a_k \leftarrow k \bmod 4$ for $0 \le k < 2^{10}$, except that $a_k = 4$ when $k \bmod 16 = 15$ or $k \bmod 64 = 42$ or $k \bmod 256 = 133$. Also set $(j_0, j_1, j_2, j_3, j_4) \leftarrow (0, 2, 4, 6, 8)$. Then for $k = 0$, 1, ..., 1023, set $\delta_k \leftarrow j_{a_k}$ and $j_{a_k} \leftarrow 1 + 4a_k - j_{a_k}$. (This construction generalizes the method of exercise 61.)

64. (a) Each element u_k appears together with $\{v_k, v_{k+2^m}, \dots, v_{k+2^m(2^{n-1}-1)}\}$ and $\{v_{k+1}, v_{k+1+2^m}, \dots, v_{k+1+2^m(2^{n-1}-1)}\}$. Thus the permutation $\sigma_0 \dots \sigma_{2^m-1}$ must be a 2^{n-1}-cycle containing the n-bit vertices of even parity, times an arbitrary permutation of the other vertices. This condition is also sufficient.

(b) Let τ_j be the permutation that takes $v \mapsto v \oplus 2^j$, and let $\pi_j(u, w)$ be the permutation $(uw)\tau_j$. If $u \oplus w = 2^i + 2^j$ then $\pi_j(u, w)$ takes $u \mapsto u \oplus 2^i$ and $w \mapsto w \oplus 2^i$, while $v \mapsto v \oplus 2^j$ for all other vertices v, so it takes each vertex to a neighbor.

If S is any set $\subseteq \{0, \dots, n-1\}$, let $\sigma(S)$ be the stream of all permutations τ_j for all $j \in \{0, \dots, n-1\} \setminus S$, in increasing order of j, repeated twice; for example, if $n = 5$

we have $\sigma(\{1,2\}) = \tau_0\tau_3\tau_4\tau_0\tau_3\tau_4$. Then the Gray stream

$$\Sigma(i,j,u) = \sigma(\{i,j\})\,\pi_j(u, u\oplus 2^i\oplus 2^j)\,\sigma(\{i,j\})\,\tau_j\,\sigma(\{j\})$$

consists of $6n-8$ permutations whose product is the transposition $(u\ \ u\oplus 2^i\oplus 2^j)$. Moreover, when this stream is applied to any n-bit vertex v, its runs all have length $\ge n-2$.

We may assume that $n \ge 5$. Let $\delta_0 \ldots \delta_{2^n-1}$ be the delta sequence for an n-bit Gray code $(v_0, v_1, \ldots, v_{2^n-1})$ with all runs of length 3 or more. Then the product of all permutations in

$$\Sigma = \prod_{k=1}^{2^{n-1}-1} \big(\Sigma(\delta_{2k-1}, \delta_{2k}, v_{2k-1})\,\Sigma(\delta_{2k}, \delta_{2k+1}, v_{2k})\big)$$

is $(v_1\,v_3)(v_2\,v_4)\ldots(v_{2^n-3}\,v_{2^n-1})(v_{2^n-2}\,v_0) = (v_{2^n-1}\ldots v_1)(v_{2^n-2}\ldots v_0)$, so it satisfies the cycle condition of (a).

Moreover, all powers $(\sigma(\emptyset)\Sigma)^t$ produce runs of length $\ge n-2$ when applied to any vertex v. By repeating individual factors $\sigma(\{i,j\})$ or $\sigma(\{j\})$ in Σ as many times as we wish, we can adjust the length of $\sigma(\emptyset)\Sigma$, obtaining $2n+(2^{n-1}-1)(12n-16)+2(n-2)a+2(n-1)b$ for any integers $a, b \ge 0$; thus we can increase its length to exactly 2^m, provided that $2^m \ge 2n+(2^{n-1}-1)(12n-16)+2(n^2-5n+6)$, by exercise 5.2.1–21.

(c) The bound $r(n) \ge n - 4\lg n + 8$ can be proved for $n \ge 5$ as follows. First we observe that it holds for $5 \le n < 33$ by the methods of exercises 60–63. Then we observe that every integer $N \ge 33$ can be written as $N = m+n$ or $N = m+n+1$, for some $m \ge 20$, where

$$n = m - \lfloor 4\lg m \rfloor + 10.$$

If $m \ge 20$, 2^m is sufficiently large for the construction in part (b) to be valid; so we have

$$\begin{aligned} r(N) \ge r(m+n) \ge 2\min\big(r(m), n-2\big) &\ge 2\big(m - \lfloor 4\lg m\rfloor + 8\big) \\ &= m+n+1-\lfloor 4\lg(m+n) - 1 + \epsilon\rfloor + 8 \\ &\ge N - 4\lg N + 8 \end{aligned}$$

where $\epsilon = 4\lg\big(2m/(m+n)\big) < 1$. [To appear.] Recursive use of (b) gives, in fact, $r(1024) \ge 1000$.

65. A computer search reveals that eight essentially different patterns (and their reverses) are possible. One of them has the delta sequence 0102031420302404123 4214103234103, and it is close to two of the others.

66. (Solution by Mark Cooke.) One suitable delta sequence is 012345607012132435 657607102135346267015374123625670173142620657013421465605731024645375710204353761407363046427370356402713275054121027564150240365425013602541615604312576032572043157624321760452041751635476703564757062543724213262416152341751436714316431 4.

67. Let $v_{2k+1} = \overline{v}_{2k}$ and $v_{2k} = 0u_k$, where $(u_0, u_1, \ldots, u_{2^n-1})$ is any $(n-1)$-bit Gray code. [See Robinson and Cohn, *IEEE Trans.* **C-30** (1981), 17–23.]

68. Yes. The simplest way is probably to take $(n-1)$-trit modular Gray ternary code and add $0\ldots0$, $1\ldots1$, $2\ldots2$ to each string (modulo 3). For example, when $n=3$ the code is 000, 111, 222, 001, 112, 220, 002, 110, 221, 012, 120, 201, ..., 020, 101, 212.

69. (a) We need only verify the change in h when bits $b_{j-1}\dots b_0$ are simultaneously complemented, for $j = 1, 2, \dots$; and these changes are respectively $(1110)_2$, $(1101)_2$, $(0111)_2$, $(1011)_2$, $(10011)_2$, $(100011)_2$, ... To prove that every n-tuple occurs, note that $0 \le h(k) < 2^n$ when $0 \le k < 2^n$ and $n > 3$; also $h^{[-1]}((a_{n-1}\dots a_0)_2) = (b_{n-1}\dots b_0)_2$, where $b_0 = a_0 \oplus a_1 \oplus a_2 \oplus \cdots$, $b_1 = a_0$, $b_2 = a_2 \oplus a_3 \oplus a_4 \oplus \cdots$, $b_3 = a_0 \oplus a_1 \oplus a_3 \oplus \cdots$, and $b_j = a_j \oplus a_{j+1} \oplus \cdots$ for $j \ge 4$.

(b) Let $h(k) = (\dots a_2a_1a_0)_2$ where $a_j = b_j \oplus b_{j+1} \oplus b_0[j \le t] \oplus b_{t-1}[t-1 \le j \le t]$.

70. As in (32) and (33), we can remove a factor of $n!$ by assuming that the strings of weight 1 occur in order. Then there are 14 solutions for $n = 5$ starting with 00000, and 21 starting with 00001. When $n = 6$ there are 46,935 of each type (related by reversal and complementation). When $n = 7$ the number is much, much larger, yet very small by comparison with the total number of 7-bit Gray paths.

71. Suppose that $\alpha_{n(j+1)}$ differs from α_{nj} in coordinate t_j, for $0 \le j < n-1$. Then $t_j = j\pi_n$, by (44) and (38). Now Eq. (34) tells us that $t_0 = n-1$; and if $0 < j < n-1$ we have $t_j = ((j-1)\pi_{n-1})\pi_{n-1}$ by (40). Thus $t_j = j\sigma_n\pi_{n-1}^2$ for $0 \le j < n-1$, and the value of $(n-1)\pi_n$ is whatever is left. (Notations for permutations are notoriously confusing, so it is always wise to check a few small cases carefully.)

72. The delta sequence is 0102132430201234012313041021323.

73. Let $Q_{nj} = P_{nj}^R$ and denote the sequences (41), (42) by S_n and T_n. Thus $S_n = P_{n0}Q_{n1}P_{n2}\dots$ and $T_n = Q_{n0}P_{n1}Q_{n2}\dots$, if we omit the commas; and we have

$$S_{n+1} = 0P_{n0}\ 0Q_{n1}\ 1Q_{n0}^{\pi}\ 1P_{n1}^{\pi}\ 0P_{n2}\ 0Q_{n3}\ 1Q_{n2}^{\pi}\ 1P_{n3}^{\pi}\ 0P_{n4}\ \dots,$$
$$T_{n+1} = 0Q_{n0}\ 1P_{n0}^{\pi}\ 0P_{n1}\ 0Q_{n2}\ 1Q_{n1}^{\pi}\ 1P_{n2}^{\pi}\ 0P_{n3}\ 0Q_{n4}\ 1Q_{n3}^{\pi}\ \dots,$$

where $\pi = \pi_n$, revealing a reasonably simple joint recursion between the delta sequences Δ_n and E_n of S_n and T_n. Namely, if we write

$$\Delta_n = \phi_1\, a_1\, \phi_2\, a_2 \dots \phi_{n-1}\, a_{n-1}\, \phi_n, \qquad E_n = \psi_1\, b_1\, \psi_2\, b_2 \dots \psi_{n-1}\, b_{n-1}\, \psi_n,$$

where each ϕ_j and ψ_j is a string of length $2\binom{n-1}{j-1} - 1$, the next sequences are

$$\Delta_{n+1} = \phi_1\ a_1\ \phi_2\ n\ \psi_1\pi\ b_1\pi\ \psi_2\pi\ n\ \phi_3\ a_3\ \phi_4\ n\ \psi_3\pi\ b_3\pi\ \psi_4\pi\ n\ \dots$$
$$E_{n+1} = \psi_1\ n\ \phi_1\pi\ n\ \psi_2\ b_2\ \psi_3\ n\ \phi_2\pi\ a_2\pi\ \phi_3\pi\ n\ \psi_4\ b_4\ \psi_5\ n\ \phi_4\pi\ a_4\pi\ \phi_5\pi\ n\ \dots$$

For example, we have $\Delta_3 = 0\,\underline{1}\,0\,2\,1\,\underline{0}\,1$ and $E_3 = 0\,\underline{2}\,1\,2\,0\,\underline{2}\,1$, if we underline the a's and b's to distinguish them from the ϕ's and ψ's; and

$$\Delta_4 = 0\ 1\ 0\ 2\ 1\ 3\ 0\pi\ 2\pi\ 1\pi\ 2\pi\ 0\pi\ 3\ 1\ 3\ 1\pi = 0\ \underline{1}\ 0\ 2\ 1\ 3\ 2\ \underline{1}\ 0\ 1\ 2\ 3\ 1\ \underline{3}\ 0,$$
$$E_4 = 0\ 3\ 0\pi\ 3\ 1\ 2\ 0\ 2\ 1\ 3\ 0\pi\ 2\pi\ 1\pi\ 0\pi\ 1\pi = 0\ \underline{3}\ 2\ 3\ 1\ 2\ 0\ \underline{2}\ 1\ 3\ 2\ 1\ 0\ \underline{2}\ 0;$$

here $a_3\phi_4$ and $b_3\psi_4$ are empty. Elements have been underlined for the next step.

Thus we can compute the delta sequences in memory as follows. Here $p[j] = j\pi_n$ for $1 \le j < n$; $s_k = \delta_k$, $t_k = \varepsilon_k$, and $u_k = [\delta_k$ and ε_k are underlined], for $0 \le k < 2^n - 1$.

R1. [Initialize.] Set $n \leftarrow 1$, $p[0] \leftarrow 0$, $s_0 \leftarrow t_0 \leftarrow u_0 \leftarrow 0$.

R2. [Advance n.] Perform Algorithm S below, which computes the arrays s', t', and u' for the next value of n; then set $n \leftarrow n+1$.

R3. [Ready?] If n is sufficiently large, the desired delta sequence Δ_n is in array s'; terminate. Otherwise keep going.

R4. [Compute π_n.] Set $p'[0] = n-1$, and $p'[j] = p[p[j-1]]$ for $1 \le j < n$.

R5. [Prepare to advance.] Set $p[j] \leftarrow p'[j]$ for $0 \le j < n$; set $s_k \leftarrow s'_k$, $t_k \leftarrow t'_k$, and $u_k \leftarrow u'_k$ for $0 \le k < 2^n - 1$. Return to R2. ∎

In the following steps, "Transmit stuff(l,j) while $u_j=0$" is an abbreviation for "If $u_j=0$, repeatedly stuff(l,j), $l\leftarrow l+1$, $j\leftarrow j+1$, until $u_j\neq 0$."

S1. [Prepare to compute Δ_{n+1}.] Set $j\leftarrow k\leftarrow l\leftarrow 0$ and $u_{2^n-1}\leftarrow -1$.

S2. [Advance j.] Transmit $s'_l\leftarrow s_j$ and $u'_l\leftarrow 0$ while $u_j=0$. Then go to S5 if $u_j<0$.

S3. [Advance j and k.] Set $s'_l\leftarrow s_j$, $u'_l\leftarrow 1$, $l\leftarrow l+1$, $j\leftarrow j+1$. Then transmit $s'_l\leftarrow s_j$ and $u'_l\leftarrow 0$ while $u_j=0$. Then set $s'_l\leftarrow n$, $u'_l\leftarrow 0$, $l\leftarrow l+1$. Then transmit $s'_l\leftarrow p[t_k]$ and $u'_l\leftarrow 0$ while $u_k=0$. Then set $s'_l\leftarrow p[t_k]$, $u'_l\leftarrow 1$, $l\leftarrow l+1$, $k\leftarrow k+1$. And once again transmit $s'_l\leftarrow p[t_k]$ and $u'_l\leftarrow 0$ while $u_k=0$.

S4. [Done with Δ_{n+1}?] If $u_k<0$, go to S6. Otherwise set $s'_l\leftarrow n$, $u'_l\leftarrow 0$, $l\leftarrow l+1$, $j\leftarrow j+1$, $k\leftarrow k+1$, and return to S2.

S5. [Finish Δ_{n+1}.] Set $s'_l\leftarrow n$, $u'_l\leftarrow 1$, $l\leftarrow l+1$. Then transmit $s'_l\leftarrow p[t[k]]$ and $u'_l\leftarrow 0$ while $u_k=0$.

S6. [Prepare to compute E_{n+1}.] Set $j\leftarrow k\leftarrow l\leftarrow 0$. Transmit $t'_l\leftarrow t_k$ while $u_k=0$. Then set $t'_l\leftarrow n$, $l\leftarrow l+1$.

S7. [Advance j.] Transmit $t'_l\leftarrow p[s_j]$ while $u_j=0$. Then terminate if $u_j<0$; otherwise set $t'_l\leftarrow n$, $l\leftarrow l+1$, $j\leftarrow j+1$, $k\leftarrow k+1$.

S8. [Advance k.] Transmit $t'_l\leftarrow t_k$ while $u_k=0$. Then go to S10 if $u_k<0$.

S9. [Advance k and j.] Set $t'_l\leftarrow t_k$, $l\leftarrow l+1$, $k\leftarrow k+1$. Then transmit $t'_l\leftarrow t_k$ while $u_k=0$. Then set $t'_l\leftarrow n$, $l\leftarrow l+1$. Then transmit $t'_l\leftarrow p[s_j]$ while $u_j=0$. Then set $t'_l\leftarrow p[s_j]$, $l\leftarrow l+1$, $j\leftarrow j+1$. Return to S7.

S10. [Finish E_{n+1}.] Set $t'_l\leftarrow n$, $l\leftarrow l+1$. Then transmit $t'_l\leftarrow p[s_j]$ while $u_j=0$. ∎

To generate the monotonic Savage–Winkler path for fairly large n, one can first generate Δ_{10} and E_{10}, say, or even Δ_{20} and E_{20}. Using these tables, a suitable recursive procedure will then be able to reach higher values of n with very little computational overhead per step, on the average.

74. If the monotonic path is $v_0,\ldots,v_{2^n-1}$ and if v_k has weight j, we have

$$2\sum_{t>0}\binom{n}{j-2t}+\big((j+\nu(v_0))\bmod 2\big)\le k\le 2\sum_{t\ge 0}\binom{n}{j-2t}+\big((j+\nu(v_0))\bmod 2\big)-2.$$

Therefore the maximum distance between vertices of respective weights j and $j+1$ is $2\big(\binom{n-1}{j-1}+\binom{n-1}{j}+\binom{n-1}{j+1}\big)-1$. The maximum value, approximately $3\cdot 2^n/\sqrt{2\pi n}$, occurs when j is approximately $n/2$. [This is only about three times the smallest value achievable in *any* ordering of the vertices, which is $\sum_{j=0}^{n-1}\binom{j}{\lfloor j/2\rfloor}$ by exercise 7.10–00.]

75. There are only five essentially distinct solutions, all of which turn out in fact to be Gray *codes*. The delta sequences are

$$0\,1\,2\,3\,0\,1\,2\,4\,2\,1\,0\,3\,2\,1\,0\,1\,2\,1\,0\,3\,2\,1\,0\,4\,0\,1\,2\,3\,0\,1\,2\ (1)$$
$$0\,1\,2\,3\,0\,1\,2\,4\,2\,1\,0\,3\,2\,1\,0\,1\,3\,0\,1\,2\,3\,0\,1\,4\,1\,0\,3\,2\,1\,0\,3\ (1)$$
$$0\,1\,2\,3\,0\,1\,2\,4\,2\,1\,0\,3\,2\,1\,0\,2\,0\,3\,2\,1\,0\,3\,2\,4\,2\,3\,0\,1\,2\,3\,0\ (2)$$
$$0\,1\,2\,3\,0\,1\,2\,4\,2\,3\,0\,1\,2\,3\,0\,2\,0\,1\,2\,3\,0\,1\,2\,4\,2\,3\,0\,1\,2\,3\,0\ (2)$$
$$0\,1\,2\,3\,4\,1\,0\,1\,2\,1\,0\,3\,0\,1\,4\,3\,2\,1\,0\,3\,0\,1\,4\,1\,0\,1\,2\,3\,4\,1\,0\ (3)$$

76. If $v_0, \ldots, v_{2^n-1}$ is trend-free, so is the $(n+1)$-bit code $0v_0$, $1v_0$, $1v_1$, $0v_1$, $0v_2$, $1v_2$, $\ldots$, $1v_{2^n-1}$, $0v_{2^n-1}$. Fig. 14(g) shows a somewhat more interesting construction, which generalizes the first solution of exercise 75 to an $(n+2)$-bit code

$$00\Gamma''^R,\ 01\Gamma'^R,\ 11\Gamma',\ 10\Gamma'',\ 10\Gamma,\ 11\Gamma''',\ 01\Gamma'''^R,\ 00\Gamma^R$$

where Γ is the n-bit sequences $g(1), \ldots, g(2^{n-1})$ and $\Gamma' = \Gamma \oplus g(1)$, $\Gamma'' = \Gamma \oplus g(2^{n-1})$, $\Gamma''' = \Gamma \oplus g(2^{n-1}+1)$. [An n-bit trend-free design that is *almost* a Gray code, having just four steps in which $\nu(v_k \oplus v_{k+1}) = 2$, was found for all $n \geq 3$ by C. S. Cheng, *Proc. Berkeley Conf. Neyman and Kiefer* **2** (Hayward, Calif.: Inst. of Math. Statistics, 1985), 619–633.]

77. Replace the array $(d_{n-1}, \ldots, d_0)$ by an array of sentinel values $(s_{n-1}, \ldots, s_0)$, with $s_j \leftarrow m_j - 1$ in step H1. Set $a_j \leftarrow (a_j + 1) \bmod m_j$ in step H4. If $a_j = s_j$ in step H5, set $s_j \leftarrow (s_j - 1) \bmod m_j$, $f_j \leftarrow f_{j+1}$, $f_{j+1} \leftarrow j+1$.

78. For (50), notice that B_{j+1} is the number of times reflection has occurred in coordinate j, because we bypass coordinate j on steps that are multiples of $m_j \ldots m_0$. Hence, if $b_j < m_j$, an increase of b_j by 1 causes a_j to increase or decrease by 1 as appropriate. Furthermore, if $b_i = m_i - 1$ for $0 \leq i < j$, changing all these b_i to 0 when incrementing b_j will increase each of $B_0, \ldots, B_j$ by 1, thereby leaving the values $a_0, \ldots, a_{j-1}$ unchanged in (50).

For (51), note that $B_j = m_j B_{j+1} + b_j \equiv m_j B_{j+1} + a_j + (m_j - 1)B_{j+1} \equiv a_j + B_{j+1}$ (modulo 2); hence $B_j \equiv a_j + a_{j+1} + \cdots$, and (51) is obviously equivalent to (50).

In the modular Gray code for general radices $(m_{n-1}, \ldots, m_0)$, let

$$mg(k) = \begin{bmatrix} a_{n-1}, & \ldots, & a_2, & a_1, & a_0 \\ m_{n-1}, & \ldots, & m_2, & m_1, & m_0 \end{bmatrix}$$

when k is given by (46). Then $a_j = (b_j - B_{j+1}) \bmod m_j$, because coordinate j has increased modulo m_j exactly $B_j - B_{j+1}$ times if we start at $(0, \ldots, 0)$. The inverse function, which determines the b's from the modular Gray a's, is $b_j = (a_j + a_{j+1} + a_{j+2} + \cdots) \bmod m_j$ in the special case that each m_j is a divisor of m_{j+1} (for example, if all m_j are equal). But the inverse has no simple form in general; it can be computed by using the recurrences $b_j = (a_j + B_{j+1}) \bmod m_j$, $B_j = m_j B_{j+1} + b_j$ for $j = n-1$, $\ldots$, 0, starting with $B_n = 0$.

[Reflected Gray codes for radix $m > 2$ were introduced by Ivan Flores in *IRE Trans.* **EC-5** (1956), 79–82; he derived (50) and (51) in the case that all m_j are equal. Modular Gray codes with general mixed radices were implicitly discussed by Joseph Rosenbaum in *AMM* **45** (1938), 694–696, but without the conversion formulas; conversion formulas when all m_j have a common value m were published by Martin Cohn, *Info. and Control* **6** (1963), 70–78.]

79. (a) The last n-tuple always has $a_{n-1} = m_{n-1} - 1$, so it is one step from $(0, \ldots, 0)$ only if $m_{n-1} = 2$. And this condition suffices to make the final n-tuple $(1, 0, \ldots, 0)$. [Similarly, the final subforest output by Algorithm K is adjacent to the initial one if and only if the leftmost tree is an isolated vertex.]

(b) The last n-tuple is $(m_{n-1}-1, 0, \ldots, 0)$ if and only if $m_{n-1} \ldots m_{j+1} \bmod m_j = 0$ for $0 \leq j < n-1$, because $b_j = m_j - 1$ and $B_j = m_{n-1} \ldots m_j - 1$.

80. Run through $p_1^{a_1} \ldots p_t^{a_t}$ using reflected Gray code with radices $m_j = e_j + 1$.

81. The first cycle contains the edge from (x, y) to $\big(x, (y+1) \bmod m\big)$ if and only if $(x+y) \bmod m \neq m-1$ if and only if the second cycle contains the edge from (x, y) to $\big((x+1) \bmod m, y\big)$.

82. There are two 4-bit Gray codes $(u_0, \ldots, u_{15})$ and $(v_0, \ldots, v_{15})$ that cover all edges of the 4-cube. (Indeed, the non-edges of classes A, B, D, H, and I in exercise 56 form Gray codes, belonging to the same class as their complement.) Therefore with 16-ary modular Gray code we can form the four desired cycles $(u_0u_0, u_0u_1, \ldots, u_0u_{15}, u_1u_{15}, \ldots, u_{15}u_0)$, $(u_0u_0, u_1u_0, \ldots, u_{15}u_0, u_{15}u_1, \ldots, u_0u_{15})$, $(v_0v_0, \ldots, v_{15}u_0)$, $(v_0v_0, \ldots, v_0v_{15})$.

In a similar way we can show that $n/2$ edge-disjoint n-bit Gray codes exist when n is 16, 32, 64, etc. [*Abhandlungen Math. Sem. Hamburg* **20** (1956), 13–16.] J. Aubert and B. Schneider [*Discrete Math.* **38** (1982), 7–16] have proved that the same property holds for *all* even values of $n \geq 4$, but no simple construction is known.

84. Calling the initial position $(2, 2)$, the 8-step solution in Fig. A-1 shows how the sequence progresses down to $(0, 0)$. In the first move, for example, the front half of the cord passes around and behind the right comb, then through the large right loop. The middle line should be read from right to left. The generalization to n pairs of loops would, similarly, take $3^n - 1$ steps.

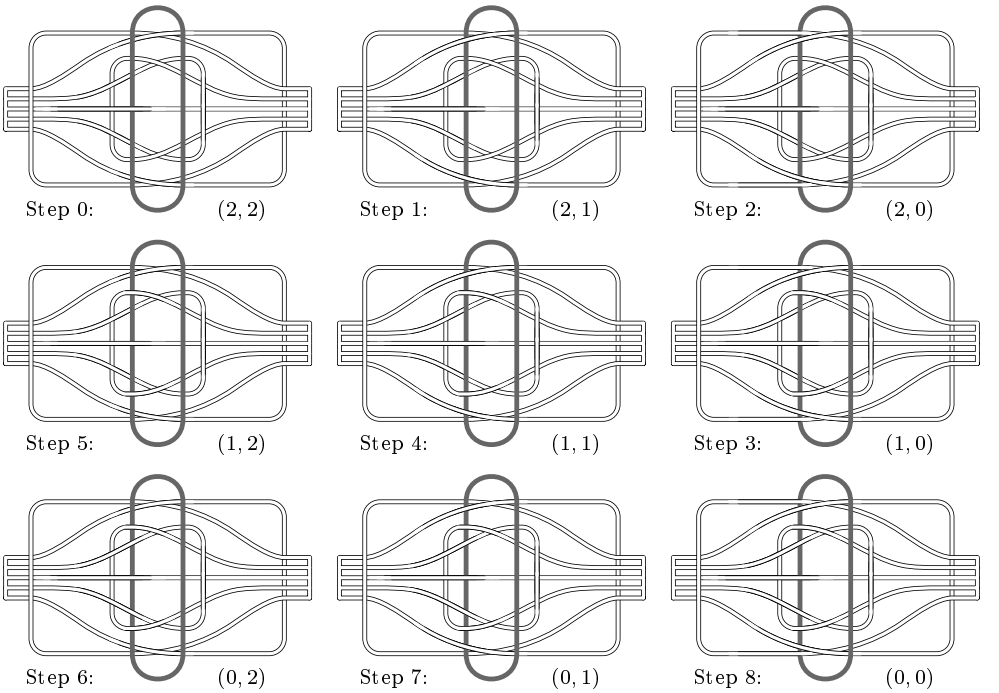


Fig. A-1.

[The origin of this delightful puzzle is obscure. *The Book of Ingenious & Diabolical Puzzles* by Jerry Slocum and Jack Botermans (1994) shows a 2-loop version carved from horn, probably made in China about 1850 [page 101], and a modern 6-loop version made in Malaysia about 1988 [page 93]. Slocum also owns a 4-loop version made from bamboo in England about 1884. He has found it listed in Henry Novra's *Catalogue of Conjuring Tricks and Puzzles* (1858 or 1859) and W. H. Cremer's *Games, Amusements, Pastimes and Magic* (1867), as well as in Hamleys' catalog of 1895, under the name "Marvelous Canoe Puzzle." Dyckman noted its connection to reflected Gray ternary in a letter to Martin Gardner, dated 2 August 1972.]

85. By (50), element $\left[\begin{smallmatrix} b, & b' \\ t, & t' \end{smallmatrix}\right]$ of $\Gamma \boxtimes \Gamma'$ is $\alpha_a \alpha'_{a'}$ if $rg(\left[\begin{smallmatrix} b, & b' \\ t, & t' \end{smallmatrix}\right]) = \left[\begin{smallmatrix} a, & a' \\ t, & t' \end{smallmatrix}\right]$ in the reflected Gray code for radices (t, t'). We can now show that element $\left[\begin{smallmatrix} b, & b', & b'' \\ t, & t', & t'' \end{smallmatrix}\right]$ of both $(\Gamma \boxtimes \Gamma') \boxtimes \Gamma''$ and $\Gamma \boxtimes (\Gamma' \boxtimes \Gamma'')$ is $\alpha_a \alpha'_{a'} \alpha''_{a''}$ if $rg(\left[\begin{smallmatrix} b, & b', & b'' \\ t, & t', & t'' \end{smallmatrix}\right]) = \left[\begin{smallmatrix} a, & a', & a'' \\ t, & t', & t'' \end{smallmatrix}\right]$ in the reflected Gray code for radices (t, t', t''). See exercise 4.1–10, and note also the mixed-radix law

$$m_1 \ldots m_n - 1 - \begin{bmatrix} x_1, & \ldots, & x_n \\ m_1, & \ldots, & m_n \end{bmatrix} = \begin{bmatrix} m_1 - 1 - x_1, & \ldots, & m_n - 1 - x_n \\ m_1, & \ldots, & m_n \end{bmatrix}.$$

In general, the reflected Gray code for radices $(m_1, \ldots, m_n)$ is $(0, \ldots, m_1 - 1) \boxtimes \cdots \boxtimes (0, \ldots, m_n - 1)$. [*Information Processing Letters* **22** (1986), 201–205.]

86. Let Γ_{mn} be the reflected m-ary Gray path, which can be defined by $\Gamma_{m0} = \epsilon$ and

$$\Gamma_{m(n+1)} = (0, 1, \ldots, m-1) \boxtimes \Gamma_{mn}, \qquad n \geq 0.$$

This path runs from $(0, 0, \ldots, 0)$ to $(m-1, 0, \ldots, 0)$ when m is even. Consider the Gray path Π_{mn} defined by $\Pi_{m0} = \emptyset$ and

$$\Pi_{m(n+1)} = \begin{cases} (0, 1, \ldots, m-1) \boxtimes \Pi_{mn},\ m\Gamma^R_{(m+1)n}, & \text{if } m \text{ is odd;} \\ (0, 1, \ldots, m) \boxtimes \Pi_{mn},\ m\Gamma^R_{mn}, & \text{if } m \text{ is even.} \end{cases}$$

This path traverses all of the $(m+1)^n - m^n$ nonnegative integer n-tuples for which $\max(a_1, \ldots, a_n) = m$, starting with $(0, \ldots, 0, m)$ and ending with $(m, 0, \ldots, 0)$. The desired infinite Gray path is Π_{0n}, Π^R_{1n}, Π_{2n}, Π^R_{3n}, ...

87. This is impossible when n is odd, because the n-tuples with $\max(|a_1|, \ldots, |a_n|) = 1$ include $\frac{1}{2}(3^n + 1)$ with odd parity and $\frac{1}{2}(3^n - 3)$ with even parity. When $n = 2$ we can use a spiral Σ_0, Σ_1, Σ_2, ..., where Σ_m winds clockwise from $(m, 1-m)$ to $(m, -m)$ when $m > 0$. For even values of $n \geq 2$, if T_m is a path of n-tuples from $(m, 1-m, m-1, 1-m, \ldots, m-1, 1-m)$ to $(m, -m, m, -m, \ldots, m, -m)$, we can use $\Sigma_m \boxtimes (T_0, \ldots, T_{m-1}), (\Sigma_0, \ldots, \Sigma_m)^R \,\overline{\boxtimes}\, T_m$ for $(n+2)$-tuples with the same property, where $\overline{\boxtimes}$ is the dual operation

$$\Gamma \,\overline{\boxtimes}\, \Gamma' = (\alpha_0\alpha'_0, \ldots, \alpha_{t-1}\alpha'_0, \alpha_{t-1}\alpha'_1, \ldots, \alpha_0\alpha'_1, \alpha_0\alpha'_2, \ldots, \alpha_{t-1}\alpha'_2, \alpha_{t-1}\alpha'_3, \ldots).$$

[Infinite n-dimensional Gray paths *without* the magnitude constraint were first constructed by E. Vázsonyi, *Acta Litterarum ac Scientiarum*, sectio Scientiarum Mathematicarum **9** (Szeged: 1938), 163–173.]

88. It would visit all the subforests again, but in reverse order, ending with $(0, \ldots, 0)$ and returning to the state it had after the initialization step K1. (This reflection principle is, in fact, the key to understanding how Algorithm K works.)

89. (a) Let $M_0 = \epsilon$, $M_1 = \bullet$, and $M_{n+2} = \bullet\, M^R_{n+1}$, $-M^R_n$. This construction works because the last element of M^R_{n+1} is the first element of M_{n+1}, namely a dot followed by the first element of M^R_n.

(b) Given a string $d_1 \ldots d_l$ where each d_j is $\bullet$ or $-$, we can find its successor by letting $k = l - [d_l = \bullet]$ and proceeding as follows: If k is odd and $d_k = \bullet$, change $d_k d_{k+1}$ to $-$; if k is even and $d_k = -$, change d_k to $\bullet\bullet$; otherwise decrease k by 1 and repeat until either making a change or reaching $k = 0$. The successor of the given word is $\bullet - - \bullet\bullet\bullet - \bullet - \bullet$.

90. A cycle can exist only when the number of code words is even, since the number of dashes changes by ± 1 at each step. Thus we must have $n \bmod 3 = 2$. The Gray paths M_n of exercise 89 are not suitable; they begin with $(\bullet -)^{\lfloor n/3 \rfloor} \bullet^{n \bmod 3}$ and end

with $(\mathit{—}\bullet)^{\lfloor n/3\rfloor}\bullet^{[n \bmod 3=1]}\mathit{—}^{[n \bmod 3=2]}$. But $M_{3k+1}\bullet$, $M_{3k}^R\mathit{—}$ is a Hamiltonian circuit in the Morse code graph when $n = 3k+2$.

91. Equivalently, the n-tuples $a_1\bar{a}_2a_3\bar{a}_4\ldots$ have no two consecutive 1s. Such n-tuples correspond to Morse code sequences of length $n+1$, if we append 0 and then represent $\bullet$ and $\bullet\mathit{—}$ respectively by 0 and 10. Under this correspondence we can convert the path M_{n+1} of exercise 89 into a procedure like Algorithm K, with the fringe containing the indices where each dot or dash begins (except for a final dot).

Q1. [Initialize.] Set $a_j \leftarrow \lfloor((j-1) \bmod 6)/3\rfloor$ and $f_j \leftarrow j$ for $1 \le j \le n$. Also set $f_0 \leftarrow 0$, $r_0 \leftarrow 1$, $l_1 \leftarrow 0$, $r_j \leftarrow j+(j \bmod 3)$ and $l_{j+(j \bmod 3)} \leftarrow j$ for $1 \le j \le n$, except if $j+(j \bmod 3) > n$ set $r_j \leftarrow 0$ and $l_0 \leftarrow j$. (The "fringe" now contains 1, 2, 4, 5, 7, 8, ...)

Q2. [Visit.] Visit the n-tuple $(a_1, \ldots, a_n)$.

Q3. [Choose p.] Set $q \leftarrow l_0$, $p \leftarrow f_q$, $f_q \leftarrow q$.

Q4. [Check a_p.] Terminate the algorithm if $p = 0$. Otherwise set $a_p \leftarrow 1 - a_p$ and go to Q6 if $a_p + p$ is now even.

Q5. [Insert $p+1$.] If $p < n$, set $q \leftarrow r_p$, $l_q \leftarrow p+1$, $r_{p+1} \leftarrow q$, $r_p \leftarrow p+1$, $l_{p+1} \leftarrow p$. Go to Q7.

Q6. [Delete $p+1$.] If $p < n$, set $q \leftarrow r_{p+1}$, $r_p \leftarrow q$, $l_q \leftarrow p$.

Q7. [Make p passive.] Set $f_p \leftarrow f_{l_p}$ and $f_{l_p} \leftarrow l_p$. Return to Q2. ∎

This algorithm can also be derived as a special case of a considerably more general method due to Gang Li, Frank Ruskey, and D. E. Knuth, which extends Algorithm K by allowing the user to specify either $a_p \ge a_q$ or $a_p \le a_q$ for each (parent, child) pair (p, q). [See Knuth and Ruskey, "Deconstructing coroutines," to appear.] A generalization in another direction, which produces all strings of length n that do not contain certain substrings, has been discovered by M. B. Squire, *Electronic J. Combinatorics* **3** (1996), #R17, 1–29.

92. Yes, because the digraph of all $(n-1)$-tuples $(x_1, \ldots, x_{n-1})$ with $x_1, \ldots, x_{n-1} \le m$ and with arcs $(x_1, \ldots, x_{n-1}) \to (x_2, \ldots, x_n)$ whenever $\max(x_1, \ldots, x_n) = m$ is connected and balanced; see Theorem 2.3.4.2G. Indeed, we get such a sequence from Algorithm F if we note that the final k^n elements of the prime strings of length dividing n, when subtracted from $m-1$, are the same for all $m \ge k$. When $n = 4$, for example, the first 81 digits of the sequence Φ_4 are $2 - \alpha^R = 0\,0001\,01\,0011\ldots$, where α is the string (62). [There also are infinite m-ary sequences whose first m^n elements are de Bruijn cycles for all n, given any fixed $m \ge 3$. See L. J. Cummings and D. Wiedemann, *Cong. Numerantium* **53** (1986), 155–160.]

93. The cycle generated by $f()$ is a cyclic permutation of $\alpha 1$, where α has length $m^n - 1$ and ends with 1^{n-1}. The cycle generated by Algorithm R is a cyclic permutation of $\gamma = c_0 \ldots c_{m^{n+1}-1}$, where $c_k = (c_0 + b_0 + \cdots + b_{k-1}) \bmod m$ and $b_0 \ldots b_{m^{n+1}-1} = \beta = \alpha^m 1^m$.

If $x_0 \ldots x_n$ occurs in γ, say $x_j = c_{k+j}$ for $0 \le j \le n$, then $y_j = b_{k+j}$ for $0 \le j < n$, where $y_j = (x_{j+1} - x_j) \bmod m$. [This is the connection with modular m-ary Gray code; see exercise 78.] Now if $y_0 \ldots y_{n-1} = 1^n$ we have $m^{n+1} - m - n < k \le m^{n+1} - n$; otherwise there is an index k' such that $-n < k' < m^n - n$ and $y_0 \ldots y_{n-1}$ occurs in β at positions $k = (k' + r(m^n - 1)) \bmod m^{n+1}$ for $0 \le r < m$. In both cases the m choices of k have different values of x_0, because the sum of all elements in α is $m-1$

(modulo m) when $n \geq 2$. [Algorithm R is valid also for $n = 1$ if $m \bmod 4 \neq 2$, because $m \perp \sum \alpha$ in that case.]

94. 0<u>01</u>0<u>2</u>03041<u>12</u>1<u>3</u>142<u>2</u>3243344. (The underlined digits are effectively inserted into the interleaving of 00112234 with 34. Algorithm D can be used in general when $n = 1$ and $r = m - 2 \geq 0$; but it is pointless to do so, in view of (54).)

95. (a) Let $c_0 c_1 c_2 \ldots$ have period r. If r is odd we have $p = q = r$, so $r = pq$ only in the trivial case when $p = q = 1$ and $a_0 = b_0$. Otherwise $r/2 = \operatorname{lcm}(p, q) = pq/\gcd(p, q)$ by 4.5.2–(10), hence $\gcd(p, q) = 2$. In the latter case the $2n$-tuples $c_l c_{l+1} \ldots c_{l+2n-1}$ that occur are $a_j b_k \ldots a_{j+n-1} b_{k+n-1}$ for $0 \leq j < p$, $0 \leq k < q$, $j \equiv k$ (modulo 2), and $b_k a_j \ldots b_{k+n-1} a_{j+n-1}$ for $0 \leq j < p$, $0 \leq k < q$, $j \not\equiv k$ (modulo 2).

(b) The output would interleave two sequences $a_0 a_1 \ldots$ and $b_0 b_1 \ldots$ whose periods are respectively $m^n + r$ and $m^n - r$; the a's are the cycle of $f()$ with x^n changed to x^{n+1} and the b's are the cycle of $g()$ with x^n changed to x^{n-1}, for $0 \leq x < r$. By (58) and part (a), the period length is $m^{2n} - r^2$, and every $2n$-tuple occurs with the exception of $(xy)^n$ for $0 \leq x, y < r$.

(c) The real step D6 alters the behavior of (b) by going to D3 when $t \geq n$ and $0 \leq x' = x < r$; this emits an extra x at the time when x^{2n-1} has just been output and b is about to be emitted, where b is the digit following x^n in g's cycle. D6 also allows control to pass to D7 and then D3 with $t' = n$ in the case that $t \geq n$ and $x < x' < r$; this emits an extra $x'x$ at the time when $(xx')^{n-1}x$ has just been output and b will be next. These r^2 extra bits provide the r^2 missing $2n$-tuples of (b).

96. (a) The recurrences $S_2 = 1$, $S_{2n+1} = S_{2n} = 2S_n$, $R_2 = 0$, $R_{2n+1} = 1 + R_{2n}$, $R_{2n} = 2R_n$, $D_2 = 0$, $D_{2n+1} = D_{2n} = 1 + 2D_n$ have the solution $S_n = 2^{\lfloor \lg n \rfloor - 1}$, $R_n = n - 2S_n$, $D_n = S_n - 1$. Thus $S_n + R_n + D_n = n - 1$.

(b) Each top-level output usually involves $\lfloor \lg n \rfloor - 1$ D-activations and $\nu(n) - 1$ R-activations, plus one basic activation at the bottom level. But there are exceptions: Algorithm R might invoke its $f()$ twice, if the first activation completed a sequence 1^n; and sometimes Algorithm R doesn't need to invoke $f()$ at all. Algorithm D might invoke its $g()$ twice, if the first activation completed a sequence $(x')^n$; but sometimes Algorithm D doesn't need to invoke either $f()$ or $g()$.

Algorithm R completes a sequence x^{n+1} if and only if its child $f()$ has just completed a sequence 0^n. Algorithm D completes a sequence x^{2n} for $x < r$ if and only if it has just jumped from D6 to D3 without invoking any child.

From these observations we can conclude that at most $\lfloor \lg n \rfloor + \nu(n) + 1$ activations are possible per top-level output, if $r > 1$; such a case happens when Algorithm D for $n = 6$ goes from D6 to D4. But when $r = 1$ we can have as many as $2\lfloor \lg n \rfloor + 3$ activations, for example when Algorithm R for $n = 25$ goes from R4 to R2.

97. (a) (0011), (00011101), (0000101001111011), and (0000011000101101111 1001110101001). Thus $j_2 = 2$, $j_3 = 3$, $j_4 = 9$, $j_5 = 15$.

(b) We obviously have $f_{n+1}(k) = \Sigma f_n(k) \bmod 2$ for $0 \leq k < j_n + n$. The next value, $f_{n+1}(j_n + n)$, depends on whether step R4 jumps to R2 after computing $y = f_n(j_n + n - 1)$. If it does (namely, if $f_{n+1}(j_n + n - 1) \neq 0$), we have $f_{n+1}(k) \equiv 1 + \Sigma(k+1)$ for $j_n + n \leq k < 2^n + j_n + n$; otherwise we have $f_{n+1}(k) \equiv 1 + \Sigma(k-1)$ for those values of k. In particular, $f_{n+1}(k) = 1$ when $2^n \leq k + \delta_n \leq 2^n + n$. The stated formula, which has simpler ranges for the index k, holds because $1 + \Sigma(k \pm 1) \equiv \Sigma(k)$ when $j_n < k < j_n + n$ or $2^n + j_n < k < 2^n + j_n + n$.

(c) The interleaved cycle has $c_n(2k) = f_n^+(k)$ and $c_n(2k+1) = f_n^-(k)$, where

$$f_n^+(k) = \begin{cases} f_n(k-1), & \text{if } 0 < k \le j_n+1; \\ f_n(k-2), & \text{if } j_n+1 < k \le 2^n+2; \end{cases} \qquad f_n^-(k) = \begin{cases} f_n(k+1), & \text{if } 0 \le k < j_n; \\ f_n(k+2), & \text{if } j_n \le k < 2^n-2; \end{cases}$$

$f_n^+(k) = f_n^+\big(k \bmod (2^n+2)\big)$, $f_n^-(k) = f_n^-\big(k \bmod (2^n-2)\big)$. Therefore the subsequence 1^{2n-1} begins at position $k_n = (2^{n-1}-2)(2^n+2) + 2j_n + 2$ in the c_n cycle; this will make j_{2n} odd. The subsequence $(01)^{n-1}0$ begins at position $l_n = (2^{n-1}+1)(j_n-1)$ if $j_n \bmod 4 = 1$, at $l_n = (2^{n-1}+1)(2^n+j_n-3)$ if $j_n \bmod 4 = 3$. Also $k_2 = 6$, $l_2 = 2$.

(d) Algorithm D inserts four elements into the c_n cycle; hence

$$f_{2n}(k) = \begin{cases} c_n(k-1), & \text{if } 0 < k \le l_n+2; \\ c_n(k-3), & \text{if } l_n+2 < k \le k_n+3; \\ c_n(k-4), & \text{if } k_n+3 < k \le 2^{2n}; \end{cases} \quad \text{when } j_n \bmod 4 < 3 \ (l_n < k_n);$$

$$= \begin{cases} c_n(k-1), & \text{if } 0 < k \le k_n+1; \\ c_n(k-2), & \text{if } k_n+1 < k \le l_n+3; \\ c_n(k-4), & \text{if } l_n+3 < k \le 2^{2n}. \end{cases} \quad \text{when } j_n \bmod 4 = 3 \ (k_n < l_n);$$

(e) Consequently $j_{2n} = k_n + 1 + 2[j_n \bmod 4 < 3]$. Indeed, the elements preceding 1^{2n} consist of $2^{n-2}-1$ complete periods of $f_n^+()$ interleaved with 2^{n-2} complete periods of $f_n^-()$, with one 0 inserted and also with 10 inserted if $l_n < k_n$, followed by $f_n(1)f_n(1)f_n(2)f_n(2)\ldots f_n(j_n-1)f_n(j_n-1)$. The sum of all these elements is odd, unless $l_n < k_n$; therefore $\delta_{2n} = 1 - 2[j_n \bmod 4 = 3]$.

Let $n = 2^t q$, where q is odd and $n > 2$. The recurrences imply that, if $q = 1$, we have $j_n = 2^{n-1} + b_t$ where $b_t = 2^t/3 - (-1)^t/3$. And if $q > 1$ we have $j_n = 2^{n-1} \pm b_{t+2}$, where the $+$ sign is chosen if and only if $\lfloor \lg q \rfloor + [\lfloor 4q/2^{\lfloor \lg q \rfloor} \rfloor = 5]$ is even.

98. If $f(k) = g(k)$ when k lies in a certain range, there's a constant C such that $\Sigma f(k) = C + \Sigma g(k)$ for k in that range. We can therefore continue almost mindlessly to derive additional recurrences: If $n > 1$ we have

$$\Sigma f_{2n}(k), \text{ when } j_n \bmod 4 < 3 \ (l_n < k_n): \quad \equiv \begin{cases} \Sigma c_n(k-1), & \text{if } 0 < k \le l_n+2; \\ 1+\Sigma c_n(k-3), & \text{if } l_n+2 < k \le k_n+3; \\ \Sigma c_n(k-4), & \text{if } k_n+3 < k \le 2^{2n}; \end{cases}$$

$$\text{when } j_n \bmod 4 = 3 \ (k_n < l_n): \quad \equiv \begin{cases} \Sigma c_n(k-1), & \text{if } 0 < k \le k_n+1; \\ 1+\Sigma c_n(k-2), & \text{if } k_n+1 < k \le l_n+3; \\ \Sigma c_n(k-4), & \text{if } l_n+3 < k \le 2^{2n}. \end{cases}$$

$$\Sigma c_n(k) \equiv \Sigma f_n^+\big(\lceil k/2 \rceil\big) + \Sigma f_n^-\big(\lfloor k/2 \rfloor\big).$$

$$\Sigma f_n^+(k) \equiv \begin{cases} \Sigma f_n(k-1), & \text{if } 0 < k \le j_n+1; \\ 1+\Sigma f_n(k-2), & \text{if } j_n+1 < k \le 2^n+2; \end{cases} \qquad \Sigma f_n^-(k) \equiv \begin{cases} \Sigma f_n(k+1), & \text{if } 0 \le k < j_n; \\ 1+\Sigma f_n(k+2), & \text{if } j_n \le k < 2^n-2; \end{cases}$$

$$\Sigma f_n^\pm(k) \equiv \lfloor k/(2^n \pm 2) \rfloor + \Sigma f_n^\pm\big(k \bmod (2^n \pm 2)\big); \quad \Sigma f_n(k) = \Sigma f_n(k \bmod 2^n).$$

$$\Sigma f_{2n+1}(k) \equiv \begin{cases} \Sigma\Sigma f_{2n}(k), & \text{if } 0 < k \le j_{2n} \text{ or } 2^{2n}+j_{2n} < k \le 2^{2n+1}; \\ 1+k+\Sigma\Sigma f_{2n}(k+\delta_{2n}), & \text{if } j_{2n} < k \le 2^{2n}+j_{2n}. \end{cases}$$

$$\Sigma\Sigma f_{2n}(k), \text{ when } j_n \bmod 4 < 3 \ (l_n < k_n): \quad \equiv \begin{cases} \Sigma\Sigma c_n(k-1), & \text{if } 0 < k \le l_n+2; \\ 1+k+\Sigma\Sigma c_n(k-3), & \text{if } l_n+2 < k \le k_n+3; \\ \Sigma\Sigma c_n(k-4), & \text{if } k_n+3 < k \le 2^{2n}; \end{cases}$$

$$\text{when } j_n \bmod 4 = 3 \ (k_n < l_n): \quad \equiv \begin{cases} \Sigma\Sigma c_n(k-1), & \text{if } 0 < k \le k_n+1; \\ 1+k+\Sigma c_n(k-2), & \text{if } k_n+1 < k \le l_n+3; \\ 1+\Sigma\Sigma c_n(k-4), & \text{if } l_n+3 < k \le 2^{2n}. \end{cases}$$

$$\Sigma\Sigma f_{2n}(k) \equiv [j_n \bmod 4 < 3]\lfloor k/2^{2n} \rfloor + \Sigma\Sigma f_{2n}(k \bmod 2^{2n}).$$

And then, aha, there is closure:

$$\Sigma\Sigma c_n(2k) = \Sigma f_n^+(k), \qquad \Sigma\Sigma c_n(2k+1) = \Sigma f_n^-(k).$$

If $n = 2^t q$ where q is odd, the running time to evaluate $f_n(k)$ by this system of recursive formulas is $O(t + S(q))$, where $S(1) = 1$, $S(2k) = 1 + 2S(k)$, and $S(2k+1) = 1 + S(k)$. Clearly $S(k) < 2k$, so the evaluations involve at most $O(n)$ simple operations on n-bit numbers. In fact, the method is often significantly faster: If we average $S(k)$ over all k with $\lfloor \lg k \rfloor = s$ we get $(3^{s+1} - 2^{s+1})/2^s$, which is less than $3k^{\lg(3/2)} < 3k^{0.59}$. (Incidentally, if $k = 2^{s+1} - 1 - (2^{s-e_1} + 2^{s-e_2} + \cdots + 2^{s-e_t})$ we have $S(k) = s + 1 + e_t + 2e_{t-1} + 4e_{t-2} + \cdots + 2^t e_1$.)

99. A string that starts at position k in $f_n()$ starts at position $k^+ = k + 1 + [k > j_n]$ in $f_n^+()$ and at position $k^- = k - 1 - [k > j_n]$ in $f_n^-()$, except that 0^n and 1^n occur twice in $f_n^+()$ but not at all in $f_n^-()$.

To find $\gamma = a_0 b_0 \ldots a_{n-1} b_{n-1}$ in the cycle $f_{2n}()$, let $\alpha = a_0 \ldots a_{n-1}$ and $\beta = b_0 \ldots b_{n-1}$. Suppose α starts at position j and β at position k in $f_n()$, and assume that neither α nor β is 0^n or 1^n. If $j^+ \equiv k^+$ (modulo 2), let $l/2$ be a solution to the equation $j^+ + (2^n + 2)x = k^- + (2^n - 2)y$; we may take $l/2 = k + (2^n - 2)\big(2^{n-3}(j-k) \bmod (2^{n-1}+1)\big)$ if $j \geq k$, otherwise $l/2 = j + (2^n + 2)\big(2^{n-3}(k-j) \bmod (2^{n-1}-1)\big)$. Otherwise let $(l-1)/2 = k^+ + (2^n + 2)x = j^- + (2^n - 2)y$. Then γ starts at position l in the cycle $c_n()$; hence it starts at position $l + 1 + [l \geq k_n] + 2[l \geq l_n]$ in the cycle $f_{2n}()$. Similar formulas hold when $\alpha \in \{0^n, 1^n\}$ or $\beta \in \{0^n, 1^n\}$ (but not both). Finally, 0^{2n}, 1^{2n}, $(01)^n$, and $(10)^n$ start respectively in positions 0, j_{2n}, $l_n + 1 + [k_n < l_n]$, and $l_n + 2 + [k_n < l_n]$.

To find $\beta = b_0 b_1 \ldots b_n$ in $f_{n+1}()$ when n is even, suppose that the n-bit string $(b_0 \oplus b_1) \ldots (b_{n-1} \oplus b_n)$ starts at position j in $f_n()$. Then β starts at position $k = j - \delta_n[j \geq j_n] + 2^n[j = j_n][\delta_n = 1]$ if $f_{n+1}(k) = b_0$, otherwise at position $k + (2^n - \delta_n, \delta_n, 2^n + \delta_n)$ according as $(j < j_n, j = j_n, j > j_n)$.

The running time of this recursion satisfies $T(n) = O(n) + 2T\big(\lfloor n/2 \rfloor\big)$, so it is $O(n \log n)$. [Exercises 97–99 are based on the work of J. Tuliani, who also has developed methods for certain larger values of m; see *Discrete Math.* **226** (2001), 313–336.]

100. No obvious defects are apparent, but extensive testing should be done before any sequence can be recommended. By contrast, the de Bruijn cycle produced implicitly by Algorithm F is a terrible source of supposedly random bits, even though it is n-distributed in the sense of Definition 3.5D, because 0s predominate at the beginning. Indeed, when n is prime, bits $tn + 1$ of that sequence are zero for $0 \leq t < (2^n - 2)/n$.

101. (a) Let β be a proper suffix of $\lambda\lambda'$ with $\beta \leq \lambda\lambda'$. Either β is a suffix of λ', whence $\lambda < \lambda' \leq \beta$, or $\beta = \alpha\lambda'$ and we have $\lambda < \alpha < \beta$.

Now $\lambda < \beta \leq \lambda\lambda'$ implies that $\beta = \lambda\gamma$ for some $\gamma \leq \lambda'$. But γ is a suffix of β with $1 \leq |\gamma| = |\beta| - |\lambda| < |\lambda'|$; hence γ is a proper suffix of λ', and $\lambda' < \gamma$. Contradiction.

(b) Any string of length 1 is prime. Combine adjacent primes by (a), in any order, until no further combination is possible. [See the more general results of M. P. Schützenberger, *Proc. Amer. Math. Soc.* **16** (1965), 21–24.]

(c) If $t \neq 0$, let λ be the smallest suffix of $\lambda_1 \ldots \lambda_t$. Then λ is prime by definition, and it has the form $\beta\gamma$ where β is a nonempty suffix of some λ_j. Therefore $\lambda_t \leq \lambda_j \leq \beta \leq \beta\gamma = \lambda \leq \lambda_t$, so we must have $\lambda = \lambda_t$. Remove λ_t and repeat until $t = 0$.

(d) True. For if we had $\alpha = \lambda\beta$ for some prime λ with $|\lambda| > |\lambda_1|$, we could append the factors of β to obtain another factorization of α.

(e) $3 \cdot 14159265358979323846264338327 95 \cdot 02884197$. (Knowing more digits of π would not change the first two factors. The infinite decimal expansion of any number that is "normal" in the sense of Borel (see Section 3.5) factors into primes of finite length.)

102. We must have $1/(1-mz) = 1/\prod_{n=1}^{\infty}(1-z^n)^{L_m(n)}$. This implies (6o) as in exercise 4.6.2–4.

103. When $n = p$ is prime, (59) tells us that $L_m(1) + pL_m(p) = m^p$, and we also have $L_m(1) = m$. [This combinatorial proof provides an interesting contrast to the traditional algebraic proof of Theorem 1.2.4F.]

104. The 4483 nonprimes are `abaca`, `agora`, `ahead`, ...; the 1274 primes are ..., `rusts`, `rusty`, `rutty`. (Since `prime` isn't prime, we should perhaps call prime strings `lowly`.)

105. (a) Let α' be α with its last letter increased, and suppose $\alpha' = \beta\gamma'$ where $\alpha = \beta\gamma$ and $\beta \neq \epsilon$, $\gamma \neq \epsilon$. Let θ be the prefix of α with $|\theta| = |\gamma|$. By hypothesis there is a string ω such that $\alpha\omega$ is prime; hence $\theta \le \alpha\omega < \gamma\omega$, so we must have $\theta \le \gamma$. Consequently $\theta < \gamma'$, and we have $\alpha' < \gamma'$.

(b) Let $\alpha = \lambda_1\beta = a_1 \ldots a_n$ where $\lambda_1\beta\omega$ is prime. The condition $\lambda_1\beta\omega < \beta\omega$ implies that $a_j \le a_{j+r}$ for $1 \le j \le n-r$, where $r = |\lambda_1|$. But we cannot have $a_j < a_{j+r}$; otherwise α would begin with a prime longer than λ_1, contradicting exercise 101(d).

(c) If α is the n-extension of both λ and λ', where $|\lambda| > |\lambda'|$, we must have $\lambda = (\lambda')^q\theta$ where θ is a proper prefix of λ'. But then $\theta < \lambda' < \lambda < \theta$.

106. **B1.** [Initialize.] Set $a_1 \leftarrow \cdots \leftarrow a_n \leftarrow m-1$, $a_{n+1} \leftarrow -1$, and $j \leftarrow 1$.

B2. [Visit.] Visit $(a_1, \ldots, a_n)$ with index j.

B3. [Subtract one.] Terminate if $a_j = 0$. Otherwise set $a_j \leftarrow a_j - 1$, and $a_k \leftarrow m-1$ for $j < k \le n$.

B4. [Prepare to factor.] (According to exercise 105(b), we now want to find the first prime factor λ_1 of $a_1 \ldots a_n$.) Set $j \leftarrow 1$ and $k \leftarrow 2$.

B5. [Find the new j.] (Now $a_1 \ldots a_{k-1}$ is the $(k-1)$-extension of the prime $a_1 \ldots a_j$.) If $a_{k-j} > a_k$, return to B2. Otherwise, if $a_{k-j} < a_k$, set $j \leftarrow k$. Then increase k by 1 and repeat this step. ∎

The efficient factoring algorithm in steps B4 and B5 is due to J. P. Duval, *J. Algorithms* **4** (1983), 363–381. For further information, see Cattell, Ruskey, Sawada, Serra, and Miers, *J. Algorithms* **37** (2000), 267–282.

107. The number of n-tuples visited is $P_m(n) = \sum_{j=1}^{n} L_m(j)$. Since $L_m(n) = \frac{1}{n}m^n + O(m^{n/2}/n)$, we have $P_m(n) = Q(m,n) + O\big(Q(\sqrt{m}, n)\big)$, where

$$Q(m,n) = \sum_{k=1}^{n} \frac{m^k}{k} = \frac{m^n}{n}\, R(m,n);$$

$$R(m,n) = \sum_{k=0}^{n-1} \frac{m^{-k}}{1-k/n} = \sum_{k=0}^{n/2} \frac{m^{-k}}{1-k/n} + O(nm^{-n/2})$$

$$= \frac{m}{m-1} \sum_{j=0}^{t-1} \frac{1}{n^j} \sum_{l} \left\langle {j \atop l} \right\rangle \frac{m^l}{(m-1)^j} + O(n^{-t}).$$

The main contributions to the running time come from the loops in steps F3 and F5, which cost $n-j$ for each prime of length j, hence a total of $nP(n) - \sum_{j=1}^{n} jL_m(j) = m^{n+1}/((m-1)^2 n) + O(1/(mn^2))$. This is less than the time needed to output the m^n individual digits of the de Bruijn cycle.

108. (a) If $\alpha \neq 9 \ldots 9$, we have $\lambda_{k+1} \le \beta 9^{|\alpha|}$, because the latter is prime.

(b) We can assume that β is not all 0s, since 9^j0^{n-j} is a substring of $\lambda_{t-1}\lambda_t\lambda_1\lambda_2 = 89^n0^n1$. Let k be maximal with $\beta \le \lambda_k$; then $\lambda_k \le \beta\alpha$, so β is a prefix of λ_k. Since β is a preprime, it is the $|\beta|$-extension of some prime $\beta' \le \beta$. The preprime visited by Algorithm F just before β' is $(\beta'-1)9^{n-|\beta'|}$, by exercise 106, where $\beta'-1$ denotes the decimal number that is one less than β'. Thus, if β' is not λ_{k-1}, the hint (which also follows from exercise 106) implies that λ_{k-1} ends with at least $n-|\beta'| \ge n-|\beta|$ 9s, and α is a suffix of λ_{k-1}. On the other hand if $\beta' = \lambda_{k-1}$, α is a suffix of λ_{k-2}, and β is a prefix of $\lambda_{k-1}\lambda_k$.

(c) If $\alpha \ne 9\ldots9$, we have $\lambda_{k+1} \le (\beta\alpha)^{d-1}\beta9^{|\alpha|}$, because the latter is prime. Otherwise λ_{k-1} ends with at least $(d-1)|\beta\alpha|$ 9s, and $\lambda_{k+1} \le (\beta\alpha)^{d-1}9^{|\beta\alpha|}$, so $(\alpha\beta)^d$ is a substring of $\lambda_{k-1}\lambda_k\lambda_{k+1}$.

(d) Within the primes 135899 135914, 787899 787979, 129999 13 131314, 09 090911, 089999 09 090911, 089999 119 119122.

[In all cases, the position of $a_1\ldots a_n$ precedes the position of $a_1\ldots a_{n-1}(a_n+1)$, if $0 \le a_n < 9$ (and if we assume that strings like 9^j0^{n-j} occur at the beginning). Furthermore 9^j0^{n-j-1} occurs only after $9^{j-1}0^{n-j}a$ has appeared for $1 \le a \le 9$, so we must not place 0 after 9^j0^{n-j-1}. Therefore no m-ary de Bruijn cycle of length m^n can be lexicographically smaller than $\lambda_1\ldots\lambda_t$.]

109. Suppose we want to locate the submatrix

$$\begin{pmatrix} (w_{n-1}\ldots w_1w_0)_2 & (x_{n-1}\ldots x_1x_0)_2 \\ (y_{n-1}\ldots y_1y_0)_2 & (z_{n-1}\ldots z_1z_0)_2 \end{pmatrix}.$$

The binary case $n=1$ is the given example, and if $n>1$ we can assume by induction that we only need to determine the leading bits a_{2n-1}, a_{2n-2}, b_{2n-1}, and b_{2n-2}. The case $n=3$ is typical: We must solve

$$\begin{aligned}
b_5 &= w_2, & b_4 &= x_2, & a_5\oplus b_5 &= y_2, & a_4\oplus b_4 &= z_2, & &\text{if } a_0=0,\ b_0=0;\\
b_4 &= w_2, & b'_5 &= x_2, & a_4\oplus b_4 &= y_2, & a_5\oplus b'_5 &= z_2, & &\text{if } a_0=0,\ b_0=1;\\
a_5\oplus b_5 &= w_2, & a_4\oplus b_4 &= x_2, & b_5 &= y_2, & b_4 &= z_2, & &\text{if } a_0=1,\ b_0=0;\\
a_4\oplus b_4 &= w_2, & a_5\oplus b'_5 &= x_2, & b_4 &= y_2, & b'_5 &= z_2, & &\text{if } a_0=1,\ b_0=1;
\end{aligned}$$

here $b'_5 = b_5 \oplus b_4b_3b_2b_1$ takes account of carrying when j becomes $j+1$.

110. Let $a_0a_1\ldots a_{m^2-1}$ be an m-ary de Bruijn cycle, such as the first m^2 elements of (54). If m is odd, let $a_{ij} = a_j$ when i is even, $a_{ij} = a_{(j+(i-1)/2) \bmod m^2}$ when i is odd. [The first of many people to discover this construction seems to have been John C. Cock, who also constructed de Bruijn toruses of other shapes and sizes in *Discrete Math.* **70** (1988), 209–210.]

If $m = m'm''$ where $m' \perp m''$, we use the Chinese remainder theorem to define

$$a_{ij} \equiv a'_{ij} \text{ (modulo } m') \qquad \text{and} \qquad a_{ij} \equiv a''_{ij} \text{ (modulo } m'')$$

in terms of matrices that solve the problem for m' and m''. Thus the previous exercise leads to a solution for arbitrary m.

Another interesting solution for even values of m was found by Zoltán Tóth [*2nd Conf. Automata, Languages, and Programming Systems* (1988), 165–172; see also Hurlbert and Isaak, *Contemp. Math.* **178** (1994), 153–160]. The first m^2 elements a_j of the infinite sequence

0011 021331203223 041524355342514054450617263746577564 . . . 07667 08 . . .

define a de Bruijn cycle with the property that the distance between the appearances of ab and ba is always even. Then we can let $a_{ij} = a_j$ if $i + j$ is even, $a_{ij} = a_i$ if $i + j$ is odd. For example, when $m = 4$ we have

$$\begin{pmatrix}
0&0&1&0&0&2&1&2&2&0&3&0&2&2&3&2\\
0&0&0&1&0&2&0&3&2&0&2&1&2&2&2&3\\
0&1&1&1&0&3&1&3&2&1&3&1&2&3&3&3\\
1&0&1&1&1&2&1&3&3&0&3&1&3&2&3&3\\
0&0&1&0&0&2&1&2&2&0&3&0&2&2&3&2\\
0&2&0&3&0&0&0&1&2&2&2&3&2&0&2&1\\
0&1&1&1&0&3&1&3&2&1&3&1&2&3&3&3\\
1&2&1&3&1&0&1&1&3&2&3&3&3&0&3&1\\
0&0&1&0&0&2&1&2&2&0&3&0&2&2&3&2\\
2&0&2&1&2&2&2&3&0&0&0&1&0&2&0&3\\
0&1&1&1&0&3&1&3&2&1&3&1&2&3&3&3\\
3&0&3&1&3&2&3&3&1&0&1&1&1&2&1&3\\
0&0&1&0&0&2&1&2&2&0&3&0&2&2&3&2\\
2&2&2&3&2&0&2&1&0&2&0&3&0&0&0&1\\
0&1&1&1&0&3&1&3&2&1&3&1&2&3&3&3\\
3&2&3&3&3&0&3&1&1&2&1&3&1&0&1&1
\end{pmatrix}$$ (exercise 109);

$$\begin{pmatrix}
0&0&1&0&0&0&1&0&3&0&2&0&3&0&2&0\\
0&0&0&1&0&2&0&3&0&1&0&0&0&2&0&3\\
0&1&1&1&0&1&1&1&3&1&2&1&3&1&2&1\\
1&0&1&1&1&2&1&3&1&1&1&0&1&2&1&3\\
0&0&1&0&0&0&1&0&3&0&2&0&3&0&2&0\\
2&0&2&1&2&2&2&3&2&1&2&0&2&2&2&3\\
0&1&1&1&0&1&1&1&3&1&2&1&3&1&2&1\\
3&0&3&1&3&2&3&3&3&1&3&0&3&2&3&3\\
0&3&1&3&0&3&1&3&3&3&2&3&3&3&2&3\\
1&0&1&1&1&2&1&3&1&1&1&0&1&2&1&3\\
0&2&1&2&0&2&1&2&3&2&2&2&3&2&2&2\\
0&0&0&1&0&2&0&3&0&1&0&0&0&2&0&3\\
0&3&1&3&0&3&1&3&3&3&2&3&3&3&2&3\\
2&0&2&1&2&2&2&3&2&1&2&0&2&2&2&3\\
0&2&1&2&0&2&1&2&3&2&2&2&3&2&2&2\\
3&0&3&1&3&2&3&3&3&1&3&0&3&2&3&3
\end{pmatrix}$$ (Tóth).

111. (a) Let $d_j = j$ and $0 \le a_j < 3$ for $1 \le j \le 9$, $a_9 \ne 0$. Form sequences s_j, t_j by the rules $s_1 = 0$, $t_1 = d_1$; $t_{j+1} = d_{j+1} + 10t_j[a_j = 0]$ for $1 \le j < 9$; $s_{j+1} = s_j + (0, t_j, -t_j)$ for $a_j = (0, 1, 2)$ and $1 \le j \le 9$. Then s_{10} is a possible result; we need only remember the smallish values that occur. More than half the work is saved by disallowing $a_k = 2$ when $s_k = 0$, then using $|s_{10}|$ instead of s_{10}. Since fewer than $3^8 = 6561$ possibilities need to be tried, brute force via the ternary version of Algorithm M works well; fewer than 24,000 mems and 1600 multiplications are needed to deduce that all integers less than 211 are representable, but 211 is not.

Another approach, using Gray code to vary the signs after breaking the digits into blocks in 2^8 possible ways, reduces the number of multiplications to 255, but at the cost of about 500 additional mems. Therefore Gray code is not advantageous in this application.

(b) Now (with 73,000 mems and 4900 multiplications) we can reach all numbers less than 241, but not 241. There are 46 ways to represent 100, including the remarkable $9 - 87 + 6 + 5 - 43 + 210$.

[H. E. Dudeney introduced his "century" problem in *The Weekly Dispatch* (4 and 18 June 1899); see also *The Numerology of Dr. Matrix* by Martin Gardner, Chapter 6.]

112. The method of exercise 111 now needs more than 167 million mems and 10 million multiplications, because 3^{16} is so much larger than 3^8. We can do much better (10.4 million mems, 1100 mults) by first tabulating the possibilities obtainable from the first k and last k digits, for $1 \le k < 9$, then considering all blocks of digits that use the 9. There are 60,318 ways to represent 100, and the first unreachable number is 16,040.

INDEX AND GLOSSARY

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information. An answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.

THE ART OF COMPUTER PROGRAMMING

PRE-FASCICLE 2B

A DRAFT OF SECTION 7.2.1.2: GENERATING ALL PERMUTATIONS

DONALD E. KNUTH *Stanford University*

ADDISON–WESLEY

Internet page `http://www-cs-faculty.stanford.edu/~knuth/taocp.html` contains current information about this book and related books.

See also `http://www-cs-faculty.stanford.edu/~knuth/sgb.html` for information about *The Stanford GraphBase*, including downloadable software for dealing with the graphs used in many of the examples in Chapter 7.

See also `http://www-cs-faculty.stanford.edu/~knuth/mmixware.html` for downloadable software to simulate the `MMIX` computer.

Zeroth printing (revision 5), 30 March 2002

PREFACE

I thought it worth a Dayes labour,
to write something on this Art or Science,
that the Rules thereof might not be lost and obscured.
— RICHARD DUCKWORTH, *Tintinnalogia* (1668)

THIS BOOKLET contains draft material that I'm circulating to experts in the field, in hopes that they can help remove its most egregious errors before too many other people see it. I am also, however, posting it on the Internet for courageous and/or random readers who don't mind the risk of reading a few pages that have not yet reached a very mature state. *Beware:* This material has not yet been proofread as thoroughly as the manuscripts of Volumes 1, 2, and 3 were at the time of their first printings. Those volumes, alas, were subsequently found to contain thousands of mistakes.

Given this caveat, I hope that my errors this time will not be so numerous and/or obtrusive that you will be discouraged from reading the material carefully. I did try to make it both interesting and authoritative, as far as it goes. But the field is so vast, I cannot hope to have surrounded it enough to corral it completely. Therefore I beg you to let me know about any deficiencies you discover.

To put the material in context, this is Section 7.2.1.2 of a long, long chapter on combinatorial algorithms. Chapter 7 will eventually fill three volumes (namely Volumes 4A, 4B, and 4C), assuming that I'm able to remain healthy. It will begin with a short review of graph theory, with emphasis on some highlights of significant graphs in The Stanford GraphBase (from which I will be drawing many examples). Then comes Section 7.1, which deals with the topic of bitwise manipulations. (I drafted about 60 pages about that subject in 1977, but those pages need extensive revision; meanwhile I've decided to work for awhile on the material that follows it, so that I can get a better feel for how much to cut.) Section 7.2 is about generating all possibilities, and it begins with Section 7.2.1: Generating Basic Combinatorial Patterns—which, in turn, begins with Section 7.2.1.1, "Generating all n-tuples." (Readers of the present booklet should have already looked at Section 7.2.1.1, a draft of which is available as Prefascicle 2A.) That sets the stage for the main contents of this booklet, Section 7.2.1.2: "Generating all permutations." Then will come Section 7.2.1.3 (about combinations), etc. Section 7.2.2 will deal with backtracking in general. And so it will go on, if all goes well; an outline of the entire Chapter 7 as currently envisaged appears on the `taocp` webpage that is cited on page ii.

Even the apparently lowly topic of permutation generation turns out to be surprisingly rich, with ties to Sections 1.2.9, 1.3.3, 2.2.3, 2.3.4.2, 3.4.2, 4.1, 5.1.1, 5.1.2, 5.1.4, 5.2.1, 5.2.2, 5.3.1, and 6.1 of the first three volumes. There also is material related to the `MMIX` computer, defined in Section 1.3.1′ of Fascicle 1. I strongly believe in building up a firm foundation, so I have discussed this topic much more thoroughly than I will be able to do with material that is newer or less basic. To my surprise, I came up with 111 exercises, even though — believe it or not — I had to eliminate quite a bit of the interesting material that appears in my files.

Some of the things presented are new, to the best of my knowledge, although I will not be at all surprised to learn that my own little "discoveries" have been discovered before. Please look, for example, at the exercises that I've classed as research problems (rated with difficulty level 46 or higher), namely exercises 108 and 111; I've also implicitly posed additional unsolved questions in the answers to exercises 28, 58, 63, 67, 88, 99, 105, and 111. Are those problems still open? Please let me know if you know of a solution to any of these intriguing questions. And of course if no solution is known today but you do make progress on any of them in the future, I hope you'll let me know.

I urgently need your help also with respect to some exercises that I made up as I was preparing this material. I certainly don't like to get credit for things that have already been published by others, and most of these results are quite natural "fruits" that were just waiting to be "plucked." Therefore please tell me if you know who I should have credited, with respect to the ideas found in exercises 6, 7, 20, 25, 41, 55, 60, 65, 66, 67, 69, 70, 75, 88, 98, and/or 103.

I shall happily pay a finder's fee of \$2.56 for each error in this draft when it is first reported to me, whether that error be typographical, technical, or historical. The same reward holds for items that I forgot to put in the index. And valuable suggestions for improvements to the text are worth 32¢ each. (Furthermore, if you find a better solution to an exercise, I'll actually reward you with immortal glory instead of mere money, by publishing your name in the eventual book:–)

Happy reading!

Stanford, California D. E. K.
31 December 2001

Tin tan din dan bim bam bom bo —
tan tin din dan bam bim bo bom —
tin tan dan din bim bam bom bo —
tan tin dan din bam bim bo bom —
tan dan tin bam din bo bim bom —
. . . . Tin tan din dan bim bam bom bo.

— DOROTHY L. SAYERS, *The Nine Tailors* (1934)

A permutation on the ten decimal digits is simply a 10 digit decimal number in which all digits are distinct. Hence all we need to do is to produce all 10 digit numbers and select only those whose digits are distinct. Isn't it wonderful how high speed computing saves us from the drudgery of thinking! We simply program $k + 1 \rightarrow k$ *and examine the digits of* k *for undesirable equalities. This gives us the permutations in dictionary order too! On second sober thought . . . we do need to think of something else.*

— D. H. LEHMER (1957)

7.2.1.2. Generating all permutations. After n-tuples, the next most important item on nearly everybody's wish list for combinatorial generation is the task of visiting all *permutations* of some given set or multiset. Many different ways have been devised to solve this problem. In fact, almost as many different algorithms have been published for unsorting as for sorting! We will study the most important permutation generators in this section, beginning with a classical method that is both simple and flexible:

Algorithm L (*Lexicographic permutation generation*). Given a sequence of n elements $a_1 a_2 \dots a_n$, initially sorted so that

$$a_1 \le a_2 \le \cdots \le a_n, \tag{1}$$

this algorithm generates all permutations of $\{a_1, a_2, \dots, a_n\}$, visiting them in lexicographic order. (For example, the permutations of $\{1, 2, 2, 3\}$ are

1223, 1232, 1322, 2123, 2132, 2213, 2231, 2312, 2321, 3122, 3212, 3221,

ordered lexicographically.) An auxiliary element a_0 is assumed to be present for convenience; a_0 must be strictly less than the largest element a_n.

L1. [Visit.] Visit the permutation $a_1 a_2 \dots a_n$.

L2. [Find j.] Set $j \leftarrow n-1$. If $a_j \geq a_{j+1}$, decrease j by 1 repeatedly until $a_j < a_{j+1}$. Terminate the algorithm if $j = 0$. (At this point j is the largest subscript such that we have already visited all permutations beginning with $a_1 \ldots a_j$. Therefore the lexicographically next permutation will increase the value of a_j.)

L3. [Increase a_j.] Set $l \leftarrow n$. If $a_j \geq a_l$, decrease l by 1 repeatedly until $a_j < a_l$. Then interchange $a_j \leftrightarrow a_l$. (Since $a_{j+1} \geq \cdots \geq a_n$, element a_l is the smallest element greater than a_j that can legitimately follow $a_1 \ldots a_{j-1}$ in a permutation. Before the interchange we had $a_{j+1} \geq \cdots \geq a_{l-1} \geq a_l > a_j \geq a_{l+1} \geq \cdots \geq a_n$; after the interchange, we have $a_{j+1} \geq \cdots \geq a_{l-1} \geq a_j > a_l \geq a_{l+1} \geq \cdots \geq a_n$.)

L4. [Reverse $a_{j+1} \ldots a_n$.] Set $k \leftarrow j+1$ and $l \leftarrow n$. Then, if $k < l$, interchange $a_k \leftrightarrow a_l$, set $k \leftarrow k+1$, $l \leftarrow l-1$, and repeat until $k \geq l$. Return to L1. ∎

This algorithm goes back at least to the 18th century, in C. F. Hindenburg's preface to *Specimen Analyticum de Lineis Curvis Secundi Ordinis* by C. F. Rüdiger (Leipzig: 1784), xlvi–xlvii, and it has been frequently rediscovered ever since. The parenthetical remarks in steps L2 and L3 explain why it works.

In general, the lexicographic successor of any combinatorial pattern $a_1 \ldots a_n$ is obtainable by a three-step procedure:

1) Find the largest j such that a_j can be increased.
2) Increase a_j by the smallest feasible amount.
3) Find the lexicographically least way to extend the new $a_1 \ldots a_j$ to a complete pattern.

Algorithm L follows this general procedure in the case of permutation generation, just as Algorithm 7.2.1.1M followed it in the case of n-tuple generation; we will see numerous further instances later, as we consider other kinds of combinatorial patterns. Notice that we have $a_{j+1} \geq \cdots \geq a_n$ at the beginning of step L4. Therefore the first permutation beginning with the current prefix $a_1 \ldots a_j$ is $a_1 \ldots a_j a_n \ldots a_{j+1}$, and step L4 produces it by doing $\lfloor (n-j)/2 \rfloor$ interchanges.

In practice, step L2 finds $j = n-1$ half of the time when the elements are distinct, because exactly $n!/2$ of the $n!$ permutations have $a_{n-1} < a_n$. Therefore Algorithm L can be speeded up by recognizing this special case, without making it significantly more complicated. (See exercise 1.) Similarly, the probability that $j \leq n-t$ is only $1/t!$ when the a's are distinct; hence the loops in steps L2–L4 usually go very fast. Exercise 6 analyzes the running time in general, showing that Algorithm L is reasonably efficient even when equal elements are present, unless some values appear much more often than others do in the multiset $\{a_1, a_2, \ldots, a_n\}$.

Adjacent interchanges. We saw in Section 7.2.1.1 that Gray codes are advantageous for generating n-tuples, and similar considerations apply when we want to generate permutations. The simplest possible change to a permutation is to interchange adjacent elements, and we know from Chapter 5 that any permutation can be sorted into order if we make a suitable sequence of such

interchanges. (For example, Algorithm 5.2.2B works in this way.) Hence we can go backward and obtain any desired permutation, by starting with all elements in order and then exchanging appropriate pairs of adjacent elements.

A natural question now arises: Is it possible to run through *all* permutations of a given multiset in such a way that only two adjacent elements change places at every step? If so, the overall program that is examining all permutations will often be simpler and faster, because it will only need to calculate the effect of an exchange instead of to reprocess an entirely new array $a_1 \ldots a_n$ each time.

Alas, when the multiset has repeated elements, we can't always find such a Gray-like sequence. For example, the six permutations of $\{1, 1, 2, 2\}$ are connected to each other in the following way by adjacent interchanges:

$$1122 \text{ —— } 1212 \begin{array}{l} \diagup \\ \diagdown \end{array} \begin{array}{c} 2112 \\ 1221 \end{array} \begin{array}{l} \diagdown \\ \diagup \end{array} 2121 \text{ —— } 2211; \tag{2}$$

this graph has no Hamiltonian path.

But most applications deal with permutations of *distinct* elements, and for this case there is good news: A simple algorithm makes it possible to generate all $n!$ permutations by making just $n! - 1$ adjacent interchanges. Furthermore, another such interchange returns to the starting point, so we have a Hamiltonian circuit analogous to Gray binary code.

The idea is to take such a sequence for $\{1, \ldots, n-1\}$ and to insert the number n into each permutation in all ways. For example, if $n = 4$ the sequence $(123, 132, 312, 321, 231, 213)$ leads to the columns of the array

$$\begin{array}{cccccc} 1234 & 1324 & 3124 & 3214 & 2314 & 2134 \\ 1243 & 1342 & 3142 & 3241 & 2341 & 2143 \\ 1423 & 1432 & 3412 & 3421 & 2431 & 2413 \\ 4123 & 4132 & 4312 & 4321 & 4231 & 4213 \end{array} \tag{3}$$

when 4 is inserted in all four possible positions. Now we obtain the desired sequence by reading downwards in the first column, upwards in the second, downwards in the third, ..., upwards in the last: $(1234, 1243, 1423, 4123, 4132, 1432, 1342, 1324, 3124, 3142, \ldots, 2143, 2134)$.

In Section 5.1.1 we studied the inversions of a permutation, namely the pairs of elements (not necessarily adjacent) that are out of order. Every interchange of adjacent elements changes the total number of inversions by ± 1. In fact, when we consider the so-called inversion table $c_1 \ldots c_n$ of exercise 5.1.1–7, where c_j is the number of elements lying to the right of j that are less than j, we find that the permutations in (3) have the following inversion tables:

$$\begin{array}{cccccc} 0000 & 0010 & 0020 & 0120 & 0110 & 0100 \\ 0001 & 0011 & 0021 & 0121 & 0111 & 0101 \\ 0002 & 0012 & 0022 & 0122 & 0112 & 0102 \\ 0003 & 0013 & 0023 & 0123 & 0113 & 0103 \end{array} \tag{4}$$

And if we read these columns alternately down and up as before, we obtain precisely the reflected Gray code for mixed radices $(1, 2, 3, 4)$, as in Eqs. (46)–(51)

of Section 7.2.1.1. The same property holds for all n, as noticed by E. W. Dijkstra [*Acta Informatica* **6** (1976), 357–359], and it leads us to the following formulation:

Algorithm P (*Plain changes*). Given a sequence $a_1a_2\ldots a_n$ of n distinct elements, this algorithm generates all of their permutations by repeatedly interchanging adjacent pairs. It uses an auxiliary array $c_1c_2\ldots c_n$, which represents inversions as described above, running through all sequences of integers such that

$$0 \le c_j < j \qquad \text{for } 1 \le j \le n. \tag{5}$$

Another array $d_1d_2\ldots d_n$ governs the directions by which the entries c_j change.

P1. [Initialize.] Set $c_j \leftarrow 0$ and $d_j \leftarrow 1$ for $1 \le j \le n$.

P2. [Visit.] Visit the permutation $a_1a_2\ldots a_n$.

P3. [Prepare for change.] Set $j \leftarrow n$ and $s \leftarrow 0$. (The following steps determine the coordinate j for which c_j is about to change, preserving (5); variable s is the number of indices $k > j$ such that $c_k = k - 1$.)

P4. [Ready to change?] Set $q \leftarrow c_j + d_j$. If $q < 0$, go to P7; if $q = j$, go to P6.

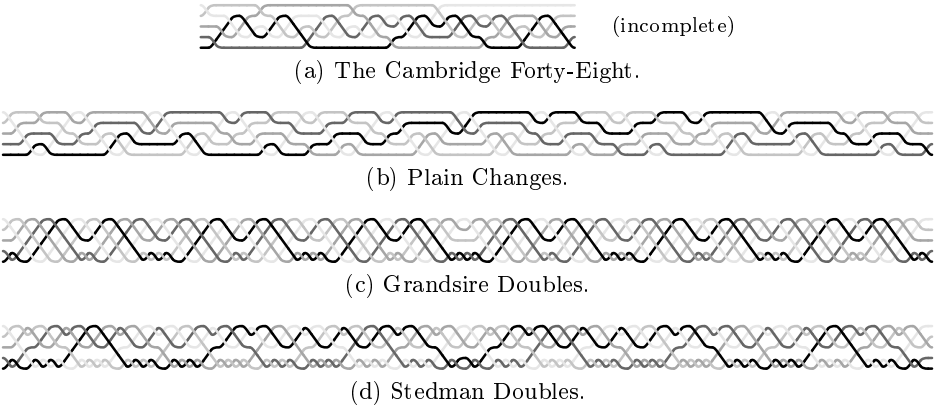
P5. [Change.] Interchange $a_{j-c_j+s} \leftrightarrow a_{j-q+s}$. Then set $c_j \leftarrow q$ and return to P2.

P6. [Increase s.] Terminate if $j = 1$; otherwise set $s \leftarrow s + 1$.

P7. [Switch direction.] Set $d_j \leftarrow -d_j$, $j \leftarrow j - 1$, and go back to P4. ∎

This procedure, which clearly works for all $n \ge 1$, originated in 17th-century England, when bell ringers began the delightful custom of ringing a set of bells in all possible permutations. They called Algorithm P the method of *plain changes*. Figure 18(a) illustrates the "Cambridge Forty-Eight," an irregular and ad hoc sequence of 48 permutations on 5 bells that had been used in the early 1600s, before the plain-change principle revealed how to achieve all $5! = 120$ possibilities. The venerable history of Algorithm P has been traced to a manuscript by Peter Mundy now in the Bodleian Library, written about 1653 and transcribed by Ernest Morris in *The History and Art of Change Ringing* (1931), 29–30. Shortly afterwards, a famous book called *Tintinnalogia*, published anonymously in 1668 but now known to have been written by Richard Duckworth and Fabian Stedman, devoted its first 60 pages to a detailed description of plain changes, working up from $n = 3$ to the case of arbitrarily large n.

> Cambridge Forty-eight, *for many years,*
> *was the greatest* Peal *that was* Rang *or invented; but now,*
> *neither* Forty-eight, *nor a* Hundred, *nor* Seven-hundred and twenty,
> *nor any Number can confine us; for we can* Ring Changes, Ad infinitum.
> . . . *On four Bells, there are* Twenty four several Changes,
> *in* Ringing *of which, there is one Bell called the* Hunt,
> *and the other three are* Extream *Bells;*
> *the* Hunt *moves, and* hunts *up and down continually . . . ;*
> *two of the* Extream *Bells makes a* Change
> *every time the* Hunt *comes before or behind them.*
>
> — DUCKWORTH and STEDMAN (1668)



(a) The Cambridge Forty-Eight.

(b) Plain Changes.

(c) Grandsire Doubles.

(d) Stedman Doubles.

Fig. 18. Four patterns used to ring church-bells in 17th-century England. Pattern (b) corresponds to Algorithm P.

British bellringing enthusiasts soon went on to develop more complicated schemes in which two or more pairs of bells change places simultaneously. For example, they devised the pattern in Fig. 18(c) known as Grandsire Doubles, "the best and most ingenious Peal that ever was composed, to be rang on five bells" [*Tintinnalogia*, page 95]. Such fancier methods are more interesting than Algorithm P from a musical or mathematical standpoint, but they are less useful in computer applications, so we shall not dwell on them here. Interested readers can learn more by reading W. G. Wilson's book, *Change Ringing* (1965); see also A. T. White, *AMM* **103** (1996), 771–778.

H. F. Trotter published the first computer implementation of plain changes in *CACM* **5** (1962), 434–435. The algorithm is quite efficient, especially when it is streamlined as in exercise 16, because $n-1$ out of every n permutations are generated without using steps P6 and P7. By contrast, Algorithm L enjoys its best case only about half of the time.

The fact that Algorithm P does exactly one interchange per visit means that the permutations it generates are alternately even and odd (see exercise 5.1.1–13). Therefore we can generate all the even permutations by simply bypassing the odd ones. In fact, the c and d tables make it easy to keep track of the current total number of inversions, $c_1 + \cdots + c_n$, as we go.

Many programs need to generate the same permutations repeatedly, and in such cases we needn't run through the steps of Algorithm P each time. We can simply prepare a list of suitable transitions, using the following method:

Algorithm T (*Plain change transitions*). This algorithm computes a table $t[1]$, $t[2]$, $\ldots$, $t[n!-1]$ such that the actions of Algorithm P are equivalent to the successive interchanges $a_{t[k]} \leftrightarrow a_{t[k]+1}$ for $1 \le k < n!$. We assume that $n \ge 2$.

T1. [Initialize.] Set $N \leftarrow n!$, $d \leftarrow N/2$, $t[d] \leftarrow 1$, and $m \leftarrow 2$.

T2. [Loop on m.] Terminate if $m = n$. Otherwise set $m \leftarrow m+1$, $d \leftarrow d/m$, and $k \leftarrow 0$. (We maintain the condition $d = n!/m!$.)

T3. [Hunt down.] Set $k \leftarrow k+d$ and $j \leftarrow m-1$. Then while $j > 0$, set $t[k] \leftarrow j$, $j \leftarrow j-1$, and $k \leftarrow k+d$, until $j = 0$.

T4. [Offset.] Set $t[k] \leftarrow t[k]+1$ and $k \leftarrow k+d$.

T5. [Hunt up.] While $j < m-1$, set $j \leftarrow j+1$, $t[k] \leftarrow j$, and $k \leftarrow k+d$. Return to T3 if $k < N$, otherwise return to T2. ∎

For example, if $n = 4$ we get the table $(t[1], t[2], \ldots, t[23]) = (3,2,1,3,1,2,3,1,3,2,1,3,1,2,3,1,3,2,1,3,1,2,3)$.

Alphametics. Now let's consider a simple kind of puzzle in which permutations are useful: How can the pattern

$$\begin{array}{r} \texttt{SEND} \\ +\,\texttt{MORE} \\ \hline \texttt{MONEY} \end{array} \tag{6}$$

represent a correct sum, if every letter stands for a different decimal digit? [H. E. Dudeney, *Strand* **68** (1924), 97, 214.] Such puzzles are often called "alphametics," a word coined by J. A. H. Hunter [*Globe and Mail* (Toronto: 27 October 1955), 27]; another term, "cryptarithm," has also been suggested by S. Vatriquant [*Sphinx* **1** (May 1931), 50].

The classic alphametic (6) can easily be solved by hand (see exercise 21). But let's suppose we want to deal with a large set of complicated alphametics, some of which may be unsolvable while others may have dozens of solutions. Then we can save time by programming a computer to try out all permutations of digits that match a given pattern, seeing which permutations yield a correct sum. [A computer program for solving alphametics was published by John Beidler in *Creative Computing* **4**, 6 (November–December 1978), 110–113.]

We might as well raise our sights slightly and consider additive alphametics in general, dealing not only with simple sums like (6) but also with examples like

$$\texttt{VIOLIN} + \texttt{VIOLIN} + \texttt{VIOLA} \;=\; \texttt{TRIO} + \texttt{SONATA}.$$

Equivalently, we want to solve puzzles such as

$$2(\texttt{VIOLIN}) + \texttt{VIOLA} - \texttt{TRIO} - \texttt{SONATA} \;=\; 0, \tag{7}$$

where a sum of terms with integer coefficients is given and the goal is to obtain zero by substituting distinct decimal digits for the different letters. Each letter in such a problem has a "signature" obtained by substituting 1 for that letter and 0 for the others; for example, the signature for `I` in (7) is

$$2(010010) + 01000 - 0010 - 000000,$$

namely 21010. If we arbitrarily assign the codes $(1, 2, \ldots, 10)$ to the letters $(\texttt{V}, \texttt{I}, \texttt{O}, \texttt{L}, \texttt{N}, \texttt{A}, \texttt{T}, \texttt{R}, \texttt{S}, \texttt{X})$, the respective signatures corresponding to (7) are

$$\begin{array}{llllll} s_1 = 210000, & s_2 = 21010, & s_3 = -7901, & s_4 = 210, & s_5 = -998, \\ s_6 = -100, & s_7 = -1010, & s_8 = -100, & s_9 = -100000, & s_{10} = 0. \end{array} \tag{8}$$

The problem then is to find all permutations $a_1 \ldots a_{10}$ of $\{0, 1, \ldots, 9\}$ such that

$$a \cdot s \;=\; \sum_{j=1}^{10} a_j s_j \;=\; 0. \tag{9}$$

There also is a side condition, because the numbers in alphametics should not have zero as a leading digit. For example, the sums

$$\begin{array}{r} 7316 \\ +\,0823 \\ \hline 08139 \end{array} \quad \text{and} \quad \begin{array}{r} 5731 \\ +\,0647 \\ \hline 06378 \end{array} \quad \text{and} \quad \begin{array}{r} 6524 \\ +\,0735 \\ \hline 07259 \end{array} \quad \text{and} \quad \begin{array}{r} 2817 \\ +\,0368 \\ \hline 03185 \end{array}$$

and numerous others are *not* considered to be valid solutions of (6). In general there is a set F of first letters such that we must have

$$a_j \neq 0 \qquad \text{for all } j \in F; \tag{10}$$

the set F corresponding to (7) and (8) is $\{1, 7, 9\}$.

One way to tackle a family of additive alphametics is to start by using Algorithm T to prepare a table of $10! - 1$ transitions $t[j]$. Then, for each problem defined by a signature sequence $(s_1, \ldots, s_{10})$ and a first-letter set F, we can exhaustively look for solutions as follows:

A1. [Initialize.] Set $a_1 a_2 \ldots a_{10} \leftarrow 01 \ldots 9$, $v \leftarrow \sum_{j=1}^{10} (j-1)s_j$, $k \leftarrow 1$, and $\delta_j \leftarrow s_{j+1} - s_j$ for $1 \le j < 10$.

A2. [Test.] If $v = 0$ and if (10) holds, output the solution $a_1 \ldots a_{10}$.

A3. [Swap.] Stop if $k = 10!$. Otherwise set $j \leftarrow t[k]$, $v \leftarrow v - (a_{j+1} - a_j)\delta_j$, $a_{j+1} \leftrightarrow a_j$, $k \leftarrow k + 1$, and return to A2. ∎

Step A3 is justified by the fact that swapping a_j with a_{j+1} simply decreases $a \cdot s$ by $(a_{j+1} - a_j)(s_{j+1} - s_j)$. Even though 10! is 3,628,800, a fairly large number, the operations in step A3 are so simple that the whole job takes only a fraction of a second on a modern computer.

An alphametic is said to be *pure* if it has a unique solution. Unfortunately (7) is not pure; the permutations 1764802539 and 3546281970 both solve (9) and (10), hence we have both

$$176478 + 176478 + 17640 \;=\; 2576 + 368020$$

and

$$354652 + 354652 + 35468 \;=\; 1954 + 742818.$$

Furthermore $s_6 = s_8$ in (8), so we can obtain two more solutions by interchanging the digits assigned to `A` and `R`.

On the other hand (6) *is* pure, yet the method we have described will find two different permutations that solve it. The reason is that (6) involves only eight distinct letters, hence we will set it up for solution by using two dummy signatures $s_9 = s_{10} = 0$. In general, an alphametic with m distinct letters will have $10 - m$ dummy signatures $s_{m+1} = \cdots = s_{10} = 0$, and each of its solutions will be found $(10 - m)!$ times unless we insist that, say, $a_{m+1} < \cdots < a_{10}$.

A general framework. A great many algorithms have been proposed for generating permutations of distinct objects, and the best way to understand them is to apply the multiplicative properties of permutations that we studied in Section 1.3.3. For this purpose we will change our notation slightly, by using 0-origin indexing and writing $a_0\,a_1\ldots a_{n-1}$ for permutations of $\{0,1,\ldots,n-1\}$ instead of writing $a_1a_2\ldots a_n$ for permutations of $\{1,2,\ldots,n\}$. More importantly, we will consider schemes for generating permutations in which most of the action takes place at the *left*, so that all permutations of $\{0,1,\ldots,k-1\}$ will be generated during the first $k!$ steps, for $1\le k\le n$. For example, one such scheme for $n=4$ is

$$\begin{aligned}&0123, 1023, 0213, 2013, 1203, 2103, 0132, 1032, 0312, 3012, 1302, 3102,\\ &0231, 2031, 0321, 3021, 2301, 3201, 1230, 2130, 1320, 3120, 2310, 3210;\end{aligned}\qquad(11)$$

this is called "reverse colex order," because if we reflect the strings from right to left we get 3210, 3201, 3120, ..., 0123, the reverse of lexicographic order. Another way to think of (11) is to view the entries as $(n-a_n)\ldots(n-a_2)(n-a_1)$, where $a_1a_2\ldots a_n$ runs lexicographically through the permutations of $\{1,2,\ldots,n\}$.

Let's recall that a permutation like $\alpha=250143$ can be written either in the two-line form

$$\alpha=\begin{pmatrix}012345\\250143\end{pmatrix}$$

or in the more compact cycle form

$$\alpha=(0\ 2)(1\ 5\ 3),$$

with the meaning that α takes $0\mapsto 2$, $1\mapsto 5$, $2\mapsto 0$, $3\mapsto 1$, $4\mapsto 4$, and $5\mapsto 3$; a 1-cycle like '(4)' need not be indicated. Since 4 is a fixed point of this permutation we say that "α fixes 4." We also write $0\alpha=2$, $1\alpha=5$, and so on, saying that $j\alpha$ is "the image of j under α." Multiplication of permutations, like α times β where $\beta=543210$, is readily carried out either in the two-line form

$$\alpha\beta=\begin{pmatrix}012345\\250143\end{pmatrix}\begin{pmatrix}012345\\543210\end{pmatrix}=\begin{pmatrix}012345\\250143\end{pmatrix}\begin{pmatrix}250143\\305412\end{pmatrix}=\begin{pmatrix}012345\\305412\end{pmatrix}$$

or in the cycle form

$$\alpha\beta=(0\ 2)(1\ 5\ 3)\ \cdot\ (0\ 5)(1\ 4)(2\ 3)\ =\ (0\ 3\ 4\ 1)(2\ 5).$$

Notice that the image of 1 under $\alpha\beta$ is $1(\alpha\beta)=(1\alpha)\beta=5\beta=0$, etc. *Warning:* About half of all books that deal with permutations multiply them the other way (from right to left), imagining that $\alpha\beta$ means that β should be applied before α. The reason is that traditional functional notation, in which one writes $\alpha(1)=5$, makes it natural to think that $\alpha\beta(1)$ should mean $\alpha(\beta(1))=\alpha(4)=4$. However, the present book subscribes to the other philosophy, and we shall always multiply permutations from left to right.

The order of multiplication needs to be understood carefully when permutations are represented by arrays of numbers. For example, if we "apply" the reflection $\beta=543210$ to the permutation $\alpha=250143$, the result 341052 is not $\alpha\beta$

but $\beta\alpha$. In general, the operation of replacing a permutation $\alpha = a_0 a_1 \dots a_{n-1}$ by some rearrangement $a_{0\beta} a_{1\beta} \dots a_{(n-1)\beta}$ takes $k \mapsto a_{k\beta} = k\beta\alpha$. Permuting the *positions* by β corresponds to *premultiplication* by β, changing α to $\beta\alpha$; permuting the *values* by β corresponds to *postmultiplication* by β, changing α to $\alpha\beta$. Thus, for example, a permutation generator that interchanges $a_1 \leftrightarrow a_2$ is premultiplying the current permutation by (1 2), postmultiplying it by $(a_1\ a_2)$.

Following a proposal made by Évariste Galois in 1830, a nonempty set G of permutations is said to form a *group* if it is closed under multiplication, that is, if the product $\alpha\beta$ is in G whenever α and β are elements of G [see *Écrits et Mémoires Mathématiques d'Évariste Galois* (Paris: 1962), 47]. Consider, for example, the 4-cube represented as a 4×4 torus

$$\begin{array}{|cccc|}\hline 0 & 1 & 3 & 2 \\ 4 & 5 & 7 & 6 \\ c & d & f & e \\ 8 & 9 & b & a \\ \hline\end{array} \tag{12}$$

as in exercise 7.2.1.1–17, and let G be the set of all permutations of the vertices $\{0, 1, \dots, f\}$ that preserve adjacency: A permutation α is in G if and only if u — v implies $u\alpha$ — $v\alpha$ in the 4-cube. (Here we are using hexadecimal digits $(0, 1, \dots, f)$ to stand for the integers $(0, 1, \dots, 15)$. The labels in (12) are chosen so that u — v if and only if u and v differ in only one bit position.) This set G is obviously a group, and its elements are called the symmetries or "automorphisms" of the 4-cube.

Groups of permutations G are conveniently represented inside a computer by means of a *Sims table*, introduced by Charles C. Sims [*Computational Methods in Abstract Algebra* (Oxford: Pergamon, 1970), 169–183], which is a family of subsets S_1, S_2, ... of G having the following property: S_k contains exactly one permutation σ_{kj} that takes $k \mapsto j$ and fixes the values of all elements greater than k, whenever G contains such a permutation. We let σ_{kk} be the identity permutation, which is always present in G; but when $0 \le j < k$, any suitable permutation can be selected to play the role of σ_{kj}. The main advantage of a Sims table is that it provides a convenient representation of the entire group:

Lemma S. *Let S_1, S_2, ..., S_{n-1} be a Sims table for a group G of permutations on $\{0, 1, \dots, n-1\}$. Then every element α of G has a unique representation*

$$\alpha = \sigma_1 \sigma_2 \dots \sigma_{n-1}, \qquad \text{where } \sigma_k \in S_k \text{ for } 1 \le k < n. \tag{13}$$

Proof. If α has such a representation and if σ_{n-1} is the permutation $\sigma_{(n-1)j} \in S_{n-1}$, then α takes $n-1 \mapsto j$, because all elements of $S_1 \cup \dots \cup S_{n-2}$ fix the value of $n-1$. Conversely, if α takes $n-1 \mapsto j$ we have $\alpha = \alpha' \sigma_{(n-1)j}$, where

$$\alpha' = \alpha\, \sigma^{-}_{(n-1)j}$$

is a permutation of G that fixes $n-1$. The set G' of all such permutations is a group, and S_1, ..., S_{n-2} is a Sims table for G'; therefore the result follows by induction on n. ∎

For example, a bit of calculation shows that one possible Sims table for the automorphism group of the 4-cube is

$$\begin{aligned}
S_{\mathtt{f}} &= \{(),\ (\mathtt{01})(\mathtt{23})(\mathtt{45})(\mathtt{67})(\mathtt{89})(\mathtt{ab})(\mathtt{cd})(\mathtt{ef}),\ \ldots,\\
&\qquad\qquad (\mathtt{0f})(\mathtt{1e})(\mathtt{2d})(\mathtt{3c})(\mathtt{4b})(\mathtt{5a})(\mathtt{69})(\mathtt{78})\};\\
S_{\mathtt{e}} &= \{(),\ (\mathtt{12})(\mathtt{56})(\mathtt{9a})(\mathtt{de}),\ (\mathtt{14})(\mathtt{36})(\mathtt{9c})(\mathtt{be}),\ (\mathtt{18})(\mathtt{3a})(\mathtt{5c})(\mathtt{7e})\};\\
S_{\mathtt{d}} &= \{(),\ (\mathtt{24})(\mathtt{35})(\mathtt{ac})(\mathtt{bd}),\ (\mathtt{28})(\mathtt{39})(\mathtt{6c})(\mathtt{7d})\};\\
S_{\mathtt{c}} &= \{()\};\\
S_{\mathtt{b}} &= \{(),\ (\mathtt{48})(\mathtt{59})(\mathtt{6a})(\mathtt{7b})\};\\
S_{\mathtt{a}} &= S_{\mathtt{9}} = \cdots = S_{\mathtt{1}} = \{()\};
\end{aligned} \tag{14}$$

here $S_{\mathtt{f}}$ contains 16 permutations $\sigma_{\mathtt{f}j}$ for $0 \le j \le 15$, which respectively take $i \mapsto i \oplus (15 - j)$ for $0 \le i \le 15$. The set $S_{\mathtt{e}}$ contains only four permutations, because an automorphism that fixes `f` must take `e` into a neighbor of `f`; thus the image of `e` must be either `e` or `d` or `b` or `7`. The set $S_{\mathtt{c}}$ contains only the identity permutation, because an automorphism that fixes `f`, `e`, and `d` must also fix `c`. Most groups have $S_k = \{()\}$ for all small values of k, as in this example; hence a Sims table usually needs to contain only a fairly small number of permutations although the group itself might be quite large.

The Sims representation (13) makes it easy to test if a given permutation α lies in G: First we determine $\sigma_{n-1} = \sigma_{(n-1)j}$, where α takes $n-1 \mapsto j$, and we let $\alpha' = \alpha\sigma_{n-1}^-$; then we determine $\sigma_{n-2} = \sigma_{(n-2)j'}$, where α' takes $n-2 \mapsto j'$, and we let $\alpha'' = \alpha'\sigma_{n-2}^-$; and so on. If at any stage the required σ_{kj} does not exist in S_k, the original permutation α does not belong to G. In the case of (14), this process must reduce α to the identity after finding $\sigma_{\mathtt{f}}$, $\sigma_{\mathtt{e}}$, $\sigma_{\mathtt{d}}$, $\sigma_{\mathtt{c}}$, and $\sigma_{\mathtt{b}}$.

For example, let α be the permutation $(\mathtt{14})(\mathtt{28})(\mathtt{3c})(\mathtt{69})(\mathtt{7d})(\mathtt{be})$, which corresponds to transposing (12) about its main diagonal $\{\mathtt{0},\mathtt{5},\mathtt{f},\mathtt{a}\}$. Since α fixes `f`, $\sigma_{\mathtt{f}}$ will be the identity permutation (), and $\alpha' = \alpha$. Then $\sigma_{\mathtt{e}}$ is the member of $S_{\mathtt{e}}$ that takes $\mathtt{e} \mapsto \mathtt{b}$, namely $(\mathtt{14})(\mathtt{36})(\mathtt{9c})(\mathtt{be})$, and we find $\alpha'' = (\mathtt{28})(\mathtt{39})(\mathtt{6c})(\mathtt{7d})$. This permutation belongs to $S_{\mathtt{d}}$, so α is indeed an automorphism of the 4-cube.

Conversely, (13) also makes it easy to generate all elements of the corresponding group. We simply run through all permutations of the form

$$\sigma(1, c_1)\sigma(2, c_2)\ldots\sigma(n-1, c_{n-1}),$$

where $\sigma(k, c_k)$ is the $(c_k + 1)$st element of S_k for $0 \le c_k < s_k = \|S_k\|$ and $1 \le k < n$, using any algorithm of Section 7.2.1.1 that runs through all $(n-1)$-tuples $(c_1, \ldots, c_{n-1})$ for the respective radices $(s_1, \ldots, s_{n-1})$.

Using the general framework. Our chief concern is the group of *all* permutations on $\{0, 1, \ldots, n-1\}$, and in this case every set S_k of a Sims table will contain $k+1$ elements $\{\sigma(k,0), \sigma(k,1), \ldots, \sigma(k,k)\}$, where $\sigma(k,0)$ is the identity and the others take k to the values $\{0, \ldots, k-1\}$ in some order (fixing all elements greater than k). Every such Sims table leads to a permutation generator, according to the following outline:

Algorithm G (*General permutation generator*). Given a Sims table $(S_1, S_2, \ldots, S_{n-1})$ where each S_k has $k+1$ elements $\sigma(k,j)$ as just described, this algorithm generates all permutations $a_0 a_1 \ldots a_{n-1}$ of $\{0, 1, \ldots, n-1\}$, using an auxiliary control table $c_n \ldots c_2 c_1$.

G1. [Initialize.] Set $a_j \leftarrow j$ and $c_{j+1} \leftarrow 0$ for $0 \le j < n$.

G2. [Visit.] (At this point the mixed-radix number $\left[\begin{smallmatrix} c_{n-1}, \ldots, c_2, c_1 \\ n, \ \ldots, \ 3, \ 2 \end{smallmatrix}\right]$ is the number of permutations visited so far.) Visit the permutation $a_0 a_1 \ldots a_{n-1}$.

G3. [Add 1 to $c_n \ldots c_2 c_1$.] Set $k \leftarrow 1$. If $c_k = k$, set $c_k \leftarrow 0$, $k \leftarrow k+1$, and repeat until $c_k < k$. Terminate the algorithm if $k = n$; otherwise set $c_k \leftarrow c_k + 1$.

G4. [Permute.] Apply the permutation $\tau(k, c_k)\omega(k-1)^-$ to $a_0 a_1 \ldots a_{n-1}$, as explained below, and return to G2. ∎

Applying a permutation π to $a_0 a_1 \ldots a_{n-1}$ means replacing a_j by $a_{j\pi}$ for $0 \le j < n$; this corresponds to premultiplication by π as explained earlier. Let us define

$$\tau(k,j) = \sigma(k,j)\sigma(k,j-1)^- \qquad \text{for } 1 \le j \le k; \tag{15}$$

$$\omega(k) = \sigma(1,1)\ldots\sigma(k,k). \tag{16}$$

Then steps G3 and G4 maintain the property that

$$a_0 a_1 \ldots a_{n-1} \text{ is the permutation } \sigma(1,c_1)\sigma(2,c_2)\ldots\sigma(n-1,c_{n-1}), \tag{17}$$

and Lemma S proves that every permutation is visited exactly once.

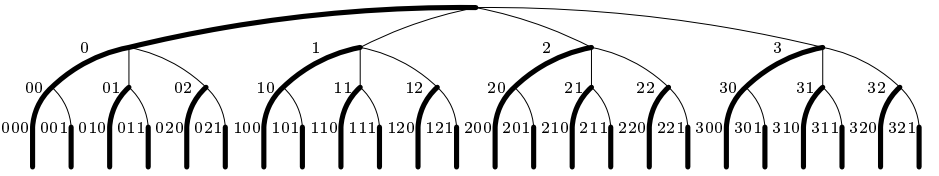


Fig. 19. Algorithm G implicitly traverses this tree when $n = 4$.

The tree in Fig. 19 illustrates Algorithm G in the case $n = 4$. According to (17), every permutation $a_0 a_1 a_2 a_3$ of $\{0,1,2,3\}$ corresponds to a three-digit control string $c_3 c_2 c_1$, with $0 \le c_3 \le 3$, $0 \le c_2 \le 2$, and $0 \le c_1 \le 1$. Some nodes of the tree are labeled by a single digit c_3; these correspond to the permutations $\sigma(3, c_3)$ of the Sims table being used. Other nodes, labeled with two digits $c_3 c_2$, correspond to the permutations $\sigma(2,c_2)\sigma(3,c_3)$. A heavy line connects node c_3 to node $c_3 0$ and node $c_3 c_2$ to node $c_3 c_2 0$, because $\sigma(2,0)$ and $\sigma(1,0)$ are the identity permutation and these nodes are essentially equivalent. Adding 1 to the mixed-radix number $c_3 c_2 c_1$ in step G3 corresponds to moving from one node of Fig. 19 to its successor in preorder, and the transformation in step G4 changes the permutations accordingly. For example, when $c_3 c_2 c_1$ changes from 121 to 200, step G4 premultiplies the current permutation by

$$\tau(3,2)\omega(2)^- = \tau(3,2)\,\sigma(2,2)^-\sigma(1,1)^-;$$

premultiplying by $\sigma(1,1)^-$ takes us from node 121 to node 12, premultiplying by $\sigma(2,2)^-$ takes us from node 12 to node 1, and premultiplying by $\tau(3,2) = \sigma(3,2)\sigma(3,1)^-$ takes us from node 1 to node $2 \equiv 200$, which is the preorder successor of node 121. Stating this another way, premultiplication by $\tau(3,2)\omega(2)^-$ is exactly what is needed to change $\sigma(1,1)\sigma(2,2)\sigma(3,1)$ to $\sigma(1,0)\sigma(2,0)\sigma(3,2)$, preserving (17).

Algorithm G defines a huge number of permutation generators (see exercise 37), so it is no wonder that many of its special cases have appeared in the literature. Of course some of its variants are much more efficient than others, and we want to find examples where the operations are particularly well suited to the computer we are using.

We can, for instance, obtain permutation in reverse colex order as a special case of Algorithm G (see (11)), by letting $\sigma(k,j)$ be the $(j+1)$-cycle

$$\sigma(k,j) \;=\; (k{-}j \;\; k{-}j{+}1 \;\; \ldots \;\; k). \tag{18}$$

The reason is that $\sigma(k,j)$ should be the permutation that corresponds to $c_n \ldots c_1$ in reverse colex order when $c_k = j$ and $c_i = 0$ for $i \neq k$, and this permutation $a_0 a_1 \ldots a_{n-1}$ is $01\ldots(k{-}j{-}1)(k{-}j{+}1)\ldots(k)(k{-}j)(k{+}1)\ldots(n{-}1)$. For example, when $n = 8$ and $c_n \ldots c_1 = 00030000$ the corresponding reverse colex permutation is 01345267, which is $(2\,3\,4\,5)$ in cycle form. When $\sigma(k,j)$ is given by (18), Eqs. (15) and (16) lead to the formulas

$$\tau(k,j) = (k{-}j \;\; k); \tag{19}$$

$$\omega(k) = (0\,1)(0\,1\,2)\ldots(0\,1\,\ldots\,k) = (0\,k)(1\,k{-}1)(2\,k{-}2)\ldots = \phi(k); \tag{20}$$

here $\phi(k)$ is the "$(k+1)$-flip" that changes $a_0 \ldots a_k$ to $a_k \ldots a_0$. In this case $\omega(k)$ turns out to be the same as $\omega(k)^-$, because $\phi(k)^2 = ()$.

Equations (19) and (20) are implicitly present behind the scenes in Algorithm L and in its reverse colex equivalent (exercise 2), where step L3 essentially applies a transposition and step L4 does a flip. Step G4 actually does the flip first; but the identity

$$(k{-}j\;\,k)\phi(k-1) \;=\; \phi(k-1)(j{-}1\;\,k) \tag{21}$$

shows that a flip followed by a transposition is the same as a (different) transposition followed by the flip.

In fact, equation (21) is a special case of the important identity

$$\pi^-(j_1\;j_2\;\ldots\;j_t)\,\pi \;=\; (j_1\pi\;\,j_2\pi\;\ldots\;\,j_t\pi) \tag{22}$$

valid for *any* permutation π and any t-cycle $(j_1\;j_2\;\ldots\;j_t)$. On the left of (22) we have, for example, $j_1\pi \mapsto j_1 \mapsto j_2 \mapsto j_2\pi$, in agreement with the cycle on the right. Therefore if α and π are any permutations whatever, the permutation $\pi^-\alpha\,\pi$ (called the *conjugate* of α by π) has exactly the same cycle structure as α; we simply replace each element j in each cycle by $j\pi$.

Another significant special case of Algorithm G was introduced by R. J. Ord-Smith [*CACM* **10** (1967), 452; **12** (1969), 638; see also *Comp. J.* **14** (1971),

136–139], whose algorithm is obtained by setting

$$\sigma(k,j) = (k\ \ldots\ 1\ 0)^j. \tag{23}$$

Now it is clear from (15) that

$$\tau(k,j) = (k\ \ldots\ 1\ 0); \tag{24}$$

and once again we have

$$\omega(k) = (0\ k)(1\ k{-}1)(2\ k{-}2)\ldots = \phi(k), \tag{25}$$

because $\sigma(k,k) = (0\ 1\ \ldots\ k)$ is the same as before. The nice thing about this method is that the permutation needed in step G4, namely $\tau(k,c_k)\omega(k-1)^-$, does not depend on c_k:

$$\tau(k,j)\omega(k-1)^- = (k\ \ldots\ 1\ 0)\phi(k-1)^- = \phi(k). \tag{26}$$

Thus, Ord-Smith's algorithm is the special case of Algorithm G in which step G4 simply interchanges $a_0 \leftrightarrow a_k$, $a_1 \leftrightarrow a_{k-1}$, ...; this operation is usually quick, because k is small, and it saves some of the work of Algorithm L. (See exercise 38.)

We can do even better by rigging things so that step G4 needs to do only a single transposition each time, somewhat as in Algorithm P but not necessarily on adjacent elements. Many such schemes are possible. The best is probably to let

$$\tau(k,j)\omega(k-1)^- = \begin{cases} (k\ 0), & \text{if } k \text{ is even,} \\ (k\ j{-}1), & \text{if } k \text{ is odd,} \end{cases} \tag{27}$$

as suggested by B. R. Heap [*Comp. J.* **6** (1963), 293–294]. Notice that Heap's method always transposes $a_k \leftrightarrow a_0$ except when $k = 3, 5, \ldots$; and the value of k, in 5 of every 6 steps, is either 1 or 2. Exercise 40 proves that Heap's method does indeed generate all permutations.

Bypassing unwanted blocks. One noteworthy advantage of Algorithm G is that it runs through all permutations of $a_0 \ldots a_{k-1}$ before touching a_k; then it performs another $k!$ cycles before changing a_k again, and so on. Therefore if at any time we reach a setting of the final elements $a_k \ldots a_{n-1}$ that is unimportant to the problem we're working on, we can skip quickly over all permutations that end with the undesirable suffix. More precisely, we could replace step G2 by the following substeps:

G2.0. [Acceptable?] If $a_k \ldots a_{n-1}$ is not an acceptable suffix, go to G2.1. Otherwise set $k \leftarrow k-1$. Then if $k > 0$, repeat this step; if $k = 0$, proceed to step G2.2.

G2.1. [Skip this suffix.] If $c_k = k$, apply $\sigma(k,k)^-$ to $a_0 \ldots a_{n-1}$, set $c_k \leftarrow 0$, $k \leftarrow k+1$, and repeat until $c_k < k$. Terminate if $k = n$; otherwise set $c_k \leftarrow c_k + 1$, apply $\tau(k,c_k)$ to $a_0 \ldots a_{n-1}$, and return to G2.0.

G2.2. [Visit.] Visit the permutation $a_0 \ldots a_{n-1}$. ∎

Step G1 should also set $k \leftarrow n-1$. Notice that the new steps are careful to preserve condition (17). The algorithm has become more complicated, because

we need to know the permutations $\tau(k,j)$ and $\sigma(k,k)$ in addition to the permutations $\tau(k,j)\omega(k-1)^-$ that appear in G4. But the additional complications are often worth the effort, because the resulting program might run significantly faster.

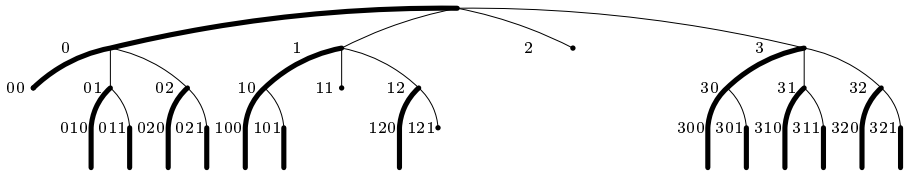


Fig. 20. Unwanted branches can be pruned from the tree of Fig. 19, if Algorithm G is suitably extended.

For example, Fig. 20 shows what happens to the tree of Fig. 19 when the suffixes of $a_0a_1a_2a_3$ that correspond to nodes 00, 11, 121, and 2 are not acceptable. (Each suffix $a_k\ldots a_{n-1}$ of the permutation $a_0\ldots a_{n-1}$ corresponds to a *prefix* $c_n\ldots c_k$ of the control string $c_n\ldots c_1$, because the permutations $\sigma(1,c_1)\ldots\sigma(k-1,c_{k-1})$ do not affect $a_k\ldots a_{n-1}$.) Step G2.1 premultiplies by $\tau(k,j)$ to move from node $c_{n-1}\ldots c_{k+1}j$ to its right sibling $c_{n-1}\ldots c_{k+1}(j{+}1)$, and it premultiplies by $\sigma(k,k)^-$ to move up from node $c_{n-1}\ldots c_{k+1}k$ to its parent $c_{n-1}\ldots c_{k+1}$. Thus, to get from the rejected prefix 121 to its preorder successor, the algorithm premultiplies by $\sigma(1,1)^-$, $\sigma(2,2)^-$, and $\tau(3,2)$, thereby moving from node 121 to 12 to 1 to 2. (This is a somewhat exceptional case, because a prefix with $k=1$ is rejected only if we don't want to visit the unique permutation $a_0a_1\ldots a_{n-1}$ that has suffix $a_1\ldots a_{n-1}$.) After node 2 is rejected, $\tau(3,3)$ takes us to node 3, etc.

Notice, incidentally, that bypassing a suffix $a_k\ldots a_{n-1}$ in this extension of Algorithm G is essentially the same as bypassing a prefix $a_1\ldots a_j$ in our original notation, if we go back to the idea of generating permutations $a_1\ldots a_n$ of $\{1,\ldots,n\}$ and doing most of the work at the right-hand end. Our original notation corresponds to choosing a_1 first, then a_2, ..., then a_n; the notation in Algorithm G essentially chooses a_{n-1} first, then a_{n-2}, ..., then a_0. Algorithm G's conventions may seem backward, but they make the formulas for Sims table manipulation a lot simpler. A good programmer soon learns to switch without difficulty from one viewpoint to another.

We can apply these ideas to alphametics, because it is clear for example that most choices of the values for the letters `D`, `E`, and `Y` will make it impossible for `SEND` plus `MORE` to equal `MONEY`: We need to have $(\mathtt{D}+\mathtt{E}-\mathtt{Y}) \bmod 10 = 0$ in that problem. Therefore many permutations can be eliminated from consideration.

In general, if r_k is the maximum power of 10 that divides the signature value s_k, we can sort the letters and assign codes $\{0,1,\ldots,9\}$ so that $r_0 \ge r_1 \ge \cdots \ge r_9$. For example, to solve the trio sonata problem (7), we could use $(0,1,\ldots,9)$ respectively for $(\mathtt{X},\mathtt{S},\mathtt{V},\mathtt{A},\mathtt{R},\mathtt{I},\mathtt{L},\mathtt{T},\mathtt{O},\mathtt{N})$, obtaining the signatures

$$s_0=0,\quad s_1=-100000,\quad s_2=210000,\quad s_3=-100,\quad s_4=-100,$$
$$s_5=21010,\quad s_6=210,\quad s_7=-1010,\quad s_8=-7901,\quad s_9=-998;$$

hence $(r_0,\ldots,r_9)=(\infty,5,4,2,2,1,1,1,0,0)$. Now if we get to step G2.0 for a value of k with $r_{k-1}\neq r_k$, we can say that the suffix $a_k\ldots a_9$ is unacceptable unless $a_k s_k+\cdots+a_9 s_9$ is a multiple of $10^{r_{k-1}}$. Also, (10) tells us that $a_k\ldots a_9$ is unacceptable if $a_k=0$ and $k\in F$; the first-letter set F is now $\{1,2,7\}$.

Our previous approach to alphametics with steps A1–A3 above used brute force to run through 10! possibilities. It operated rather fast under the circumstances, since the adjacent-transposition method allowed it to get by with only 6 memory references per permutation; but still, 10! is 3,628,800, so the entire process cost almost 22 megamems, regardless of the alphametic being solved. By contrast, the extended Algorithm G with Heap's method and the cutoffs just described will find all four solutions to (7) with fewer than 128 *kilo*mems! Thus the suffix-skipping technique runs more than 170 times faster than the previous method, which simply blasted away blindly.

Most of the 128 kilomems in the new approach are spent applying $\tau(k,c_k)$ in step G2.1. The other memory references come primarily from applications of $\sigma(k,k)^-$ in that step, but τ is needed 7812 times while σ^- is needed only 2162 times. The reason is easy to understand from Fig. 20, because the "shortcut move" $\tau(k,c_k)\omega(k-1)^-$ in step G4 hardly ever applies; in this case it is used only four times, once for each solution. Thus, preorder traversal of the tree is accomplished almost entirely by τ steps that move to the right and σ^- steps that move upward. The τ steps dominate in a problem like this, where very few complete permutations are actually visited, because each step $\sigma(k,k)^-$ is preceded by k steps $\tau(k,1)$, $\tau(k,2)$, $\ldots$, $\tau(k,k)$.

This analysis reveals that Heap's method—which goes to great lengths to optimize the permutations $\tau(k,j)\omega(k-1)^-$ so that each transition in step G4 is a simple transposition—is not especially good for the extended Algorithm G unless comparatively few suffixes are rejected in step G2.0. The simpler reverse colex order, for which $\tau(k,j)$ itself is always a simple transposition, is now much more attractive (see (19)). Indeed, Algorithm G with reverse colex order solves the alphametic (7) with only 97 kilomems.

Similar results occur with respect to other alphametic problems. For example, if we apply the extended Algorithm G to the alphametics in exercise 24, parts (a) through (h), the computations involve respectively

$$\begin{array}{l}(551,\ 110,\ 14,\ 8,\ 350,\ 84,\ 153,\ 1598)\ \text{kilomems with Heap's method;}\\ (429,\ \ 84,\ 10,\ 5,\ 256,\ 63,\ 117,\ 1189)\ \text{kilomems with reverse colex.}\end{array}\qquad(28)$$

The speedup factor for reverse colex in these examples, compared to brute force with Algorithm T, ranges from 18 in case (h) to 4200 in case (d), and it is about 80 on the average; Heap's method gives an average speedup of about 60.

We know from Algorithm L, however, that lexicographic order is easily handled *without* the complication of the control table $c_n\ldots c_1$ used by Algorithm G. And a closer look at Algorithm L shows that we can improve its behavior when permutations are frequently being skipped, by using a linked list instead of a sequential array. The improved algorithm is well-suited to a wide variety of algorithms that wish to generate restricted classes of permutations:

Algorithm X (*Lexicographic permutations with restricted prefixes*). This algorithm generates all permutations $a_1 a_2 \dots a_n$ of $\{1, 2, \dots, n\}$ that pass a given sequence of tests

$$t_1(a_1), \quad t_2(a_1, a_2), \quad \dots, \quad t_n(a_1, a_2, \dots, a_n),$$

visiting them in lexicographic order. It uses an auxiliary table of links l_0, l_1, $\dots$, l_n to maintain a cyclic list of unused elements, so that if the currently available elements are

$$\{1, \dots, n\} \setminus \{a_1, \dots, a_k\} = \{b_1, \dots, b_{n-k}\}, \qquad \text{where } b_1 < \cdots < b_{n-k}, \tag{29}$$

then we have

$$l_0 = b_1, \quad l_{b_j} = b_{j+1} \quad \text{for } 1 \le j < n-k, \quad \text{and} \quad l_{b_{n-k}} = 0. \tag{30}$$

It also uses an auxiliary table $u_1 \dots u_n$ to undo operations that have been performed on the l array.

X1. [Initialize.] Set $l_k \leftarrow k+1$ for $0 \le k < n$, and $l_n \leftarrow 0$. Then set $k \leftarrow 1$.

X2. [Enter level k.] Set $p \leftarrow 0$, $q \leftarrow l_0$.

X3. [Test $a_1 \dots a_k$.] Set $a_k \leftarrow q$. If $t_k(a_1, \dots, a_k)$ is false, go to X5. Otherwise, if $k = n$, visit $a_1 \dots a_n$ and go to X6.

X4. [Increase k.] Set $u_k \leftarrow p$, $l_p \leftarrow l_q$, $k \leftarrow k+1$, and return to X2.

X5. [Increase a_k.] Set $p \leftarrow q$, $q \leftarrow l_p$. If $q \neq 0$ return to X3.

X6. [Decrease k.] Set $k \leftarrow k-1$, and terminate if $k = 0$. Otherwise set $p \leftarrow u_k$, $q \leftarrow a_k$, $l_p \leftarrow q$, and go to X5. ∎

The basic idea of this elegant algorithm is due to M. C. Er [*Comp. J.* **30** (1987), 282]. We can apply it to alphametics by changing notation slightly, obtaining permutations $a_0 \dots a_9$ of $\{0, \dots, 9\}$ and letting l_{10} play the former role of l_0. The resulting algorithm needs only 49 kilomems to solve the trio-sonata problem (7), and it solves the alphametics of exercise 24(a)–(h) in

$$(248,\ 38,\ 4,\ 3,\ 122,\ 30,\ 55,\ 553) \text{ kilomems}, \tag{31}$$

respectively. Thus it runs about 165 times faster than the brute-force approach.

Another way to apply Algorithm X to alphametics is often faster yet (see exercise 49).

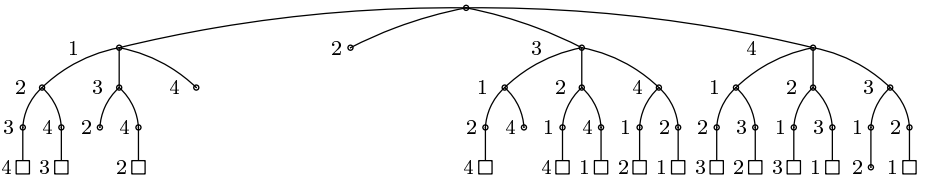


Fig. 21. The tree implicitly traversed by Algorithm X when $n = 4$, if all permutations are visited except those beginning with 132, 14, 2, 314, or 4312.

***Dual methods.** If $S_1, \ldots, S_{n-1}$ is a Sims table for a permutation group G, we learned in Lemma S that every element of G can be expressed uniquely as a product $\sigma_1 \ldots \sigma_{n-1}$, where $\sigma_k \in S_k$; see (13). Exercise 50 shows that every element α can also be expressed uniquely in the dual form

$$\alpha = \sigma_{n-1}^- \ldots \sigma_2^- \sigma_1^-, \qquad \text{where } \sigma_k \in S_k \text{ for } 1 \le k < n, \tag{32}$$

and this fact leads to another large family of permutation generators. In particular, when G is the group of all $n!$ permutations, every permutation can be written

$$\sigma(n-1, c_{n-1})^- \ldots \sigma(2, c_2)^- \sigma(1, c_1)^-, \tag{33}$$

where $0 \le c_k \le k$ for $1 \le k < n$ and the permutations $\sigma(k, j)$ are the same as in Algorithm G. Now, however, we want to vary c_{n-1} most rapidly and c_1 least rapidly, so we arrive at an algorithm of a different kind:

Algorithm H (*Dual permutation generator*). Given a Sims table as in Algorithm G, this algorithm generates all permutations $a_0 \ldots a_{n-1}$ of $\{0, \ldots, n-1\}$, using an auxiliary table $c_0 \ldots c_{n-1}$.

H1. [Initialize.] Set $a_j \leftarrow j$ and $c_j \leftarrow 0$ for $0 \le j < n$.

H2. [Visit.] (At this point the mixed-radix number $\left[\begin{smallmatrix} c_1, & c_2, & \ldots, & c_{n-1} \\ 2, & 3, & \ldots, & n \end{smallmatrix}\right]$ is the number of permutations visited so far.) Visit the permutation $a_0 a_1 \ldots a_{n-1}$.

H3. [Add 1 to $c_0 c_1 \ldots c_{n-1}$.] Set $k \leftarrow n-1$. If $c_k = k$, set $c_k \leftarrow 0$, $k \leftarrow k-1$, and repeat until $k = 0$ or $c_k < k$. Terminate the algorithm if $k = 0$; otherwise set $c_k \leftarrow c_k + 1$.

H4. [Permute.] Apply the permutation $\tau(k, c_k)\omega(k+1)^-$ to $a_0 a_1 \ldots a_{n-1}$, as explained below, and return to H2. ∎

Although this algorithm looks almost identical to Algorithm G, the permutations τ and ω that it needs in step H4 are quite different from those needed in step G4. The new rules, which replace (15) and (16), are

$$\tau(k, j) = \sigma(k, j)^- \sigma(k, j-1), \qquad \text{for } 1 \le j \le k, \tag{34}$$

$$\omega(k) = \sigma(n-1, n-1)^- \sigma(n-2, n-2)^- \ldots \sigma(k, k)^-. \tag{35}$$

The number of possibilities is just as vast as it was for Algorithm G, so we will confine our attention to a few cases that have special merit. One natural case to try is, of course, the Sims table that makes Algorithm G produce reverse colex order, namely

$$\sigma(k, j) \;=\; (k{-}j \;\; k{-}j{+}1 \;\; \ldots \;\; k) \tag{36}$$

as in (18). The resulting permutation generator turns out to be very nearly the same as the method of plain changes; so we can say that Algorithms L and P are essentially dual to each other. (See exercise 52.)

Another natural idea is to construct a Sims table for which step H4 always makes a single transposition of two elements, by analogy with the construction of (27) that achieves maximum efficiency in step G4. But such a mission now turns out to be impossible: We cannot achieve it even when $n = 4$. For if

we start with the identity permutation $a_0a_1a_2a_3 = 0123$, the transitions that take us from control table $c_0c_1c_2c_3 = 0000$ to 0001 to 0002 to 0003 must move the 3; so, if they are transpositions, they must be $(3\,a)$, $(a\,b)$, and $(b\,c)$ for some permutation abc of $\{0,1,2\}$. The permutation corresponding to $c_0c_1c_2c_3 = 0003$ is now $\sigma(3,3)^- = (b\,c)(a\,b)(3\,a) = (3\,a\,b\,c)$; and the next permutation, which corresponds to $c_0c_1c_2c_3 = 0010$, will be $\sigma(2,1)^-$, which must fix the element 3. The only suitable transposition is $(3\,c)$, hence $\sigma(2,1)^-$ must be $(3\,c)(3\,a\,b\,c) = (a\,b\,c)$. Similarly we find that $\sigma(2,2)^-$ must be $(a\,c\,b)$, and the permutation corresponding to $c_0c_1c_2c_3 = 0023$ will be $(3\,a\,b\,c)(a\,c\,b) = (3\,c)$. Step H4 is now supposed to convert this to the permutation $\sigma(1,1)^-$, which corresponds to the control table 0100 that follows 0023. But the only transposition that will convert $(3\,c)$ into a permutation that fixes 2 and 3 is $(3\,c)$; and the resulting permutation also fixes 1, so it cannot be $\sigma(1,1)^-$.

The proof in the preceding paragraph shows that we cannot use Algorithm H to generate all permutations with the minimum number of transpositions. But it also suggests a simple generation scheme that comes very close to the minimum, and the resulting algorithm is quite attractive because it needs to do extra work only once per $n(n-1)$ steps. (See exercise 53.)

Finally, let's consider the dual of Ord-Smith's method, when

$$\sigma(k,j) = (k\ \ldots\ 1\ 0)^j \tag{37}$$

as in (23). Once again the value of $\tau(k,j)$ is independent of j,

$$\tau(k,j) = (0\ 1\ \ldots\ k), \tag{38}$$

and this fact is particularly advantageous in Algorithm H because it allows us to dispense with the control table $c_0c_1\ldots c_{n-1}$. The reason is that $c_{n-1} = 0$ in step H3 if and only if $a_{n-1} = n-1$, because of (32); and indeed, when $c_j = 0$ for $k < j < n$ in step H3 we have $c_k = 0$ if and only if $a_k = k$. Therefore we can reformulate this variant of Algorithm H as follows.

Algorithm C (*Permutation generation by cyclic shifts*). This algorithm visits all permutations $a_1\ldots a_n$ of the distinct elements $\{x_1,\ldots,x_n\}$.

C1. [Initialize.] Set $a_j \leftarrow x_j$ for $1 \le j \le n$.

C2. [Visit.] Visit the permutation $a_1\ldots a_n$, and set $k \leftarrow n$.

C3. [Shift.] Replace $a_1a_2\ldots a_k$ by the cyclic shift $a_2\ldots a_ka_1$, and return to C2 if $a_k \neq x_k$.

C4. [Decrease k.] Set $k \leftarrow k-1$, and go back to C3 if $k > 1$. ∎

For example, the successive permutations of $\{1,2,3,4\}$ generated when $n = 4$ are

1234, 2341, 3412, 4123, (1234),
2314, 3142, 1423, 4231, (2314),
3124, 1243, 2431, 4312, (3124), (1234),
2134, 1342, 3421, 4213, (2134),
1324, 3241, 2413, 4132, (1324),
3214, 2143, 1432, 4321, (3214), (2134), (1234),

with unvisited intermediate permutations shown in parentheses. This algorithm may well be the simplest permutation generator of all, in terms of minimum program length. It is due to G. G. Langdon, Jr. [*CACM* **10** (1967), 298–299; **11** (1968), 392]; similar methods had been published previously by C. Tompkins [*Proc. Symp. Applied Math.* **6** (1956), 202–205] and, more explicitly, by R. Seitz [*Unternehmensforschung* **6** (1962), 2–15]. The procedure is particularly well suited to applications in which cyclic shifting is efficient, for example when successive permutations are being kept in a machine register instead of in an array.

The main disadvantage of dual methods is that they usually do not adapt well to situations where large blocks of permutations need to be skipped, because the set of all permutations with a given value of the first control entries $c_0 c_1 \ldots c_{k-1}$ is usually not of importance. The special case (36) is, however, sometimes an exception, because the $n!/k!$ permutations with $c_0 c_1 \ldots c_{k-1} = 00\ldots0$ in that case are precisely those $a_0 a_1 \ldots a_{n-1}$ in which 0 precedes 1, 1 precedes 2, ..., and $k-2$ precedes $k-1$.

***Ehrlich's swap method.** Gideon Ehrlich has discovered a completely different approach to permutation generation, based on yet another way to use a control table $c_1 \ldots c_{n-1}$. His method obtains each permutation from its predecessor by interchanging the leftmost element with another:

Algorithm E (*Ehrlich swaps*). This algorithm generates all permutations of the distinct elements $a_0 \ldots a_{n-1}$ by using auxiliary tables $b_0 \ldots b_{n-1}$ and $c_1 \ldots c_n$.

E1. [Initialize.] Set $b_j \leftarrow j$ and $c_{j+1} \leftarrow 0$ for $0 \le j < n$.

E2. [Visit.] Visit the permutation $a_0 \ldots a_{n-1}$.

E3. [Find k.] Set $k \leftarrow 1$. Then if $c_k = k$, set $c_k \leftarrow 0$, $k \leftarrow k+1$, and repeat until $c_k < k$. Terminate if $k = n$, otherwise set $c_k \leftarrow c_k + 1$.

E4. [Swap.] Interchange $a_0 \leftrightarrow a_{b_k}$.

E5. [Flip.] Set $j \leftarrow 1$, $k \leftarrow k-1$. If $j < k$, interchange $b_j \leftrightarrow b_k$, set $j \leftarrow j+1$, $k \leftarrow k-1$, and repeat until $j \ge k$. Return to E2. ∎

Notice that steps E2 and E3 are identical to steps G2 and G3 of Algorithm G. The most amazing thing about this algorithm, which Ehrlich communicated to Martin Gardner in 1987, is that it works; exercise 55 contains a proof. A similar method, which simplifies the operations of step E5, can be validated in the same way (see exercise 56). The average number of interchanges performed in step E5 is less than 0.18 (see exercise 57).

As it stands, Algorithm E isn't faster than other methods we have seen. But it has the nice property that it changes each permutation in a minimal way, using only $n-1$ different kinds of transpositions. Whereas Algorithm P used adjacent interchanges, $a_{t-1} \leftrightarrow a_t$, Algorithm E uses first-element swaps, $a_0 \leftrightarrow a_t$, also called *star transpositions*, for some well-chosen sequence of indices $t[1]$, $t[2]$, ..., $t[n!-1]$. And if we are generating permutations repeatedly for the same fairly small value of n, we can precompute this sequence, as we did in Algorithm T

for the index sequence of Algorithm P. Notice that star transpositions have an advantage over adjacent interchanges, because we always know the value of a_0 from the previous swap; we need not read it from memory.

Let E_n be the sequence of $n!-1$ indices t such that Algorithm E swaps a_0 with a_t in step E4. Since E_{n+1} begins with E_n, we can regard E_n as the first $n!-1$ elements of an infinite sequence

$$E_\infty = 1212132121231212132121243131321313121\ldots. \tag{39}$$

For example, if $n = 4$ and $a_0a_1a_2a_3 = 1234$, the permutations visited by Algorithm E are

$$\begin{aligned} &1234,\ 2134,\ 3124,\ 1324,\ 2314,\ 3214,\\ &4213,\ 1243,\ 2143,\ 4123,\ 1423,\ 2413,\\ &3412,\ 4312,\ 1342,\ 3142,\ 4132,\ 1432,\\ &2431,\ 3421,\ 4321,\ 2341,\ 3241,\ 4231. \end{aligned} \tag{40}$$

***Using fewer generators.** After seeing Algorithms P and E, we might naturally ask whether all permutations can be obtained by using just *two* basic operations, instead of $n-1$. For example, Nijenhuis and Wilf [*Combinatorial Algorithms* (1975), Exercise 6] noticed that all permutations can be generated for $n = 4$ if we replace $a_1a_2a_3\ldots a_n$ at each step by either $a_2a_3\ldots a_na_1$ or $a_2a_1a_3\ldots a_n$, and they wondered whether such a method exists for all n.

In general, if G is any group of permutations and if $\alpha_1, \ldots, \alpha_k$ are elements of G, the *Cayley graph* for G with generators $(\alpha_1, \ldots, \alpha_k)$ is the directed graph whose vertices are the permutations π of G and whose arcs go from π to $\alpha_1\pi, \ldots, \alpha_k\pi$. [Arthur Cayley, *American J. Math.* **1** (1878), 174–176.] The question of Nijenhuis and Wilf is equivalent to asking whether the Cayley graph for all permutations of $\{1, 2, \ldots, n\}$, with generators σ and τ where σ is the cyclic permutation $(1\ 2\ \ldots\ n)$ and τ is the transposition $(1\ 2)$, has a Hamiltonian path.

A basic theorem due to R. A. Rankin [*Proc. Cambridge Philos. Soc.* **44** (1948), 17–25] allows us to conclude in many cases that Cayley graphs with two generators do not have a Hamiltonian circuit:

Theorem R. *Let G be a group consisting of g permutations. If the Cayley graph for G with generators (α, β) has a Hamiltonian circuit, and if the permutations $(\alpha, \beta, \alpha\beta^-)$ are respectively of order (a, b, c), then either c is even or g/a and g/b are odd.*

(The *order* of a permutation α is the least positive integer a such that α^a is the identity.)

Proof. See exercise 72. ∎

In particular, when $\alpha = \sigma$ and $\beta = \tau$ as above, we have $g = n!$, $a = n$, $b = 2$, and $c = n-1$, because $\sigma\tau^- = (2\ \ldots\ n)$. Therefore we conclude that no Hamiltonian circuit is possible when $n \geq 4$ is even. However, a Hamiltonian *path* is easy to

construct when $n = 4$, because we can join up the 12-cycles

$$\begin{aligned} &1234 \to 2341 \to 3412 \to 4312 \to 3124 \to 1243 \to 2431 \\ &\qquad \to 4231 \to 2314 \to 3142 \to 1423 \to 4123 \to 1234, \\ &2134 \to 1342 \to 3421 \to 4321 \to 3214 \to 2143 \to 1432 \\ &\qquad \to 4132 \to 1324 \to 3241 \to 2413 \to 4213 \to 2134, \end{aligned} \tag{41}$$

by starting at 2341 and jumping from 1234 to 2134, ending at 4213.

Ruskey, Jiang, and Weston [*Discrete Applied Math.* **57** (1995), 75–83] undertook an exhaustive search in the σ–τ graph for $n = 5$ and discovered that it has five essentially distinct Hamiltonian circuits, one of which (the "most beautiful") is illustrated in Fig. 22(a). They also found a Hamiltonian path for $n = 6$; this was a difficult feat, because it is the outcome of a 720-stage binary decision tree. Unfortunately the solution they discovered has no apparent logical structure. A somewhat less complex path is described in exercise 70, but even that path cannot be called simple. Therefore a σ–τ approach will probably not be of practical interest for larger values of n unless a new construction is discovered. R. C. Compton and S. G. Williamson [*Linear and Multilinear Algebra* **35** (1993), 237–293] have proved that Hamiltonian circuits exist for all n if the three generators σ, σ^-, and τ are allowed instead of just σ and τ; their cycles have the interesting property that every nth transformation is τ, and the intervening $n-1$ transformations are either all σ or all σ^-. But their method is too complicated to explain in a short space.

Exercise 69 describes a general permutation algorithm that is reasonably simple and needs only three generators, each of order 2. Fig. 22(b) illustrates the case $n = 5$ of this method, which was motivated by examples of bell-ringing.

(a) Using only transitions (1 2 3 4 5) and (1 2).

(b) Using only transitions (1 2)(3 4), (2 3)(4 5), and (3 4).

Fig. 22. Hamiltonian circuits for 5! permutations.

Faster, faster. What is the fastest way to generate permutations? This question has often been raised in computer publications, because people who examine $n!$ possibilities want to keep the running time as small as possible. But the answers have generally been contradictory, because there are many different ways to formulate the question. Let's try to understand the related issues by studying how permutations might be generated most rapidly on the `MMIX` computer.

Suppose first that our goal is to produce permutations in an array of n consecutive memory words (octabytes). The fastest way to do this, of all those we've seen in this section, is to streamline Heap's method (27), as suggested by R. Sedgewick [*Computing Surveys* **9** (1977), 157–160].

The key idea is to optimize the code for the most common cases of steps G2 and G3, namely the cases in which all activity occurs at the beginning of the array. If registers u, v, and w contain the contents of the first three words, and if the next six permutations to be generated involve permuting those words in all six possible ways, we can clearly do the job as follows:

```
PUSHJ 0,Visit
STO v,A0;  STO u,A1;  PUSHJ 0,Visit
STO w,A0;  STO v,A2;  PUSHJ 0,Visit
STO u,A0;  STO w,A1;  PUSHJ 0,Visit
STO v,A0;  STO u,A2;  PUSHJ 0,Visit
STO w,A0;  STO v,A1;  PUSHJ 0,Visit
```
(42)

(Here `A0` is the address of octabyte a_0, etc.) A complete permutation program, which takes care of getting the right things into u, v, and w, appears in exercise 76, but the other instructions are less important because they need to be performed only $\frac{1}{6}$ of the time. The total cost per permutation, not counting the $4v$ needed for `PUSHJ` and `POP` on each call to `Visit`, comes to approximately $2.77\mu + 5.69v$ with this approach. If we use four registers u, v, w, x, and if we expand (42) to 24 calls on `Visit`, the running time per permutation drops to about $2.19\mu + 3.07v$. And with r registers and $r!$ `Visit`s, exercise 77 shows that the cost is $(2 + O(1/r!))(\mu + v)$, which is very nearly the cost of two `STO` instructions.

The latter is, of course, the minimum possible time for any method that generates all permutations in a sequential array. ... Or is it? We have assumed that the visiting routine wants to see permutations in consecutive locations, but perhaps that routine is able to read the permutations from different starting points. Then we can arrange to keep a_{n-1} fixed and to keep two copies of the other elements in its vicinity:

$$a_0 a_1 \ldots a_{n-2} a_{n-1} a_0 a_1 \ldots a_{n-2}. \tag{43}$$

If we now let $a_0 a_1 \ldots a_{n-2}$ run through $(n-1)!$ permutations, always changing both copies simultaneously by doing two `STO` commands instead of one, we can let every call to `Visit` look at the n permutations

$$a_0 a_1 \ldots a_{n-1}, \quad a_1 \ldots a_{n-1} a_0, \quad \ldots, \quad a_{n-1} a_0 \ldots a_{n-2}, \tag{44}$$

which all appear consecutively. The cost per permutation is now reduced to the cost of three simple instructions like `ADD`, `CMP`, `PBNZ`, plus $O(1/n)$. [See Varol and Rotem, *Comp. J.* **24** (1981), 173–176.]

Furthermore, we might not want to waste time storing permutations into memory at all. Suppose, for example, that our goal is to generate all permutations of $\{0, 1, \ldots, n-1\}$. The value of n will probably be at most 16, because $16! = 20{,}922{,}789{,}888{,}000$ and $17! = 355{,}687{,}428{,}096{,}000$. Therefore an entire permutation will fit in the 16 nybbles of an octabyte, and we can keep it in a single register. This will be advantageous only if the visiting routine doesn't need to unpack the individual nybbles; but let's suppose that it doesn't. How fast can we generate permutations in the nybbles of a 64-bit register?

One idea, suggested by a technique due to A. J. Goldstein [*U. S. Patent 3383661* (14 May 1968)], is to precompute the table $(t[1], \ldots, t[5039])$ of plain-change transitions for seven elements, using Algorithm T. These numbers $t[k]$ lie between 1 and 6, so we can pack 20 of them into a 64-bit word. It is convenient to put the number $\sum_{k=1}^{20} 2^{3k-1} t[20j+k]$ into word j of an auxiliary table, for $0 \le j < 252$, with $t[5040] = 1$; for example, the table begins with the codeword

00|001|010|011|100|101|110|100|110|101|100|011|010|001|110|001|010|011|100|101|110|00.

The following program reads such codes efficiently:

```
Perm  ⟨Set register a to the first permutation⟩
0H    LDA  p,T       p ← address of first codeword.
      JMP  3F
1H    ⟨Visit the permutation in register a⟩
      ⟨Swap the nybbles of a that lie t bits from the right⟩
      SRU  c,c,3     c ← c ≫ 3.
2H    AND  t,c,#1c   t ← c ∧ (11100)₂.
      PBNZ t,1B      Branch if t ≠ 0.
      ADD  p,p,8
3H    LDO  c,p,0     c ← next codeword.
      PBNZ c,2B      (The final codeword is followed by 0.)
      ⟨If not done, advance the leading n − 7 nybbles and return to 0B⟩
```
(45)

Exercise 78 shows how to ⟨Swap the nybbles ...⟩ with seven instructions, using bit manipulation operations that are found on most computers. Therefore the cost per permutation is just a bit more than $10v$. (The instructions that fetch new codewords cost only $(\mu + 5v)/20$; and the instructions that advance the leading $n-7$ nybbles are even more negligible since their cost is divided by 5040.) Notice that there is now no need for `PUSHJ` and `POP` as there was with (42); we ignored those instructions before, but they did cost $4v$.

We can, however, do even better by adapting Langdon's cyclic-shift method, Algorithm C. Suppose we start with the lexicographically largest permutation and operate as follows:

```
      GREG @
0H    OCTA #fedcba9876543210&(1<<(4*N)-1)
Perm  LDOU a,0B                          Set a ← #...3210.
      JMP  2F
1H    SRU  a,a,4*(16-N)                  a ← ⌊a/16^(16−n)⌋.
      OR   a,a,t                         a ← a ∨ t.
2H    ⟨Visit the permutation in register a⟩
      SRU  t,a,4*(N-1)                   t ← ⌊a/16^(n−1)⌋.
      SLU  a,a,4*(17-N)                  a ← 16^(17−n) a mod 16^16.
      PBNZ t,1B                          To 1B if t ≠ 0.
      ⟨Continue with Langdon's method⟩
```
(46)

The running time per permutation is now only $5v + O(1/n)$, again without the need for `PUSHJ` and `POP`. See exercise 80 for an interesting way to extend (46) to a complete program, obtaining a remarkably short and fast routine.

Fast permutation generators are amusing, but in practice we can usually save more time by streamlining the visiting routine than by speeding up the generator.

Topological sorting. Instead of working with all $n!$ permutations of $\{1,\ldots,n\}$, we often want to look only at permutations that obey certain restrictions. For example, we might be interested only in permutations for which 1 precedes 3, 2 precedes 3, and 2 precedes 4; there are five such permutations of $\{1,2,3,4\}$, namely

$$1234,\ 1243,\ 2134,\ 2143,\ 2413. \tag{47}$$

The problem of *topological sorting*, which we studied in Section 2.2.3 as a first example of nontrivial data structures, is the general problem of finding a permutation that satisfies m such conditions $x_1 \prec y_1, \ldots, x_m \prec y_m$, where $x \prec y$ means that x should precede y in the permutation. This problem arises frequently in practice, so it has several different names; for example, it is often called the *linear embedding* problem, because we want to arrange objects in a line while preserving certain order relationships. It is also the problem of extending a partial ordering to a total ordering (see exercise 2.2.3–14).

Our goal in Section 2.2.3 was to find a *single* permutation that satisfied all the relations. But now we want rather to find *all* such permutations, all topological sorts. Indeed, we will assume in the present section that the elements x and y on which the relations are defined are integers between 1 and n, and that we have $x < y$ whenever $x \prec y$. Consequently the permutation $12\ldots n$ will always be topologically correct. (If this simplifying assumption is not met, we can preprocess the data by using Algorithm 2.2.3T to rename the objects appropriately.)

Many important classes of permutations are special cases of this topological ordering problem. For example, the permutations of $\{1,\ldots,8\}$ such that

$$1 \prec 2, \quad 2 \prec 3, \quad 3 \prec 4, \quad 6 \prec 7, \quad 7 \prec 8$$

are equivalent to permutations of the multiset $\{1,1,1,1,2,3,3,3\}$, because we can map $\{1,2,3,4\} \mapsto 1$, $5 \mapsto 2$, and $\{6,7,8\} \mapsto 3$. We know how to generate permutations of a multiset using Algorithm L, but now we will learn another way.

Notice that x precedes y in a permutation $a_1\ldots a_n$ if and only if $a'_x < a'_y$ in the inverse permutation $a'_1\ldots a'_n$. Therefore the algorithm we are about to study will also find all permutations $a'_1\ldots a'_n$ such that $a'_j < a'_k$ whenever $j \prec k$. For example, we learned in Section 5.1.4 that a Young tableau is an arrangement of $\{1,\ldots,n\}$ in rows and columns so that each row is increasing from left to right and each column is increasing from top to bottom. The problem of generating all 3×3 Young tableaux is therefore equivalent to generating all $a'_1\ldots a'_9$ such that

$$\begin{aligned} &a'_1 < a'_2 < a'_3, \quad a'_4 < a'_5 < a'_6, \quad a'_7 < a'_8 < a'_9,\\ &a'_1 < a'_4 < a'_7, \quad a'_2 < a'_5 < a'_8, \quad a'_3 < a'_6 < a'_9, \end{aligned} \tag{48}$$

and this is a special kind of topological sorting.

We might also want to find all *matchings* of $2n$ elements, namely all ways to partition $\{1,\ldots,2n\}$ into n pairs. There are $(2n-1)(2n-3)\ldots(1) = (2n)!/(2^n n!)$ ways to do this, and they correspond to permutations that satisfy

$$a'_1 < a'_2, \quad a'_3 < a'_4, \quad \ldots, \quad a'_{2n-1} < a'_{2n}, \qquad a'_1 < a'_3 < \cdots < a'_{2n-1}. \tag{49}$$

An elegant algorithm for exhaustive topological sorting was discovered by Y. L. Varol and D. Rotem [*Comp. J.* **24** (1981), 83–84], who realized that a method analogous to plain changes (Algorithm P) can be used. Suppose we have found a way to arrange $\{1,\ldots,n-1\}$ topologically, so that $a_1\ldots a_{n-1}$ satisfies all the conditions that do not involve n. Then we can easily write down all the allowable ways to insert the final element n without changing the relative order of $a_1\ldots a_{n-1}$: We simply start with $a_1\ldots a_{n-1}\,n$, then shift n left one step at a time until it cannot move further. Applying this idea recursively yields the following straightforward procedure.

Algorithm V (*All topological sorts*). Given a relation $\prec$ on $\{1,\ldots,n\}$ with the property that $x \prec y$ implies $x < y$, this algorithm generates all permutations $a_1\ldots a_n$ and their inverses $a'_1\ldots a'_n$ with the property that $a'_j < a'_k$ whenever $j \prec k$. We assume for convenience that $a_0 = 0$ and that $0 \prec k$ for $1 \le k \le n$.

V1. [Initialize.] Set $a_j \leftarrow j$ and $a'_j \leftarrow j$ for $0 \le j \le n$.

V2. [Visit.] Visit the permutation $a_1\ldots a_n$ and its inverse $a'_1\ldots a'_n$. Then set $k \leftarrow n$.

V3. [Can k move left?] Set $j \leftarrow a'_k$ and $l \leftarrow a_{j-1}$. If $l \prec k$, go to V5.

V4. [Yes, move it.] Set $a_{j-1} \leftarrow k$, $a_j \leftarrow l$, $a'_k \leftarrow j-1$, and $a'_l \leftarrow j$. Go to V2.

V5. [No, put k back.] While $j < k$, set $l \leftarrow a_{j+1}$, $a_j \leftarrow l$, $a'_l \leftarrow j$, and $j \leftarrow j+1$. Then set $a_k \leftarrow a'_k \leftarrow k$. Decrease k by 1 and return to V3 if $k > 0$. ∎

For example, Theorem 5.1.4H tells us that there are exactly 42 Young tableaux of size 3×3. If we apply Algorithm V to the relations (48) and write the inverse permutation in array form

$$\begin{bmatrix} a'_1 & a'_2 & a'_3 \\ a'_4 & a'_5 & a'_6 \\ a'_7 & a'_8 & a'_9 \end{bmatrix}, \tag{50}$$

we get the following 42 results:

$$\begin{bmatrix}1&2&3\\4&5&6\\7&8&9\end{bmatrix}\quad\begin{bmatrix}1&2&3\\4&5&7\\6&8&9\end{bmatrix}\quad\begin{bmatrix}1&2&3\\4&5&8\\6&7&9\end{bmatrix}\quad\begin{bmatrix}1&2&3\\4&6&7\\5&8&9\end{bmatrix}\quad\begin{bmatrix}1&2&3\\4&6&8\\5&7&9\end{bmatrix}\quad\begin{bmatrix}1&2&4\\3&5&6\\7&8&9\end{bmatrix}\quad\begin{bmatrix}1&2&4\\3&5&7\\6&8&9\end{bmatrix}\quad\begin{bmatrix}1&2&4\\3&5&8\\6&7&9\end{bmatrix}\quad\begin{bmatrix}1&2&4\\3&6&7\\5&8&9\end{bmatrix}\quad\begin{bmatrix}1&2&4\\3&6&8\\5&7&9\end{bmatrix}\quad\begin{bmatrix}1&2&5\\3&6&7\\4&8&9\end{bmatrix}\quad\begin{bmatrix}1&2&5\\3&6&8\\4&7&9\end{bmatrix}\quad\begin{bmatrix}1&2&5\\3&6&7\\4&8&9\end{bmatrix}\quad\begin{bmatrix}1&2&5\\3&6&8\\4&7&9\end{bmatrix}$$

$$\begin{bmatrix}1&2&5\\3&4&8\\6&7&9\end{bmatrix}\quad\begin{bmatrix}1&2&6\\3&4&7\\5&8&9\end{bmatrix}\quad\begin{bmatrix}1&2&6\\3&4&8\\5&7&9\end{bmatrix}\quad\begin{bmatrix}1&2&7\\3&4&8\\5&6&9\end{bmatrix}\quad\begin{bmatrix}1&2&6\\3&5&7\\4&8&9\end{bmatrix}\quad\begin{bmatrix}1&2&6\\3&5&8\\4&7&9\end{bmatrix}\quad\begin{bmatrix}1&2&7\\3&5&8\\4&6&9\end{bmatrix}\quad\begin{bmatrix}1&2&6\\3&7&8\\4&5&9\end{bmatrix}\quad\begin{bmatrix}1&2&7\\3&6&8\\4&5&9\end{bmatrix}\quad\begin{bmatrix}1&2&7\\3&6&8\\4&5&9\end{bmatrix}\quad\begin{bmatrix}1&2&7\\3&6&8\\4&5&9\end{bmatrix}\quad\begin{bmatrix}1&2&8\\3&6&7\\4&5&9\end{bmatrix}$$

$$\begin{bmatrix}1&4&5\\2&6&7\\3&8&9\end{bmatrix}\quad\begin{bmatrix}1&4&5\\2&6&8\\3&7&9\end{bmatrix}\quad\begin{bmatrix}1&3&5\\2&6&7\\4&8&9\end{bmatrix}\quad\begin{bmatrix}1&3&5\\2&4&7\\6&8&9\end{bmatrix}\quad\begin{bmatrix}1&3&6\\2&4&7\\5&8&9\end{bmatrix}\quad\begin{bmatrix}1&3&6\\2&4&8\\5&7&9\end{bmatrix}\quad\begin{bmatrix}1&3&7\\2&4&8\\5&6&9\end{bmatrix}\quad\begin{bmatrix}1&3&6\\2&5&7\\4&8&9\end{bmatrix}\quad\begin{bmatrix}1&3&6\\2&5&8\\4&7&9\end{bmatrix}\quad\begin{bmatrix}1&3&7\\2&5&8\\4&6&9\end{bmatrix}\quad\begin{bmatrix}1&4&6\\2&5&7\\3&8&9\end{bmatrix}\quad\begin{bmatrix}1&4&6\\2&5&8\\3&7&9\end{bmatrix}\quad\begin{bmatrix}1&4&7\\2&5&8\\3&6&9\end{bmatrix}$$

Let t_r be the number of topological sorts for which the final $n-r$ elements are in their initial position $a_j = j$ for $r < j \le n$. Equivalently, t_r is the number of topological sorts $a_1 \dots a_r$ of $\{1, \dots, r\}$, when we ignore the relations involving elements greater than r. Then the recursive mechanism underlying Algorithm V shows that step V2 is performed N times and step V3 is performed M times, where

$$M = t_n + \cdots + t_1 \qquad \text{and} \qquad N = t_n. \tag{51}$$

Also, step V4 and the loop operations of V5 are performed $N-1$ times; the rest of step V5 is done $M - N + 1$ times. Therefore the total running time of the algorithm is a linear combination of M, N, and n.

If the element labels are chosen poorly, M might be much larger than N. For example, if the constraints input to Algorithm V are

$$2 \prec 3, \quad 3 \prec 4, \quad \dots, \quad n-1 \prec n, \tag{52}$$

then $t_j = j$ for $1 \le j \le n$ and we have $M = \frac{1}{2}(n^2 + n)$, $N = n$. But those constraints are also equivalent to

$$1 \prec 2, \quad 2 \prec 3, \quad \dots, \quad n-2 \prec n-1, \tag{53}$$

under renaming of the elements; then M is reduced to $2n - 1 = 2N - 1$.

Exercise 88 shows that a simple preprocessing step will find element labels so that a slight modification of Algorithm V is able to generate all topological sorts in $O(N+n)$ steps. Thus topological sorting can always be done efficiently.

Think twice before you permute. We have seen several attractive algorithms for permutation generation in this section, but many algorithms are known by which permutations that are optimum for particular purposes can be found *without* running through all possibilities. For example, Theorem 6.1S showed that we can find the best way to arrange records on a sequential storage simply by sorting them with respect to a certain cost criterion, and this process takes only $O(n \log n)$ steps. In Section 7.5.2 we will study the *assignment problem*, which asks how to permute the columns of a square matrix so that the sum of the diagonal elements is maximized. That problem can be solved in at most $O(n^3)$ operations, so it would be foolish to use a method of order $n!$ unless n is extremely small. Even in cases like the traveling salesrep problem, when no efficient algorithm is known, we can usually find a much better approach than to examine every possible solution. Permutation generation is best used when there is good reason to look at each permutation individually.

EXERCISES

▸ **1.** [*20*] Explain how to make Algorithm L run faster, by streamlining its operations when the value of j is near n.

2. [*20*] Rewrite Algorithm L so that it produces all permutations of $a_1 \dots a_n$ in reverse colex order. (In other words, the values of the reflections $a_n \dots a_1$ should be lexicographically decreasing, as in (11). This form of the algorithm is often simpler and faster than the original, because fewer calculations depend on the value of n.)

▸ **3.** [*M21*] The *rank* of a combinatorial arrangement X with respect to a generation algorithm is the number of other arrangements that the algorithm visits prior to X. Explain how to compute the rank of a given permutation $a_1 \dots a_n$ with respect to Algorithm L, if $\{a_1, \dots, a_n\} = \{1, \dots, n\}$. What is the rank of 314592687?

4. [*M23*] Generalizing exercise 3, explain how to compute the rank of $a_1 \dots a_n$ with respect to Algorithm L when $\{a_1, \dots, a_n\}$ is the multiset $\{n_1 \cdot x_1, \dots, n_t \cdot x_t\}$; here $n_1 + \cdots + n_t = n$ and $x_1 < \cdots < x_t$. (The total number of permutations is, of course, the multinomial coefficient

$$\binom{n}{n_1, \dots, n_t} = \frac{n!}{n_1! \dots n_t!};$$

see Eq. 5.1.2–(3).) What is the rank of 314159265?

5. [*HM25*] Compute the mean and variance of the number of comparisons made by Algorithm L in (a) step L2, (b) step L3, when the elements $\{a_1, \dots, a_n\}$ are distinct.

6. [*HM34*] Derive generating functions for the mean number of comparisons made by Algorithm L in (a) step L2, (b) step L3, when $\{a_1, \dots, a_n\}$ is a general multiset as in exercise 4. Also give the results in closed form when $\{a_1, \dots, a_n\}$ is the binary multiset $\{m \cdot 0, (n-m) \cdot 1\}$.

7. [*HM35*] What is the limit as $t \to \infty$ of the average number of comparisons made per permutation in step L2 when Algorithm L is being applied to the multiset (a) $\{2 \cdot 1, 2 \cdot 2, \dots, 2 \cdot t\}$? (b) $\{1 \cdot 1, 2 \cdot 2, \dots, t \cdot t\}$? (c) $\{2 \cdot 1, 4 \cdot 2, \dots, 2^t \cdot t\}$?

▸ **8.** [*21*] The *variations* of a multiset are the permutations of all its submultisets. For example, the variations of $\{1, 2, 2, 3\}$ are

ϵ, 1, 12, 122, 1223, 123, 1232, 13, 132, 1322,
2, 21, 212, 2123, 213, 2132, 22, 221, 2213, 223, 2231, 23, 231, 2312, 232, 2321,
3, 31, 312, 3122, 32, 321, 3212, 322, 3221.

Show that simple changes to Algorithm L will generate all variations of a given multiset $\{a_1, a_2, \dots, a_n\}$.

9. [*22*] Continuing the previous exercise, design an algorithm to generate all r-variations of a given multiset $\{a_1, a_2, \dots, a_n\}$, namely all permutations of its r-element submultisets. (For example, the solution to an alphametic with r distinct letters is an r-variation of $\{0, 1, \dots, 9\}$.)

10. [*20*] What are the values of $a_1 a_2 \dots a_n$, $c_1 c_2 \dots c_n$, and $d_1 d_2 \dots d_n$ at the end of Algorithm P, if $a_1 a_2 \dots a_n = 12 \dots n$ at the beginning?

11. [*M22*] How many times is each step of Algorithm P performed? (Assume that $n \geq 2$.)

▸ **12.** [*M23*] What is the 1000000th permutation visited by (a) Algorithm L, (b) Algorithm P, (c) Algorithm C, if $\{a_1, \dots, a_n\} = \{0, \dots, 9\}$? *Hint:* In mixed-radix notation we have $1000000 = \left[\begin{smallmatrix} 2, & 6, & 6, & 2, & 5, & 1, & 2, & 2, & 0, & 0 \\ 10, & 9, & 8, & 7, & 6, & 5, & 4, & 3, & 2, & 1 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 0, & 0, & 1, & 2, & 3, & 0, & 2, & 7, & 1, & 0 \\ 1, & 2, & 3, & 4, & 5, & 6, & 7, & 8, & 9, & 10 \end{smallmatrix}\right]$.

13. [*M21*] (Martin Gardner, 1974.) True or false: If $a_1 a_2 \dots a_n$ is initially $12 \dots n$, Algorithm P begins by visiting all $n!/2$ permutations in which 1 precedes 2; then the next permutation is $n \dots 21$.

14. [*M22*] True or false: If $a_1 a_2 \dots a_n$ is initially $x_1 x_2 \dots x_n$ in Algorithm P, we always have $a_{j-c_j+s} = x_j$ at the beginning of step P5.

15. [*M23*] (Selmer Johnson, 1963.) Show that the offset variable s never exceeds 2 in Algorithm P.

16. [*21*] Explain how to make Algorithm P run faster, by streamlining its operations when the value of j is near n. (This problem is analogous to exercise 1.)

▶ **17.** [*20*] Extend Algorithm P so that the *inverse permutation* $a'_1 \ldots a'_n$ is available for processing when $a_1 \ldots a_n$ is visited in step P2. (The inverse satisfies $a'_k = j$ if and only if $a_j = k$.)

18. [*21*] (*Rosary permutations.*) Devise an efficient way to generate $(n-1)!/2$ permutations that represent all possible undirected cycles on the vertices $\{1, \ldots, n\}$; that is, no cyclic shift of $a_1 \ldots a_n$ or $a_n \ldots a_1$ will be generated if $a_1 \ldots a_n$ is generated. The permutations (1234, 1324, 3124) could, for example, be used when $n = 4$.

19. [*25*] Construct an algorithm that generates all permutations of n distinct elements *looplessly* in the spirit of Algorithm 7.2.1.1L.

▶ **20.** [*20*] The n-cube has $2^n n!$ symmetries, one for each way to permute and/or complement the coordinates. Such a symmetry is conveniently represented as a *signed permutation*, namely a permutation with optional signs attached to the elements. For example, $2\overline{3}1$ is a signed permutation that transforms the vertices of the 3-cube by changing $x_1x_2x_3$ to $x_2x_3\overline{x}_1$, so that $000 \mapsto 001$, $001 \mapsto 011$, $\ldots$, $111 \mapsto 110$. Design a simple algorithm that generates all signed permutations of $\{1, 2, \ldots, n\}$, where each step either interchanges two adjacent elements or negates the first element.

21. [*M21*] (E. P. McCravy, 1971.) How many solutions does the alphametic (6) have in radix b?

22. [*M15*] True or false: If an alphametic has a solution in radix b, it has a solution in radix $b + 1$.

23. [*M20*] True or false: A pure alphametic cannot have two identical signatures $s_j = s_k \neq 0$ when $j \neq k$.

24. [*25*] Solve the following alphametics by hand or by computer:

a) SEND + A + TAD + MORE = MONEY.

b) ZEROES + ONES = BINARY. (Peter Macdonald, 1977)

c) DCLIX + DLXVI = MCCXXV. (Willy Enggren, 1972)

d) COUPLE + COUPLE = QUARTET. (Michael R. W. Buckley, 1977)

e) FISH + N + CHIPS = SUPPER. (Bob Vinnicombe, 1978)

f) SATURN + URANUS + NEPTUNE + PLUTO = PLANETS. (Willy Enggren, 1968)

g) EARTH + AIR + FIRE + WATER = NATURE. (Herman Nijon, 1977)

h) AN + ACCELERATING + INFERENTIAL + ENGINEERING + TALE + ELITE + GRANT + FEE + ET + CETERA = ARTIFICIAL + INTELLIGENCE.

i) HARDY + NESTS = NASTY + HERDS.

▶ **25.** [*M21*] Devise a fast way to compute $\min(a \cdot s)$ and $\max(a \cdot s)$ over all valid permutations $a_1 \ldots a_{10}$ of $\{0, \ldots, 9\}$, given the signature vector $s = (s_1, \ldots, s_{10})$ and the first-letter set F of an alphametic problem. (Such a procedure makes it possible to rule out many cases quickly when a large family of alphametics is being considered, as in several of the exercises that follow, because a solution can exist only when $\min(a \cdot s) \leq 0 \leq \max(a \cdot s)$.)

26. [*25*] What is the unique alphametic solution to

$$\texttt{NIIHAU} \pm \texttt{KAUAI} \pm \texttt{OAHU} \pm \texttt{MOLOKAI} \pm \texttt{LANAI} \pm \texttt{MAUI} \pm \texttt{HAWAII} = 0?$$

27. [*30*] Construct pure additive alphametics in which all words have five letters.

28. [*M25*] A *partition* of the integer n is an expression of the form $n = n_1 + \cdots + n_t$ with $n_1 \geq \cdots \geq n_t > 0$. Such a partition is called *doubly true* if $\alpha(n) = \alpha(n_1) + \cdots + \alpha(n_t)$ is also a pure alphametic, where $\alpha(n)$ is the "name" of n in some language. Doubly true partitions were introduced by Alan Wayne in *AMM* **54** (1947), 38, 412–414, where he suggested solving TWENTY = SEVEN + SEVEN + SIX and a few others.

a) Find all partitions that are doubly true in English when $1 \leq n \leq 20$.

b) Wayne also gave the example EIGHTY = FIFTY + TWENTY + NINE + ONE. Find all doubly true partitions for $1 \leq n \leq 100$ in which the parts are *distinct*, using the names ONE, TWO, ..., NINETYNINE, ONEHUNDRED.

▸ **29.** [*M25*] Continuing the previous exercise, find all equations of the form $n_1 + \cdots + n_t = n'_1 + \cdots + n'_{t'}$ that are both mathematically and alphametically true in English, when $\{n_1, \ldots, n_t, n'_1, \ldots, n'_{t'}\}$ are distinct positive integers less than 20. For example,

$$\texttt{TWELVE} + \texttt{NINE} + \texttt{TWO} = \texttt{ELEVEN} + \texttt{SEVEN} + \texttt{FIVE};$$

the alphametics should all be pure.

30. [*25*] Solve these multiplicative alphametics by hand or by computer:

a) TWO × TWO = SQUARE. (H. E. Dudeney, 1929)

b) HIP × HIP = HURRAY. (Willy Enggren, 1970)

c) PI × R × R = AREA. (Brian Barwell, 1981)

d) NORTH/SOUTH = EAST/WEST. (Nob Yoshigahara, 1995)

31. [*M22*] (Nob Yoshigahara.) What is the unique solution to A/BC+D/EF+G/HI = 1, when {A, ..., I} = {1, ..., 9}?

32. [*M25*] (H. E. Dudeney, 1901.) Find all ways to represent 100 by inserting a plus sign and a slash into a permutation of the digits {1, ..., 9}. For example, $100 = 91 + 5742/638$. The plus sign should precede the slash.

33. [*25*] Continuing the previous exercise, find all positive integers less than 150 that (a) cannot be represented in such a fashion; (b) have a unique representation.

34. [*M26*] Make the equation EVEN + ODD + PRIME = x doubly true when (a) x is a perfect 5th power; (b) x is a perfect 7th power.

▸ **35.** [*M20*] The automorphisms of a 4-cube have many different Sims tables, only one of which is shown in (14). How many different Sims tables are possible for that group, when the vertices are numbered as in (12)?

36. [*M23*] Find a Sims table for the group of all automorphisms of the 4×4 tic-tac-toe board

0	1	2	3
4	5	6	7
8	9	a	b
c	d	e	f

,

namely the permutations that take lines into lines, where a "line" is a set of four elements that belong to a row, column, or diagonal.

▸ **37.** [*HM22*] How many Sims tables can be used with Algorithms G or H? Estimate the logarithm of this number as $n \to \infty$.

38. [*HM21*] Prove that the average number of transpositions per permutation when using Ord-Smith's algorithm (26) is approximately $\sinh 1 \approx 1.175$.

39. [*16*] Write down the 24 permutations generated for $n = 4$ by (a) Ord-Smith's method (26); (b) Heap's method (27).

40. [*M23*] Show that Heap's method (27) corresponds to a valid Sims table.

▸ **41.** [*M31*] Design an algorithm that generates all r-variations of $\{0, 1, \ldots, n-1\}$ by interchanging just two elements when going from one variation to the next. (See exercise 9.) *Hint:* Generalize Heap's method (27), obtaining the results in positions $a_{n-r} \ldots a_{n-1}$ of an array $a_0 \ldots a_{n-1}$. For example, one solution when $n = 5$ and $r = 2$ uses the final two elements of the respective permutations 01234, 31204, 30214, 30124, 40123, 20143, 24103, 24013, 34012, 14032, 13042, 13402, 23401, 03421, 02431, 02341, 12340, 42310, 41320, 41230.

42. [*M20*] Construct a Sims table for all permutations in which every $\sigma(k, j)$ and every $\tau(k, j)$ for $1 \le j \le k$ is a cycle of length ≤ 3.

43. [*M24*] Construct a Sims table for all permutations in which every $\sigma(k, k)$, $\omega(k)$, and $\tau(k, j)\omega(k-1)^-$ for $1 \le j \le k$ is a cycle of length ≤ 3.

44. [*20*] When blocks of unwanted permutations are being skipped by the extended Algorithm G, is the Sims table of Ord-Smith's method (23) superior to the Sims table of the reverse colex method (18)?

45. [*20*] (a) What are the indices $u_1 \ldots u_9$ when Algorithm X visits the permutation 314592687? (b) What permutation is visited when $u_1 \ldots u_9 = 314157700$?

46. [*20*] True or false: When Algorithm X visits $a_1 \ldots a_n$, we have $u_k > u_{k+1}$ if and only if $a_k > a_{k+1}$, for $1 \le k < n$.

▸ **47.** [*M21*] Express the number of times that each step of Algorithm X is performed in terms of the numbers N_0, N_1, ..., N_n, where N_k is the number of prefixes $a_1 \ldots a_k$ that satisfy $t_j(a_1, \ldots, a_j)$ for $1 \le j \le k$.

▸ **48.** [*M25*] Compare the running times of Algorithm X and Algorithm L, in the case when the tests $t_1(a_1)$, $t_2(a_1, a_2)$, ..., $t_n(a_1, a_2, \ldots, a_n)$ always are true.

▸ **49.** [*28*] The text's suggested method for solving additive alphametics with Algorithm X essentially chooses digits from right to left; in other words, it assigns tentative values to the least significant digits before considering digits that correspond to higher powers of 10.

Explore an alternative approach that chooses digits from left to right. For example, such a method will deduce immediately that $\texttt{M} = 1$ when $\texttt{SEND} + \texttt{MORE} = \texttt{MONEY}$. *Hint:* See exercise 25.

50. [*M15*] Explain why the dual formula (32) follows from (13).

51. [*M16*] True or false: If the sets $S_k = \{\sigma(k, 0), \ldots, \sigma(k, k)\}$ form a Sims table for the group of all permutations, so also do the sets $S_k^- = \{\sigma(k, 0)^-, \ldots, \sigma(k, k)^-\}$.

▸ **52.** [*M22*] What permutations $\tau(k, j)$ and $\omega(k)$ arise when Algorithm H is used with the Sims table (36)? Compare the resulting generator with Algorithm P.

▸ **53.** [*M26*] (F. M. Ives.) Construct a Sims table for which Algorithm H will generate all permutations by making only $n! + O((n-2)!)$ transpositions.

54. [*20*] Would Algorithm C work properly if step C3 did a right-cyclic shift, setting $a_1 \ldots a_{k-1} a_k \leftarrow a_k a_1 \ldots a_{k-1}$, instead of a left-cyclic shift?

55. [*M27*] Consider the *factorial ruler function*

$$\rho_!(m) = \max\{k \mid m \bmod k! = 0\}.$$

Let σ_k and τ_k be permutations of the nonnegative integers such that $\sigma_j \tau_k = \tau_k \sigma_j$ whenever $j \le k$. Let α_0 and β_0 be the identity permutation, and for $m > 0$ define

$$\alpha_m = \beta_{m-1}^- \tau_{\rho_!(m)} \beta_{m-1} \alpha_{m-1}, \qquad \beta_m = \sigma_{\rho_!(m)} \beta_{m-1}.$$

For example, if σ_k is the flip operation $(1\,k{-}1)(2\,k{-}2)\ldots = (0\,k)\phi(k)$ and if $\tau_k = (0\,k)$, and if Algorithm E is started with $a_j = j$ for $0 \le j < n$, then α_m and β_m are the contents of $a_0 \ldots a_{n-1}$ and $b_0 \ldots b_{n-1}$ after step E5 has been performed m times.

a) Prove that $\beta_{(n+1)!}\alpha_{(n+1)!} = \sigma_{n+1}\sigma_n^-\tau_{n+1}\tau_n^-(\beta_{n!}\alpha_{n!})^{n+1}$.

b) Use the result of (a) to establish the validity of Algorithm E.

56. [*M22*] Prove that Algorithm E remains valid if step E5 is replaced by

E5′. [Transpose pairs.] If $k > 2$, interchange $b_{j+1} \leftrightarrow b_j$ for $j = k-2, k-4, \ldots,$ (2 or 1). Return to E2. ▮

57. [*HM22*] What is the average number of interchanges made in step E5?

58. [*M21*] True or false: If Algorithm E begins with $a_0 \ldots a_{n-1} = x_1 \ldots x_n$ then the final permutation visited begins with $a_0 = x_n$.

59. [*M20*] Some authors define the arcs of a Cayley graph as running from π to $\pi\alpha_j$ instead of from π to $\alpha_j\pi$. Are the two definitions essentially different?

▸ **60.** [*21*] A *Gray code for permutations* is a cycle $(\pi_0, \pi_1, \ldots, \pi_{n!-1})$ that includes every permutation of $\{1, 2, \ldots, n\}$ and has the property that π_k differs from $\pi_{(k+1) \bmod n!}$ by an adjacent transposition. It can also be described as a Hamiltonian circuit on the Cayley graph for the group of all permutations on $\{1, 2, \ldots, n\}$, with the $n-1$ generators $\big((1\ 2), (2\ 3), \ldots, (n{-}1\ n)\big)$. The *delta sequence* of such a Gray code is the sequence of integers $\delta_0\delta_1 \ldots \delta_{n!-1}$ such that

$$\pi_{(k+1) \bmod n!} = (\delta_k\ \delta_k{+}1)\,\pi_k.$$

(See 7.2.1.1–(24), which describes the analogous situation for binary n-tuples.) For example, Fig. 23 illustrates the Gray code defined by plain changes when $n = 4$; its delta sequence is $(32131231)^3$.

a) Find all Gray codes for permutations of $\{1, 2, 3, 4\}$.

b) Two Gray codes are considered to be equivalent if their delta sequences can be obtained from each other by cyclic shifting $(\delta_k \ldots \delta_{n!-1}\delta_0 \ldots \delta_{k-1})$ and/or reversal $(\delta_{n!-1} \ldots \delta_1\delta_0)$ and/or complementation $((n{-}\delta_0)(n{-}\delta_1)\ldots(n{-}\delta_{n!-1}))$. Which of the Gray codes in (a) are equivalent?

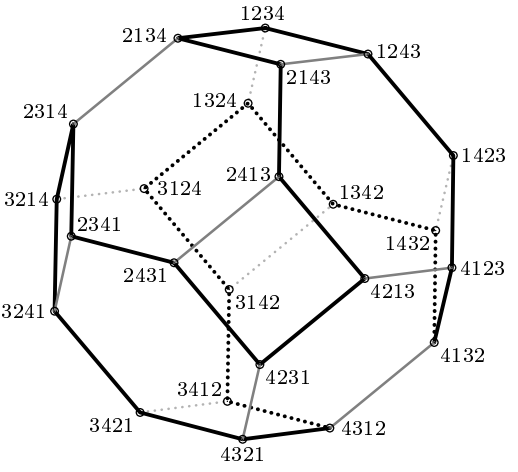


Fig. 23. Algorithm P traces out this Hamiltonian circuit on the truncated octahedron of Fig. 5–1.

61. [*21*] Continuing the previous exercise, a *Gray path for permutations* is like a Gray code except that the final permutation $\pi_{n!-1}$ is not required to be adjacent to the initial permutation π_0. Study the set of all Gray paths for $n = 4$ that start with 1234.

▸ **62.** [*M23*] What permutations can be reached as the final element of a Gray path that starts at $12\ldots n$?

63. [*M25*] Estimate the total number of Gray codes for permutations of $\{1, 2, 3, 4, 5\}$.

64. [*23*] A "doubly Gray" code for permutations is a Gray code with the additional property that $\delta_{k+1} = \delta_k \pm 1$ for all k. Compton and Williamson have proved that such codes exist for all $n \geq 3$. How many doubly Gray codes exist for $n = 5$?

65. [*M25*] For which integers N is there a Gray path through the N lexicographically smallest permutations of $\{1, \ldots, n\}$? (Exercise 7.2.1.1–26 solves the analogous problem for binary n-tuples.)

66. [*22*] Ehrlich's swap method suggests another type of Gray code for permutations, in which the $n-1$ generators are the star transpositions (1 2), (1 3), ..., (1 n). For example, Fig. 24 shows the relevant graph when $n = 4$. Analyze the Hamiltonian circuits of this graph.

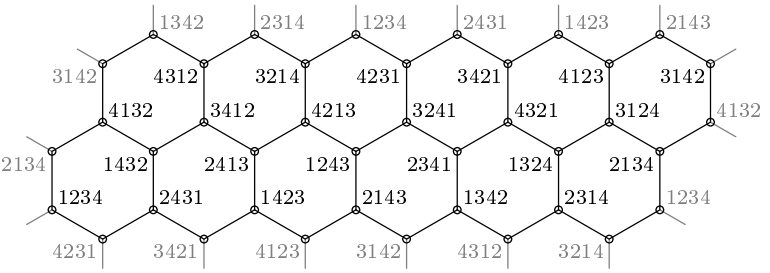


Fig. 24. The Cayley graph for permutations of $\{1, 2, 3, 4\}$, generated by the star transpositions (1 2), (1 3), and (1 4), drawn as a twisted torus.

67. [*26*] Continuing the previous exercise, find a first-element-swap Gray code for $n = 5$ in which each star transposition (1 j) occurs 30 times, for $2 \leq j \leq 5$.

68. [*M30*] (Kompel'makher and Liskovets, 1975.) Let G be the Cayley graph for all permutations of $\{1, \ldots, n\}$, with generators $(\alpha_1, \ldots, \alpha_k)$ where each α_j is a transposition $(u_j\ v_j)$; also let A be the graph with vertices $\{1, \ldots, n\}$ and edges u_j — v_j for $1 \leq j \leq k$. Prove that G has a Hamiltonian circuit if and only if A is connected. (Fig. 23 is the special case when A is a path; Fig. 24 is the special case when A is a "star.")

▸ **69.** [*28*] If $n \geq 4$, the following algorithm generates all permutations $A_1A_2A_3\ldots A_n$ of $\{1, 2, 3, \ldots, n\}$ using only three transformations,

$$\rho = (1\,2)(3\,4)(5\,6)\ldots, \qquad \sigma = (2\,3)(4\,5)(6\,7)\ldots, \qquad \tau = (3\,4)(5\,6)(7\,8)\ldots,$$

never applying ρ and τ next to each other. Explain why it works.

Z1. [Initialize.] Set $A_j \leftarrow j$ for $1 \leq j \leq n$. Also set $a_j \leftarrow 2j$ for $j \leq n/2$ and $a_{n-j} \leftarrow 2j + 1$ for $j < n/2$. Then invoke Algorithm P, but with parameter $n - 1$ instead of n. We will treat that algorithm as a coroutine, which should

return control to us whenever it "visits" $a_1 \dots a_{n-1}$ in step P2. We will also share its variables (except n).

Z2. [Set x and y.] Invoke Algorithm P again, obtaining a new permutation $a_1 \dots a_{n-1}$ and a new value of j. If $j = 2$, interchange $a_{1+s} \leftrightarrow a_{2+s}$ (thereby undoing the effect of step P5) and repeat this step; in such a case we are at the halfway point of Algorithm P. If $j = 1$ (so that Algorithm P has terminated), set $x \leftarrow y \leftarrow 0$ and go to Z3. Otherwise set

$$x \leftarrow a_{j-c_j+s+[d_j=-1]}, \qquad y \leftarrow a_{j-c_j+s-[d_j=+1]};$$

these are the two elements most recently interchanged in step P5.

Z3. [Visit.] Visit the permutation $A_1 \dots A_n$. Then go to Z5 if $A_1 = x$ and $A_2 = y$.

Z4. [Apply ρ, then σ.] Interchange $A_1 \leftrightarrow A_2$, $A_3 \leftrightarrow A_4$, $A_5 \leftrightarrow A_6$, ... Visit $A_1 \dots A_n$. Then interchange $A_2 \leftrightarrow A_3$, $A_4 \leftrightarrow A_5$, $A_6 \leftrightarrow A_7$, ... Terminate if $A_1 \dots A_n = 1 \dots n$, otherwise return to Z3.

Z5. [Apply τ, then σ.] Interchange $A_3 \leftrightarrow A_4$, $A_5 \leftrightarrow A_6$, $A_7 \leftrightarrow A_8$, ... Visit $A_1 \dots A_n$. Then interchange $A_2 \leftrightarrow A_3$, $A_4 \leftrightarrow A_5$, $A_6 \leftrightarrow A_7$, ..., and return to Z2. ∎

Hint: Show first that the algorithm works if modified so that $A_j \leftarrow n + 1 - j$ and $a_j \leftarrow j$ in step Z1, and if the "flip" permutations

$$\rho' = (1\ \ n)(2\ \ n{-}1)\dots, \qquad \sigma' = (2\ \ n)(3\ \ n{-}1)\dots, \qquad \tau' = (2\ \ n{-}1)(3\ \ n{-}2)\dots$$

are used instead of ρ, σ, τ in steps Z4 and Z5. In this modification, step Z3 should go to Z5 if $A_1 = x$ and $A_n = y$.

▸ **70.** [*M33*] The two 12-cycles (41) can be regarded as σ–τ cycles for the twelve permutations of $\{1, 1, 3, 4\}$:

$$1134 \to 1341 \to 3411 \to 4311 \to 3114 \to 1143 \to 1431 \\ \to 4131 \to 1314 \to 3141 \to 1413 \to 4113 \to 1134.$$

Replacing $\{1, 1\}$ by $\{1, 2\}$ yields disjoint cycles, and we obtained a Hamiltonian path by jumping from one to the other. Can a σ–τ path for all permutations of 6 elements be formed in a similar way, based on a 360-cycle for the permutations of $\{1, 1, 3, 4, 5, 6\}$?

71. [*M21*] Given a Cayley graph with generators $(\alpha_1, \dots, \alpha_k)$, assume that each α_j takes $x \mapsto y$. (For example, both σ and τ take $1 \mapsto 2$ when $\sigma = (1\,2\,\dots\,n)$ and $\tau = (1\,2)$.) Prove that any Hamiltonian path starting at $12 \dots n$ in G must end at a permutation that takes $y \mapsto x$.

▸ **72.** [*M30*] Let α, β, and σ be permutations of a set X, where $X = A \cup B$. Assume that $x\sigma = x\alpha$ when $x \in A$ and $x\sigma = x\beta$ when $x \in B$, and that the order of $\alpha\beta^-$ is odd.

a) Prove that all three permutations α, β, σ have the same sign; that is, they are all even or all odd. *Hint:* A permutation has odd order if and only if its cycles all have odd length.

b) Derive Theorem R from part (a).

73. [*M30*] (R. A. Rankin.) Assuming that $\alpha\beta = \beta\alpha$ in Theorem R, prove that a Hamiltonian circuit exists if and only if there is a number k such that $0 \le k \le g/c$ and $t + k \perp c$, where $\beta^{g/c} = \gamma^t$, $\gamma = \alpha\beta^-$. *Hint:* Represent elements of the group in the form $\beta^j\gamma^k$.

74. [*M25*] The directed torus $C_m \times C_n$ has mn vertices (x, y) for $0 \le x < m$, $0 \le y < n$, and arcs $(x, y) \to (x, y)\alpha = \big((x+1) \bmod m, y\big)$, $(x, y) \to (x, y)\beta = \big(x, (y+1) \bmod n\big)$. Prove that, if $m > 1$ and $n > 1$, the number of Hamiltonian circuits of this digraph is

$$\sum_{k=1}^{d-1} \binom{d}{k} [\gcd((d-k)m, kn) = d], \qquad d = \gcd(m, n).$$

75. [*M31*] The cells numbered $0, 1, \ldots, 63$ in Fig. 25 illustrate a *northeasterly knight's tour* on an 8×8 torus: If k appears in cell (x_k, y_k), then $(x_{k+1}, y_{k+1}) = (x_k + 2, y_k + 1)$ or (x_k+1, y_k+2), modulo 8, and $(x_{64}, y_{64}) = (x_0, y_0)$. How many such tours are possible on an $m \times n$ torus, when $m, n \ge 3$?

29	24	19	14	49	44	39	34
58	53	48	43	38	9	4	63
23	18	13	8	3	62	33	28
52	47	42	37	32	27	22	57
17	12	7	2	61	56	51	46
6	41	36	31	26	21	16	11
35	30	1	60	55	50	45	40
0	59	54	25	20	15	10	5

Fig. 25. A northeasterly knight's tour.

▸ **76.** [*22*] Complete the `MMIX` program whose inner loop appears in (42), using Heap's method (27).

77. [*M23*] Analyze the running time of the program in exercise 76, generalizing it so that the inner loop does $r!$ visits (with $a_0 \ldots a_{r-1}$ in global registers).

78. [*20*] What seven `MMIX` instructions will ⟨ Swap the nybbles ... ⟩ as (45) requires? For example, if register `t` contains the value 4 and register `a` contains the nybbles `#12345678`, register `a` should change to `#12345687`.

79. [*21*] Solve the previous exercise with only five `MMIX` instructions. *Hint:* Use `MXOR`.

▸ **80.** [*22*] Complete the `MMIX` program (46) by specifying how to ⟨ Continue with Langdon's method ⟩.

81. [*M21*] Analyze the running time of the program in exercise 80.

82. [*22*] Use the σ–τ path of exercise 70 to design an `MMIX` routine analogous to (42) that generates all permutations of `#123456` in register `a`.

83. [*20*] Suggest a good way to generate all $n!$ permutations of $\{1, \ldots, n\}$ on p processors that are running in parallel.

▸ **84.** [*25*] Assume that n is small enough that $n!$ fits in a computer word. What's a good way to convert a given permutation $\alpha = a_1 \ldots a_n$ of $\{1, \ldots, n\}$ into an integer $k = r(\alpha)$ in the range $0 \le k < n!$? Both functions $k = r(\alpha)$ and $\alpha = r^{[-1]}(k)$ should be computable in only $O(n)$ steps.

85. [*20*] A partial order relation is supposed to be transitive; that is, $x \prec y$ and $y \prec z$ should imply $x \prec z$. But Algorithm V does not require its input relation to satisfy this condition.

Show that if $x \prec y$ and $y \prec z$, Algorithm V will produce identical results whether or not $x \prec z$.

86. [*20*] (F. Ruskey.) Consider the inversion tables $c_1 \ldots c_n$ of the permutations visited by Algorithm V. What noteworthy property do they have? (Compare with the inversion tables (4) in Algorithm P.)

87. [*21*] Show that Algorithm V can be used to generate all ways to partition the digits $\{0, 1, \ldots, 9\}$ into two 3-element sets and two 2-element sets.

▸ **88.** [*M30*] Consider the numbers $t_0, t_1, \ldots, t_n$ in (51). Clearly $t_0 = t_1 = 1$.

a) Say that index j is "trivial" if $t_j = t_{j-1}$. For example, 9 is trivial with respect to the Young tableau relations (48). Explain how to modify Algorithm V so that the variable k takes on only nontrivial values.

b) Analyze the running time of the modified algorithm. What formulas replace (51)?

c) Say that the interval $[j\,..\,k]$ is not a chain if we do not have $l \prec l+1$ for $j \le l < k$. Prove that in such a case $t_k \ge 2t_{j-1}$.

d) Every inverse topological sort $a'_1 \ldots a'_n$ defines a labeling that corresponds to relations $a'_{j_1} \prec a'_{k_1}, \ldots, a'_{j_m} \prec a'_{k_m}$, which are equivalent to the original relations $j_1 \prec k_1, \ldots, j_m \prec k_m$. Explain how to find a labeling such that $[j\,..\,k]$ is not a chain when j and k are consecutive nontrivial indices.

e) Prove that with such a labeling, $M < 4N$ in the formulas of part (b).

89. [*M21*] Algorithm V can be used to produce all permutations that are h-ordered for all h in a given set, namely all $a'_1 \ldots a'_n$ such that $a'_j < a'_{j+h}$ for $1 \le j \le n - h$ (see Section 5.2.1). Analyze the running time of Algorithm V when it generates all permutations that are both 2-ordered and 3-ordered.

90. [*HM21*] Analyze the running time of Algorithm V when it is used with the relations (49) to find matchings.

91. [*M18*] How many permutations is Algorithm V likely to visit, in a "random" case? Let P_n be the number of partial orderings on $\{1, \ldots, n\}$, namely the number of relations that are reflexive, antisymmetric, and transitive. Let Q_n be the number of such relations with the additional property that $j < k$ whenever $j \prec k$. Express the expected number of ways to sort n elements topologically, averaged over all partial orderings, in terms of P_n and Q_n.

92. [*35*] Prove that all topological sorts can be generated in such a way that only one or two adjacent transpositions are made at each step. (The example $1 \prec 2$, $3 \prec 4$ shows that a single transposition per step cannot always be achieved, even if we allow nonadjacent swaps, because only two of the six relevant permutations are odd.)

▸ **93.** [*25*] Show that in the case of matchings, using the relations in (49), all topological sorts can be generated with just one transposition per step.

94. [*21*] Discuss how to generate all *up-down permutations* of $\{1, \ldots, n\}$, namely those $a_1 \ldots a_n$ such that $a_1 < a_2 > a_3 < a_4 > \cdots$.

95. [*21*] Discuss how to generate all *cyclic permutations* of $\{1, \ldots, n\}$, namely those $a_1 \ldots a_n$ whose cycle representation consists of a single n-cycle.

96. [*21*] Discuss how to generate all *derangements* of $\{1, \ldots, n\}$, namely those $a_1 \ldots a_n$ such that $a_1 \ne 1$, $a_2 \ne 2$, $a_3 \ne 3$, ...

97. [*HM23*] Analyze the asymptotic running time of the method in the previous exercise.

98. [*M30*] Given $n \ge 3$, show that all derangements of $\{1, \ldots, n\}$ can be generated by making at most two transpositions between visits.

99. [*21*] Discuss how to generate all *indecomposable* permutations of $\{1, \ldots, n\}$, namely those $a_1 \ldots a_n$ such that $\{a_1, \ldots, a_j\} \ne \{1, \ldots, j\}$ for $1 \le j < n$.

100. [*21*] Discuss how to generate all *involutions* of $\{1, \ldots, n\}$, namely those permutations $a_1 \ldots a_n$ with $a_{a_1} \ldots a_{a_n} = 1 \ldots n$.

101. [*M30*] Show that all involutions of $\{1,\ldots,n\}$ can be generated by making at most two transpositions between visits.

102. [*M32*] Show that all even permutations of $\{1,\ldots,n\}$ can be generated by successive *rotations of three consecutive elements*.

▸ **103.** [*M22*] A permutation $a_1\ldots a_n$ of $\{1,\ldots,n\}$ is *well-balanced* if

$$\sum_{k=1}^{n} k a_k = \sum_{k=1}^{n} (n+1-k)a_k.$$

For example, 3142 is well-balanced when $n=4$.

a) Prove that no permutation is well-balanced when $n \bmod 4 = 2$.

b) Prove that if $a_1\ldots a_n$ is well-balanced, so are its reversal $a_n\ldots a_1$, its complement $(n+1-a_1)\ldots(n+1-a_n)$, and its inverse $a'_1\ldots a'_n$.

c) Determine the number of well-balanced permutations for small values of n.

▸ **104.** [*26*] A *weak order* is a relation $\preceq$ that is transitive ($x\preceq y$ and $y\preceq z$ implies $x\preceq z$) and complete ($x\preceq y$ or $y\preceq x$ always holds). We can write $x\equiv y$ if $x\preceq y$ and $y\preceq x$; $x\prec y$ if $x\preceq y$ and $y\not\preceq x$. There are thirteen weak orders on three elements $\{1,2,3\}$, namely

$$1\equiv 2\equiv 3,\quad 1\equiv 2\prec 3,\quad 1\prec 2\equiv 3,\quad 1\prec 2\prec 3,\quad 1\equiv 3\prec 2,\quad 1\prec 3\prec 2,$$
$$2\prec 1\equiv 3,\quad 2\prec 1\prec 3,\quad 2\equiv 3\prec 1,\quad 2\prec 3\prec 1,\quad 3\prec 1\equiv 2,\quad 3\prec 1\prec 2,\quad 3\prec 2\prec 1.$$

a) Explain how to generate all weak orders of $\{1,\ldots,n\}$ systematically, as sequences of digits separated by the symbols $\equiv$ or $\prec$.

b) A weak order can also be represented as a sequence $a_1\ldots a_n$ where $a_j=k$ if j is preceded by k $\prec$ signs. For example, the thirteen weak orders on $\{1,2,3\}$ are respectively 000, 001, 011, 012, 010, 021, 101, 102, 100, 201, 110, 120, 210 in this form. Find a simple way to generate all such sequences of length n.

105. [*M40*] Can exercise 104(b) be solved with a Gray-like path?

▸ **106.** [*30*] (John H. Conway, 1973.) To play the solitaire game of "topswops," start by shuffling a pack of n cards labeled $\{1,\ldots,n\}$ and place them face up in a pile. Then if the top card is $k>1$, deal out the top k cards and put them back on top of the pile, thereby changing the permutation from $a_1\ldots a_n$ to $a_k\ldots a_1 a_{k+1}\ldots a_n$. Continue until the top card is 1. For example, the 7-step sequence

$$31452 \to 41352 \to 53142 \to 24135 \to 42135 \to 31245 \to 21345 \to 12345$$

might occur when $n=5$. What is the longest sequence possible when $n=13$?

107. [*M27*] If the longest n-card game of topswops has length $f(n)$, prove that $f(n)\le F_{n+1}-1$.

108. [*M47*] Find good upper and lower bounds on the topswops function $f(n)$.

▸ **109.** [*25*] Find all permutations $a_0\ldots a_9$ of $\{0,\ldots,9\}$ such that

$$\{a_0,a_2,a_3,a_7\}=\{2,5,7,8\},$$
$$\{a_1,a_4,a_5\}=\{0,3,6\},$$
$$\{a_1,a_3,a_7,a_8\}=\{3,4,5,7\},$$
$$\{a_0,a_3,a_4\}=\{0,7,8\}.$$

Also suggest an algorithm for solving large problems of this type.

▸ **110.** [*M25*] Several permutation-oriented analogs of de Bruijn cycles have been proposed. The simplest and nicest of these is the notion of a *universal cycle of permutations*, introduced by B. W. Jackson in *Discrete Math.* **117** (1993), 141–150, namely a cycle of $n!$ digits such that each permutation of $\{1,\ldots,n\}$ occurs exactly once as a block of $n-1$ consecutive digits (with its redundant final element suppressed). For example, (121323) is a universal cycle of permutations for $n=3$, and it is essentially the only such cycle.

Find a universal cycle of permutations for $n=4$, and prove that such cycles exist for all $n\geq 2$.

111. [*M46*] Exactly how many universal cycles exist, for permutations of order n?

SECTION 7.2.1.2

1. [J. P. N. Phillips, *Comp. J.* **10** (1967), 311.] Assuming that $n \ge 3$, we can replace steps L2–L4 by:

L2′. [Easiest case?] Set $y \leftarrow a_{n-1}$ and $z \leftarrow a_n$. If $y < z$, set $a_{n-1} \leftarrow z$, $a_n \leftarrow y$, and return to L1.

L2.1′. [Next easiest case?] Set $x \leftarrow a_{n-2}$. If $x \ge y$, go on to step L2.2′. Otherwise set $(a_{n-2}, a_{n-1}, a_n) \leftarrow (z, x, y)$ if $x < z$, (y, z, x) if $x \ge z$. Return to L1.

L2.2′. [Find j.] Set $j \leftarrow n-3$ and $y \leftarrow a_j$. If $y \ge x$, set $j \leftarrow j-1$, $x \leftarrow y$, $y \leftarrow a_j$, and repeat until $y < x$. Terminate if $j = 0$.

L3′. [Easy increase?] If $y < z$, set $a_j \leftarrow z$, $a_{j+1} \leftarrow y$, $a_n \leftarrow x$, and go to L4.1′.

L3.1′. [Increase a_j.] Set $l \leftarrow n-1$; if $y \ge a_l$, repeatedly decrease l by 1 until $y < a_l$. Then set $a_j \leftarrow a_l$ and $a_l \leftarrow y$.

L4′. [Begin to reverse.] Set $a_n \leftarrow a_{j+1}$ and $a_{j+1} \leftarrow z$.

L4.1′. [Reverse $a_{j+1} \ldots a_{n-1}$.] Set $k \leftarrow j+2$, $l \leftarrow n-1$. Then, if $k < l$, interchange $a_k \leftrightarrow a_l$, set $k \leftarrow k+1$, $l \leftarrow l-1$, and repeat until $k \ge l$. Return to L1. ∎

The program might run still faster if a_t is stored in memory location $\mathtt{A}[n-t]$ for $0 \le t \le n$, or if reverse colex order is used as in the following exercise.

2. Again we assume that $a_1 \le a_2 \le \cdots \le a_n$ initially; the permutations generated from $\{1, 2, 2, 3\}$ will, however, be 1223, 2123, 2213, ..., 2321, 3221. Let a_{n+1} be an auxiliary element, *larger* than a_n.

L1. [Visit.] Visit the permutation $a_1 a_2 \ldots a_n$.

L2. [Find j.] Set $j \leftarrow 2$. If $a_{j-1} \ge a_j$, increase j by 1 until $a_{j-1} < a_j$. Terminate if $j > n$.

L3. [Decrease a_j.] Set $l \leftarrow 1$. If $a_l \ge a_j$, increase l until $a_l < a_j$. Then swap $a_l \leftrightarrow a_j$.

L4. [Reverse $a_1 \ldots a_{j-1}$.] Set $k \leftarrow 1$ and $l \leftarrow j-1$. Then, if $k < l$, swap $a_k \leftrightarrow a_l$, set $k \leftarrow k+1$, $l \leftarrow l-1$, and repeat until $k \ge l$. Return to L1. ∎

3. Let $C_1 \ldots C_n = c_{a_1} \ldots c_{a_n}$ be the inversion table, as in exercise 5.1.1–7. Then $\operatorname{rank}(a_1 \ldots a_n)$ is the mixed-radix number $\left[\begin{smallmatrix} C_n, & \ldots, & C_2, & C_1 \\ n, & \ldots, & 2, & 1 \end{smallmatrix}\right]$. [See H. A. Rothe, *Sammlung combinatorisch-analytischer Abhandlungen* **2** (1800), 263–264.] For example, 314592687 has rank $\left[\begin{smallmatrix} 2, & 0, & 1, & 1, & 4, & 0, & 0, & 1, & 0 \\ 9, & 8, & 7, & 6, & 5, & 4, & 3, & 2, & 1 \end{smallmatrix}\right] = 2 \cdot 8! + 6! + 5! + 4 \cdot 4! + 1! = 81577$; this is the factorial number system featured in Eq. 4.1–(10).

4. Use the recurrence $\operatorname{rank}(a_1 \ldots a_n) = \frac{1}{n} \sum_{j=1}^{t} n_j [x_j < a_1] \binom{n}{n_1, \ldots, n_t} + \operatorname{rank}(a_2 \ldots a_n)$. For example, rank(314159265) is

$$\tfrac{3}{9}\tbinom{9}{2,1,1,1,2,1,1} + 0 + \tfrac{2}{7}\tbinom{7}{1,1,1,2,1,1} + 0 + \tfrac{1}{5}\tbinom{5}{1,2,1,1} + \tfrac{3}{4}\tbinom{4}{1,1,1,1} + 0 + \tfrac{1}{2}\tbinom{2}{1,1} = 30991.$$

5. (a) Step L2 is performed $n!$ times. The probability that exactly k comparisons are made is $q_k - q_{k+1}$, where q_t is the probability that $a_{n-t+1} > \cdots > a_n$, namely $[t \le n]/t!$. Therefore the mean is $\sum k(q_k - q_{k+1}) = q_1 + \cdots + q_n = \lfloor n!\, e \rfloor / n! - 1 \approx e - 1 \approx 1.718$, and the variance is

$$\sum k^2(q_k - q_{k+1}) - \text{mean}^2 = q_1 + 3q_2 + \cdots + (2n-1)q_n - (q_1 + \cdots + q_n)^2 \approx e(3-e) \approx 0.766.$$

[For higher moments, see R. Kemp, *Acta Informatica* **35** (1998), 17–89, Theorem 4.]

Incidentally, the average number of interchange operations in step L4 is therefore $\sum\lfloor k/2\rfloor(q_k - q_{k+1}) = q_2 + q_4 + \cdots \approx \cosh 1 - 1 = (e + e^{-1} - 2)/2 \approx 0.543$, a result due to R. J. Ord-Smith [*Comp. J.* **13** (1970), 152–155].

(b) Step L3 is performed only $n! - 1$ times, but we will assume for convenience that it occurs once more (with 0 comparisons). Then the probability that exactly k comparisons are made is $\sum_{j=k+1}^{n} 1/j!$ for $1 \le k < n$ and $1/n!$ for $k = 0$. Hence the mean is $\frac{1}{2}\sum_{j=0}^{n-2} 1/j! \approx e/2 \approx 1.359$; exercise 1 reduces this number by $\frac{2}{3}$. The variance is $\frac{1}{3}\sum_{j=0}^{n-3} 1/j! + \frac{1}{2}\sum_{j=0}^{n-2} 1/j! - \text{mean}^2 \approx \frac{5}{6}e - \frac{1}{4}e^2 \approx 0.418$.

6. (a) Let $e_n(z) = \sum_{k=0}^{n} z^k/k!$; then the number of different prefixes $a_1 \ldots a_j$ is $j!\,[z^j]\,e_{n_1}(z)\ldots e_{n_t}(z)$. This is $N = \binom{n}{n_1,\ldots,n_t}$ times the probability q_{n-j} that at least $n-j$ comparisons are made in step L2. Therefore the mean is $\frac{1}{N}w\big(e_{n_1}(z)\ldots e_{n_t}(z)\big) - 1$, where $w\big(\sum x_k z^k/k!\big) = \sum x_k$. In the binary case the mean is $M/\binom{n}{m} - 1$, where $M = \sum_{l=0}^{m}\sum_{k=l}^{n-m+l}\binom{k}{l} = \sum_{l=0}^{m}\binom{n-m+l+1}{l+1} = \binom{n+2}{m+1} - 1 = \binom{n}{m}\big(2 + \frac{m}{n-m+1} + \frac{n-m}{m+1}\big) - 1$.

(b) If $\{a_1, \ldots, a_j\} = \{n'_1 \cdot x_1, \ldots, n'_t \cdot x_t\}$, the prefix $a_1 \ldots a_j$ contributes altogether $\sum_{1\le k<l\le t}(n_k - n'_k)[n_l < n'_l]$ to the total number of comparisons made in step L3. Thus the mean is $\frac{1}{N}\sum_{1\le k<l\le t} w\big(f_{kl}(z)\big)$, where

$$f_{kl}(z) = \Bigg(\prod_{\substack{1\le m\le t\\ m\ne k,\ m\ne l}} e_{n_m}(z)\Bigg)\Bigg(\sum_{r=0}^{n_k}(n_k - r)\frac{z^r}{r!}\Bigg)e_{n_l-1}(z)$$

$$= n_k\Bigg(\prod_{\substack{1\le m\le t\\ m\ne l}} e_{n_m}(z)\Bigg)e_{n_l-1}(z) \;-\; z\Bigg(\prod_{\substack{1\le m\le t\\ m\ne k,\ m\ne l}} e_{n_m}(z)\Bigg)e_{n_l-1}(z)e_{n_k-1}(z).$$

In the two-valued case this formula reduces to $\frac{1}{N}w\big(me_m(z) - ze_{m-1}(z)\big)e_{n-m-1}(z)\big) = \frac{m}{N}\big(\binom{n+1}{m+1} - 1\big) - \frac{1}{N}\big(\binom{n+1}{m+1}\big(m - \frac{m+1}{n-m+1}\big) + 1\big) = \frac{1}{N}\big(-m - 1 + \binom{n+1}{m}\big) = \frac{n+1}{n-m+1} - \frac{m+1}{N}$.

7. In the notation of the previous answer, the quantity $\frac{1}{N}w\big(e_{n_1}(z)\ldots e_{n_t}(z)\big) - 1$ is

$$\frac{n_1 + \cdots + n_t}{n} + \frac{(n_1n_2 + n_1n_3 + \cdots + n_{t-1}n_t) + n_1(n_1-1) + \cdots + n_t(n_t-1)}{n(n-1)} + \cdots - 1.$$

One can show using Eq. 1.2.9–(38) that the limit is $-1 + \exp\sum_{k\ge1} r_k/k$, where $r_k = \lim_{t\to\infty}(n_1^k + \cdots + n_t^k)/(n_1 + \cdots + n_t)^k$. In cases (a) and (b) we have $r_k = [k=1]$, so the limit is $e - 1 \approx 1.71828$. In case (c) we have $r_k = 1/(2^k - 1)$, so the limit is $-1 + \exp\sum_{k\ge1} 1/(k(2^k - 1)) \approx 2.46275$.

8. Assume that j is initially zero, and change step L1 to

L1′. [Visit.] Visit the variation $a_1 \ldots a_j$. If $j < n$, set $j \leftarrow j + 1$ and repeat this step. ∎

This algorithm is due to L. J. Fischer and K. C. Krause, *Lehrbuch der Combinationslehre und der Arithmetik* (Dresden: 1812), 55–57.

Incidentally, the total number of variations is $w\big(e_{n_1}(z)\ldots e_{n_t}(z)\big)$ in the notation of answer 6. This counting problem was first treated by Jakob Bernoulli in *Ars Conjectandi* (1713), Part 2, Chapter 9.

9. **V1.** [Visit.] Visit the variation $a_1 \ldots a_r$. (At this point $a_{r+1} \le \cdots \le a_n$.)

V2. [Easy case?] If $a_r < a_n$, interchange $a_r \leftrightarrow a_j$ where j is the smallest subscript such that $j > r$ and $a_j > a_r$, and return to V1.

V3. [Reverse.] Set $(a_{r+1},\ldots,a_n) \leftarrow (a_n,\ldots,a_{r+1})$ as in step L4.

V4. [Find j.] Set $j \leftarrow r-1$. If $a_j \ge a_{j+1}$, decrease j by 1 repeatedly until $a_j < a_{j+1}$. Terminate if $j = 0$.

V5. [Increase a_j.] Set $l \leftarrow n$. If $a_j \ge a_l$, decrease l by 1 repeatedly until $a_j < a_l$. Then interchange $a_j \leftrightarrow a_l$.

V6. [Reverse again.] Set $(a_{j+1},\ldots,a_n) \leftarrow (a_n,\ldots,a_{j+1})$ as in step L4, and return to V1. ▮

The number of outputs is $r!\,[z^r]\,e_{n_1}(z)\ldots e_{n_t}(z)$; this is, of course, n^r when the elements are distinct.

10. $a_1a_2\ldots a_n = 213\ldots n$, $c_1c_2\ldots c_n = 010\ldots0$, $d_1d_2\ldots d_n = 1(-1)1\ldots1$, if $n \ge 2$.

11. Step (P1, ..., P7) is performed $(1, n!, n!, n!+x_n, n!, (x_n+3)/2, x_n)$ times, where $x_n = \sum_{k=1}^{n-1} k!$, because P7 is performed $(j-1)!$ times when $2 \le j \le n$.

12. We want the permutation of rank 999999. The answers are (a) 2783915460, by exercise 3; (b) 8750426319, because the reflected mixed-radix number corresponding to $\left[\begin{smallmatrix}0,&0,&1,&2,&3,&0,&2,&7,&0,&9\\1,&2,&3,&4,&5,&6,&7,&8,&9,&10\end{smallmatrix}\right]$ is $\left[\begin{smallmatrix}0,&0,&1,&3-2,&3,&5-0,&2,&7,&8-0,&9-9\\1,&2,&3,&4,&5,&6,&7,&8,&9,&10\end{smallmatrix}\right]$ by 7.2.1.1–(50); (c) the product $(0\,1\,\ldots\,9)^9(0\,1\,\ldots\,8)^0(0\,1\,\ldots\,7)^7(0\,1\,\ldots\,6)^2\ldots(0\,1\,2)^1$, namely 9703156248.

13. The first statement is true for all $n \ge 2$. But when 2 crosses 1, namely when c_2 changes from 0 to 1, we have $c_3 = 2$, $c_4 = 3$, $c_5 = \cdots = c_n = 0$, and the next permutation when $n \ge 5$ is $432156\ldots n$. [See *Time Travel* (1988), page 74.]

14. True at the beginning of steps P4, P5, and P6, because exactly $j-1-c_j+s$ elements lie to the left of x_j, namely $j-1-c_j$ from $\{x_1,\ldots,x_{j-1}\}$ and s from $\{x_{j+1},\ldots,x_n\}$. (In a sense, this formula is the main point of Algorithm P.)

15. If $\left[\begin{smallmatrix}b_{n-1},&\ldots,&b_0\\1,&\ldots,&n\end{smallmatrix}\right]$ corresponds to the reflected Gray code $\left[\begin{smallmatrix}c_1,&\ldots,&c_n\\1,&\ldots,&n\end{smallmatrix}\right]$, we get to step P6 if and only if $b_k = k-1$ for $j \le k \le n$ and B_{n-j+1} is even, by 7.2.1.1–(50). But $b_{n-k} = k-1$ for $j \le k \le n$ implies that B_{n-k} is odd for $j < k \le m$. Therefore $s = [c_{j+1} = j] + [c_{j+2} = j+1] = [d_{j+1} < 0] + [d_{j+2} < 0]$ in step P5. [See *Math. Comp.* **17** (1963), 282–285.]

16. **P1′.** [Initialize.] Set $c_j \leftarrow j$ and $d_j \leftarrow -1$ for $1 \le j < n$; also set $z \leftarrow a_n$.

P2′. [Visit.] Visit $a_1\ldots a_n$. Then go to P3.5′ if $a_1 = z$.

P3′. [Hunt down.] For $j \leftarrow n-1, n-2, \ldots, 1$ (in this order), set $a_{j+1} \leftarrow a_j$, $a_j \leftarrow z$, and visit $a_1\ldots a_n$. Then set $j \leftarrow n-1$, $s \leftarrow 1$, and go to P4′.

P3.5′. [Hunt up.] For $j \leftarrow 1, 2, \ldots, n-1$ (in this order), set $a_j \leftarrow a_{j+1}$, $a_{j+1} \leftarrow z$, and visit $a_1\ldots a_n$. Then set $j \leftarrow n-1$, $s \leftarrow 0$.

P4′. [Ready to change?] Set $q \leftarrow c_j + d_j$. If $q = 0$, go to P6′; if $q > j$, go to P7′.

P5′. [Change.] Interchange $a_{c_j+s} \leftrightarrow a_{q+s}$. Then set $c_j \leftarrow q$ and return to P2′.

P6′. [Increase s.] Terminate if $j = 1$; otherwise set $s \leftarrow s+1$.

P7′. [Switch direction.] Set $d_j \leftarrow -d_j$, $j \leftarrow j-1$, and go back to P4′. ▮

17. Initially $a_j \leftarrow a'_j \leftarrow j$ for $1 \le j \le n$. Step P5 should now set $t \leftarrow j - c_j + s$, $u \leftarrow j-q+s$, $v \leftarrow a_u$, $a_t \leftarrow v$, $a'_v \leftarrow t$, $a_u \leftarrow j$, $a'_j \leftarrow u$, $c_j \leftarrow q$. (See exercise 14.)

But with the inverse required and available we can actually simplify the algorithm significantly, avoiding the offset variable s and letting the control table $c_1\ldots c_n$ count only downwards, as noted by G. Ehrlich [*JACM* **20** (1973), 505–506]:

Q1. [Initialize.] Set $a_j \leftarrow a'_j \leftarrow j$, $c_j \leftarrow j-1$, and $d_j \leftarrow -1$ for $1 \le j \le n$. Also set $c_0 = -1$.

Q2. [Visit.] Visit the permutation $a_1 \dots a_n$ and its inverse $a'_1 \dots a'_n$.

Q3. [Find k.] Set $k \leftarrow n$. Then if $c_k = 0$, set $c_k \leftarrow k-1$, $d_k \leftarrow -d_k$, $k \leftarrow k-1$, and repeat until $c_k \neq 0$. Terminate if $k = 0$.

Q4. [Change.] Set $c_k \leftarrow c_k - 1$, $j \leftarrow a'_k$, and $i = j + d_k$. Then set $t \leftarrow a_i$, $a_i \leftarrow k$, $a_j \leftarrow t$, $a'_t \leftarrow j$, $a'_k \leftarrow i$, and return to Q2. ∎

18. Set $a_n \leftarrow n$, and use $(n-1)!/2$ iterations of Algorithm P to generate all permutations of $\{1, \dots, n-1\}$ such that 1 precedes 2. [M. K. Roy, *CACM* **16** (1973), 312–313; see also exercise 13.]

19. For example, we can use the idea of Algorithm P, with the n-tuples $c_1 \dots c_n$ changing as in Algorithm 7.2.1.1H with respect to the radices $(1, 2, \dots, n)$. That algorithm maintains the directions correctly, although it numbers subscripts differently. The offset s needed by Algorithm P can be computed as in the answer to exercise 15, or the inverse permutation can be maintained as in exercise 17. [See G. Ehrlich, *CACM* **16** (1973), 690–691.] Other algorithms, like that of Heap, can also be implemented looplessly.

(*Note:* In most applications of permutation generation we are interested in minimizing the *total* running time, not the maximum time between successive visits; from this standpoint looplessness is usually undesirable, except on a parallel computer. Yet there's something intellectually satisfying about the fact that a loopless algorithm exists, whether practical or not.)

20. For example, when $n = 3$ we can begin 123, 132, 312, $\overline{3}12$, $1\overline{3}2$, $12\overline{3}$, $21\overline{3}$, ..., 213, $\overline{2}13$, ... If the delta sequence for n is $(\delta_1\delta_2 \dots \delta_{2^n n!})$, the corresponding sequence for $n+1$ is $(\Delta_n\delta_1\Delta_n\delta_2 \dots \Delta_n\delta_{2^n n!})$, where Δ_n is the sequence of $2n-1$ operations $n\ n{-}1\ \dots\ 1\ -\ 1\ \dots\ n{-}1\ n$; here $\delta_k = j$ means $a_j \leftrightarrow a_{j+1}$ and $\delta_k = -$ means $a_1 \leftarrow -a_1$.

(Signed permutations appear in another guise in exercises 5.1.4–43 and 44. The set of all signed permutations is called the octahedral group.)

21. Clearly $\texttt{M} = 1$, hence $\texttt{O}$ must be 0 and $\texttt{S}$ must be $b-1$. Then $\texttt{N} = \texttt{E}+1$, $\texttt{R} = b-2$, and $\texttt{D}+\texttt{E} = b+\texttt{Y}$. This leaves exactly $\max(0, b-7-k)$ choices for $\texttt{E}$ when $\texttt{Y} = k \geq 2$, hence a total of $\sum_{k=2}^{b-7}(b-7-k) = \binom{b-8}{2}$ solutions when $b \geq 8$. [*Math. Mag.* **45** (1972), 48–49.]

22. $(\texttt{XY})_b + (\texttt{XX})_b = (\texttt{XYX})_b$ is solvable only when $b = 2$.

23. Almost true, because the number of solutions will be even, *unless* $[j \in F] \neq [k \in F]$. (Consider the ternary alphametic $\texttt{X} + (\texttt{XX})_3 + (\texttt{YY})_3 + (\texttt{XZ})_3 = (\texttt{XYX})_3$.)

24. (a) $9283 + 7 + 473 + 1062 = 10825$. (b) $698392 + 3192 = 701584$. (c) $63952 + 69275 = 133227$. (d) $653924 + 653924 = 1307848$. (e) $5718 + 3 + 98741 = 104462$. (f) $127503 + 502351 + 3947539 + 46578 = 4623971$. (g) $67432 + 704 + 8046 + 97364 = 173546$. (h) $59 + 577404251698 + 69342491650 + 49869442698 + 1504 + 40614 + 82591 + 344 + 41 + 741425 = 5216367650 + 691400684974$. [All solutions are unique. References for (b)–(g): *J. Recreational Math.* **10** (1977), 155; **5** (1972), 296; **10** (1977), 41; **10** (1978), 274; **12** (1979), 133–134; **9** (1977), 207.]

(i) In this case there are $\frac{8}{10}10! = 2903040$ solutions, because *every* permutation of $\{0, 1, \dots, 9\}$ works except those that assign $\texttt{H}$ or $\texttt{N}$ to 0. (A well-written general additive alphametic solver will be careful to reduce the amount of output in such cases.)

25. We may assume that $s_1 \leq \cdots \leq s_{10}$. Let i be the least index $\notin F$, and set $a_i \leftarrow 0$; then set the remaining elements a_j in order of increasing j. A proof like that

of Theorem 6.1S shows that this procedure maximizes $a \cdot s$. A similar procedure yields the minimum, because $\min(a \cdot s) = -\max(a \cdot (-s))$.

26. 400739 + 63930 − 2379 − 1252630 + 53430 − 1390 + 738300.

27. Readers can probably improve upon the following examples: BLOOD + SWEAT + TEARS = LATER; EARTH + WATER + WRATH = HELLO + WORLD; AWAIT + ROBOT + ERROR = SOBER + WORDS; CHILD + THEME + PEACE + ETHIC = IDEAL + ALPHA + METIC. (This exercise was inspired by WHERE + SEDGE + GRASS + GROWS = MARSH [A. W. Johnson, Jr., *J. Recr. Math.* **15** (1982), 51], which would be marvelously pure except that D and O have the same signature.)

28. (a) 11 = 3 + 3 + 2 + 2 + 1, 20 = 11 + 3 + 3 + 3, 20 = 11 + 3 + 3 + 2 + 1, 20 = 11+3+3+1+1+1, 20 = 8+8+2+1+1, 20 = 7+7+6, 20 = 7+7+2+2+2, 20 = 7+7+2+1+1+1+1, 20 = 7+5+5+2+1, 20 = 7+5+2+2+2+1+1, 20 = 7+5+2+2+1+1+1+1, 20 = 7+3+3+2+2+1+1+1, 20 = 7+3+3+1+1+1+1+1+1, 20 = 5+3+3+3+3+3. [These fourteen solutions were first computed by Roy Childs in 1999. The next doubly partitionable values of n are 30 (in 20 ways), then 40 (in 94 ways), 41 (in 67), 42 (in 57), 50 (in 190 ways, including $50 = 2 + 2 + \cdots + 2$), etc.]

(b) 51 = 20 + 15 + 14 + 2, 51 = 15 + 14 + 10 + 9 + 3, 61 = 19 + 16 + 11 + 9 + 6, 65 = 17 + 16 + 15 + 9 + 7 + 1, 66 = 20 + 19 + 16 + 6 + 5, 69 = 18 + 17 + 16 + 10 + 8, 70 = 30+20+10+7+3, 70 = 20+16+12+9+7+6, 70 = 20+15+12+11+7+5, 80 = 50+20+9+1, 90 = 50+12+11+9+5+2+1, 91 = 45+19+11+10+5+1. [The two 51s are due to Steven Kahan; see his book *Have Some Sums To Solve* (Farmingdale, New York: Baywood, 1978), 36–37, 84, 112. Amazing examples with seventeen distinct terms in Italian and fifty-eight distinct terms in Roman numerals have been found by Giulio Cesare, *J. Recr. Math.* **30** (1999), 63.]

Notes: The beautiful example THREE = TWO + ONE + ZERO [Richard L. Breisch, *Recreational Math. Magazine* **12** (December 1962), 24] is unfortunately ruled out by our conventions. The total number of doubly true partitions into distinct parts is probably finite, in English, although nomenclature for arbitrary large integers is not standard. Is there an example that exceeds NINETYNINENONILLIONNINETYNINETRILLIONNINETYONE = NINETYNINENONILLIONNINETYNINETRILLIONFORTYFIVE + NINETEEN + ELEVEN + TEN + FIVE + ONE?

29. 10 + 7 + 1 = 9 + 6 + 3, 11 + 10 = 8 + 7 + 6, 12 + 7 + 6 + 5 = 11 + 10 + 9, ..., 19 + 10 + 3 = 14 + 13 + 4 + 1 (31 examples in all).

30. (a) $567^2 = 321489$, $807^2 = 651249$, or $854^2 = 729316$. (b) $958^2 = 917764$. (c) $96 \times 7^2 = 4704$. (d) 51304/61904 = 7260/8760. [*Strand* **78** (1929), 91, 208; *J. Recr. Math* **3** (1970), 43; **13** (1981), 212; **27** (1995), 137. The solutions to (b), (c), and (d) are unique. With a right-to-left approach based on Algorithm X, the answers are found in (14, 13, 11, 3423) kilomems, respectively.]

31. 5/34 + 7/68 + 9/12(!). One can verify uniqueness with Algorithm X using the side condition A < D < G, in about 265 Kμ.

32. There are eleven ways, of which the most surprising is 3 + 69258/714. [See *The Weekly Dispatch* (9 and 23 June 1901); *Amusements in Mathematics* (1917), 158–159.]

33. (a) 1, 2, 3, 4, 15, 18, 118, 146. (b) 6, 9, 16, 20, 27, 126, 127, 129, 136, 145. [*The Weekly Dispatch* (11 and 30 November, 1902); *Amusements in Math.* (1917), 159.]

In this case one suitable strategy is to find all variations where $a_k \ldots a_{l-1}/a_l \ldots a_9$ is an integer, then to record solutions for all permutations of $a_1 \ldots a_{k-1}$. There are

exactly 164959 integers with a unique solution, the largest being 9876533. There are solutions for all years in the 21st century except 2091. The most solutions (125) occur when $n = 6443$; the longest stretch of representable n's is $5109 < n < 7060$. Dudeney was able to get the correct answers by hand for small n by "casting out nines."

34. (a) $x = 10^5$, 7378+155+92467 = 7178+355+92467 = 1016+733+98251 = 100000. (b) $x = 4^7$, 3036 + 455 + 12893 = 16384 is unique. The fastest way to resolve this problem is probably to start with a list of the 2529 primes that consist of five distinct digits (namely 10243, 10247, ..., 98731) and to permute the five remaining digits.

Incidentally, the unrestricted alphametic `EVEN + ODD = PRIME` has ten solutions; both `ODD` and `PRIME` are prime in just one of them. [See M. Arisawa, *J. Recr. Math.* **8** (1975), 153.]

35. In general, if $s_k = \|S_k\|$ for $1 \le k < n$, there are $s_1 \ldots s_{k-1}$ ways to choose each of the nonidentity elements of S_k. Hence the answer is $\prod_{k=1}^{n-1}(\prod_{j=1}^{k-1} s_j^{s_{k-1}})$, which in this case is $2^2 \cdot 6^3 \cdot 24^{15} = 4361966924740238361231 36$.

(But if the vertices are renumbered, the s_k values may change. For example, if vertices `(0,3,5)` of `(12)` are interchanged with `(e,d,c)`, we have $s_{14} = 1$, $s_{13} = 6$, $s_{12} = 4$, $s_{11} = 1$, and $4^5 \cdot 24^{15}$ Sims tables.)

36. Since each of {`0,3,5,6,9,a,c,f`} lies on three lines, but every other element lies on only two, it is clear that we may let $S_{\mathtt{f}} = \{(), \sigma, \sigma^2, \sigma^3, \alpha, \alpha\sigma, \alpha\sigma^2, \alpha\sigma^3\}$, where $\sigma =$ `(03fc)(17e4)(2bd4)(56a9)` is a 90° rotation and $\alpha =$ `(05)(14)(27)(36)(8d)(9c)(af)(be)` is an inside-out twist. Also $S_{\mathtt{e}} = \{(), \beta, \gamma, \beta\gamma\}$, where $\beta =$ `(14)(28)(3c)(69)(be)` is a transposition and $\gamma =$ `(12)(48)(5a)(69)(7b)(de)` is another twist; $S_{\mathtt{d}} = \cdots = S_{\mathtt{1}} = \{()\}$. (There are $4^7 - 1$ alternative answers.)

37. The set S_k can be chosen in $k!^{k-1}$ ways (see exercise 35), and its nonidentity elements can be assigned to $\sigma(k,1)$, ..., $\sigma(k,k)$ in $k!$ further ways. So the answer is $A_n = \prod_{k=1}^{n-1} k!^k = n!^{\binom{n}{2}} / \prod_{k=1}^{n} k^{\binom{k}{2}}$. For example, $A_{10} \approx 6.256 \times 10^{148}$. We have

$$\sum_{k=1}^{n-1} \binom{k}{2} \ln k = \frac{1}{2}\int_1^n x(x-1)\ln x\,dx + O(n^2 \log n) = \frac{1}{6}n^3 \ln n + O(n^3)$$

by Euler's summation formula; thus $\ln A_n = \frac{1}{3}n^3 \ln n + O(n^3)$.

38. The probability that $\phi(k)$ is needed in step G4 is $1/k! - 1/(k+1)!$, for $1 \le k < n$; the probability is $1/n!$ that we don't get to step G4 at all. Since $\phi(k)$ does $\lceil k/2 \rceil$ transpositions, the average is $\sum_{k=1}^{n-1}(1/k! - 1/(k+1)!)\lceil k/2 \rceil = \sum_{k=1}^{n-1}(\lceil k/2 \rceil - \lceil (k-1)/2 \rceil)/k! - \lceil (n-1)/2 \rceil/n! = \sum_{k\,\mathrm{odd}} 1/k! + O(1/(n-1)!)$.

39. (a) 0123, 1023, 2013, 0213, 1203, 2103, 3012, 0312, 1302, 3102, 0132, 1032, 2301, 3201, 0231, 2031, 3021, 0321, 1230, 2130, 3120, 1320, 2310, 3210; (b) 0123, 1023, 2013, 0213, 1203, 2103, 3102, 1302, 0312, 3012, 1032, 0132, 0231, 2031, 3021, 0321, 2301, 3201, 3210, 2310, 1320, 3120, 2130, 1230.

40. By induction we find $\sigma(1,1) = (0\ 1)$, $\sigma(2,2) = (0\ 1\ 2)$,

$$\sigma(k,k) = \begin{cases} (0\ k)(k{-}1\ k{-}2\ \ldots\ 1), & \text{if } k \ge 3 \text{ is odd,} \\ (0\ k{-}1\ k{-}2\ 1\ \ldots\ k{-}3\ k), & \text{if } k \ge 4 \text{ is even;} \end{cases}$$

also $\omega(k) = (0\ k)$ when k is even, $\omega(k) = (0\ k{-}2\ \ldots\ 1\ k{-}1\ k)$ when $k \ge 3$ is odd. Thus when $k \ge 3$ is odd, $\sigma(k,1) = (k\ k{-}1\ 0)$ and $\sigma(k,j)$ takes $k \mapsto j-1$ for $1 < j < k$; when $k \ge 4$ is even, $\sigma(k,j) = (0\ k\ k{-}3\ \ldots\ 1\ k{-}2\ k{-}1)^j$ for $1 \le j \le k$.

Notes: The first scheme that causes Algorithm G to generate all permutations by single transpositions was devised by Mark Wells [*Math. Comp.* **15** (1961), 192–195], but it was considerably more complicated. W. Lipski, Jr., studied such schemes in general and found a variety of additional methods [*Computing* **23** (1979), 357–365].

41. We may assume that $r < n$. Algorithm G will generate r-variations for any Sims table if we simply change '$k \leftarrow 1$' to '$k \leftarrow n - r$' in step G3, provided that we redefine $\omega(k)$ to be $\sigma(n-r, n-r)\ldots\sigma(k,k)$ instead of using (16).

If $n-r$ is odd, the method of (27) is still valid, although the formulas in answer 40 need to be revised when $k < n-r+2$. The new formulas are $\sigma(k,j) = (k\ j{-}1\ \ldots\ 1\ 0)$ and $\omega(k) = (k\ \ldots\ 1\ 0)$ when $k = n-r$; $\sigma(k,j) = (k\ \ldots\ 1\ 0)^j$ when $k = n-r+1$.

If $n-r$ is even, we can use (27) with even and odd reversed, if $r \le 3$. But when $r \ge 4$ a more complex scheme is needed, because a fixed transposition like $(k\ 0)$ can be used for odd k only if $\omega(k-1)$ is a k-cycle, which means that $\omega(k-1)$ must be an even permutation; but $\omega(k)$ is odd for $k \ge n-r+2$.

The following scheme works when $n-r$ is even: Let $\tau(k,j) = (k\ k{-}j)$ for $1 \le j \le k = n-r$, and use (27) when $k > n-r$. Then, when $k = n-r+1$, we have $\omega(k-1) = (0\ 1\ \ldots\ k{-}1)$, hence $\sigma(k,j)$ takes $k \mapsto (2j-1) \bmod k$ for $1 \le j \le k$, and $\sigma(k,k) = (k\ k{-}1\ k{-}3\ \ldots\ 0\ k{-}2\ \ldots\ 1)$, $\omega(k) = (k\ \ldots\ 1\ 0)$, $\sigma(k{+}1, j) = (k{+}1\ \ldots\ 0)^j$.

42. If $\sigma(k,j) = (k\ j{-}1)$ we have $\tau(k,1) = (k\ 0)$ and $\tau(k,j) = (k\ j{-}1)(k\ j{-}2) = (k\ j{-}1\ j{-}2)$ for $2 \le j \le k$.

43. Of course $\omega(1) = \sigma(1,1) = \tau(1,1) = (0\ 1)$. The following construction makes $\omega(k) = (k{-}2\ k{-}1\ k)$ for all $k \ge 2$: Let $\alpha(k,j) = \tau(k,j)\omega(k-1)^-$, where $\alpha(2,1) = (2\ 0)$, $\alpha(2,2) = (2\ 0\ 1)$, $\alpha(3,1) = \alpha(3,3) = (3\ 1)$, $\alpha(3,2) = (3\ 1\ 0)$; this makes $\sigma(2,2) = (0\ 2)$, $\sigma(3,3) = (0\ 3\ 1)$. Then for $k \ge 4$, let

	$k \bmod 3 = 0$		$k \bmod 3 = 1$		$k \bmod 3 = 2$
$\alpha(k,k{-}2)$	$(k\ k{-}2\ 0)$	or	$(k\ k{-}3\ 0)$	or	$(k\ k{-}1\ 0)$,
$\alpha(k,k{-}1)$	$(k\ k{-}2\ k{-}3)$	or	$(k\ k{-}3)$	or	$(k\ k{-}1\ k{-}3)$,
$\alpha(k,k)$	$(k\ k{-}2)$	or	$(k\ k{-}3\ k{-}2)$	or	$(k\ k{-}2)$;

this makes $\sigma(k,k) = (k{-}3\ k\ k{-}2)$ as required.

44. No, because $\tau(k,j)$ is a $(k+1)$-cycle, not a transposition. (See (19) and (24).)

45. (a) 202280070, since $u_k = \max\big(\{0, 1, \ldots, a_k - 1\} \setminus \{a_1, \ldots, a_{k-1}\}\big)$. (Actually u_n is never set by the algorithm, but we can assume that it is zero.) (b) 425368917.

46. True (assuming that $u_n = 0$). If either $u_k > u_{k+1}$ or $a_k > a_{k+1}$ we must have $a_k > u_k \ge a_{k+1} > u_{k+1}$.

47. Steps $(\text{X1}, \text{X2}, \ldots, \text{X6})$ are performed respectively $(1, A, B, A-1, B-N_n, A)$ times, where $A = N_0 + \cdots + N_{n-1}$ and $B = nN_0 + (n-1)N_1 + \cdots + 1N_{n-1}$.

48. Steps $(\text{X2}, \text{X3}, \text{X4}, \text{X5}, \text{X6})$ are performed respectively $A_n + (1, n!, 0, 0, 1)$ times, where $A_n = \sum_{k=1}^{n-1} n^{\underline{k}} = n! \sum_{k=1}^{n-1} 1/k! \approx n!\,(e-1)$. Assuming that they cost respectively $(1, 1, 3, 1, 3)$ mems, for operations involving a_j, l_j, or u_j, the total cost is about $9e - 8 \approx 16.46$ mems per permutation.

Algorithm L uses approximately $(e, 2 + e/2, 2e + 2e^{-1} - 4)$ mems per permutation in steps $(\text{L2}, \text{L3}, \text{L4})$, for a total of $3.5e + 2e^{-1} - 2 \approx 8.25$ (see exercise 5).

Algorithm X could be tuned up for this case by streamlining the code when k is near n. But so can Algorithm L, as shown in exercise 1.

49. Order the signatures so that $|s_0| \geq \cdots \geq |s_9|$; also prepare tables $w_0 \ldots w_9$, $x_0 \ldots x_9$, $y_0 \ldots y_9$, so that the signatures $\{s_k, \ldots, s_9\}$ are $w_{x_k} \leq \cdots \leq w_{y_k}$. For example, when SEND + MORE = MONEY we have $(s_0, \ldots, s_9) = (-9000, 1000, -900, 91, -90, 10, 1, -1, 0, 0)$ for the respective letters (M, S, O, E, N, R, D, Y, A, B); also $(w_0, \ldots, w_9) = (-9000, -900, -90, -1, 0, 0, 1, 10, 91, 1000)$ and $x_0 \ldots x_9 = 01122333344$, $y_0 \ldots y_9 = 9988776554$. Yet another table $f_0 \ldots f_9$ has $f_j = 1$ if the digit corresponding to w_j cannot be zero; in this case $f_0 \ldots f_9 = 1000000001$. These tables make it easy to compute the largest and smallest values of

$$s_k a_k + \cdots + s_9 a_9$$

over all choices $a_k \ldots a_9$ of the remaining digits, using the method of exercise 25, since the links l_j tell us those digits in increasing order.

This method requires a rather expensive computation at each node of the search tree, but it often succeeds in keeping that tree small. For example, it solves the first eight alphametics of exercise 24 with costs of only 7, 13, 7, 9, 5, 343, 44, and 89 kilomems; this is a substantial improvement in cases (a), (b), (e), and (h), although case (f) comes out significantly worse. Another bad case is the 'CHILD' example of answer 27, where left-to-right needs 2947 kilomems compared to 588 for the right-to-left approach. Left-to-right does, however, fare better on BLOOD + SWEAT + TEARS (73 versus 360) and HELLO + WORLD (340 versus 410).

50. If α is in a permutation group, so are all its powers α^2, α^3, ..., including $\alpha^{m-1} = \alpha^-$, where m is the order of α (the least common multiple of its cycle lengths). And (32) is equivalent to $\alpha^- = \sigma_1 \sigma_2 \ldots \sigma_{n-1}$.

51. False. For example, $\sigma(k,i)^-$ and $\sigma(k,j)^-$ might both take $k \mapsto 0$.

52. $\tau(k,j) = (k{-}j\ k{-}j{+}1)$ is an adjacent interchange, and

$$\omega(k) = (n{-}1\ \ldots\ 0)(n{-}2\ \ldots\ 0)\ldots(k\ \ldots\ 0) = \phi(n-1)\phi(k-1)$$

is a k-flip followed by an n-flip. The permutation corresponding to control table $c_0 \ldots c_{n-1}$ in Algorithm H has c_j elements to the right of j that are less than j, for $0 \leq j < n$; so it is the same as the permutation corresponding to $c_1 \ldots c_n$ in Algorithm P, except that subscripts are shifted by 1.

The only essential difference between Algorithm P and this version of Algorithm H is that Algorithm P uses a reflected Gray code to run through all possibilities of its control table, while Algorithm H runs through those mixed-radix numbers in ascending (lexicographic) order.

Indeed, Gray code can be used with any Sims table, by modifying either Algorithm G or Algorithm H. Then all transitions are by $\tau(k,j)$ or by $\tau(k,j)^-$, and the permutations $\omega(k)$ are irrelevant.

53. The text's proof that $n! - 1$ transpositions cannot be achieved for $n = 4$ also shows that we can reduce the problem from n to $n-2$ at the cost of a single transposition $(n{-}1\ n{-}2)$, which was called '$(3\,c)$' in the notation of that proof.

Thus we can generate all permutations by making the following transformation in step H4: If $k = n-1$ or $k = n-2$, transpose $a_{j \bmod n} \leftrightarrow a_{(j-1) \bmod n}$, where $j = c_{n-1} - 1$. If $k = n-3$ or $k = n-4$, transpose $a_{n-1} \leftrightarrow a_{n-2}$ and also $a_{j \bmod (n-2)} \leftrightarrow a_{(j-1) \bmod (n-2)}$, where $j = c_{n-3} - 1$. And in general if $k = n-2t-1$ or $k = n-2t-2$, transpose $a_{n-2i+1} \leftrightarrow a_{n-2i}$ for $1 \leq i \leq t$ and also $a_{j \bmod (n-2t)} \leftrightarrow a_{(j-1) \bmod (n-2t)}$, where $j = c_{n-2t-1} - 1$. [See *CACM* **19** (1976), 68–72.]

The corresponding Sims table permutations can be written down as follows, although they don't appear explicitly in the algorithm itself:

$$\sigma(k,j)^- = \begin{cases} (0\ 1\ \ldots\ j{-}1\ k), & \text{if } n-k \text{ is odd;} \\ (0\ 1\ \ldots\ k)^j, & \text{if } n-k \text{ is even.} \end{cases}$$

The value of $a_{j \bmod (n-2t)}$ will be $n-2t-1$ after the interchange. For efficiency we can also use the fact that k usually equals $n-1$. The total number of transpositions is $\sum_{t=0}^{\lfloor n/2 \rfloor}(n-2t)! - \lfloor n/2 \rfloor - 1$.

54. Yes; the transformation can be any k-cycle on positions $\{1,\ldots,k\}$.

55. (a) Since $\rho_!(m) = \rho_!(m \bmod n!)$ when $n > \rho_!(m)$, we have $\rho_!(n!+m) = \rho_!(m)$ for $0 < m < n\cdot n! = (n+1)! - n!$. Therefore $\beta_{n!+m} = \sigma_{\rho_!(n!+m)}\ldots\sigma_{\rho_!(n!+1)}\beta_{n!} = \sigma_{\rho_!(m)}\ldots\sigma_{\rho_!(1)}\beta_{n!} = \beta_m\beta_{n!}$ for $0 \le m < n\cdot n!$, and we have in particular

$$\beta_{(n+1)!} = \sigma_{n+1}\beta_{(n+1)!-1} = \sigma_{n+1}\beta_{n!-1}\beta_{n!}^n = \sigma_{n+1}\sigma_n^-\beta_{n!}^{n+1}.$$

Similarly $\alpha_{n!+m} = \beta_{n!}^-\alpha_m\beta_{n!}\alpha_{n!}$ for $0 \le m < n\cdot n!$.

Since $\beta_{n!}$ commutes with τ_n and τ_{n+1} we find $\alpha_{n!} = \tau_n\alpha_{n!-1}$, and

$$\begin{aligned} \alpha_{(n+1)!} = \tau_{n+1}\alpha_{(n+1)!-1} &= \tau_{n+1}\beta_{n!}^-\alpha_{(n+1)!-1-n}\beta_{n!}\alpha_{n!} \\ &= \cdots \\ &= \tau_{n+1}\beta_{n!}^{-n}\alpha_{n!-1}(\beta_{n!}\alpha_{n!})^n \\ &= \beta_{n!}^{-n-1}\tau_{n+1}\tau_n^-(\beta_{n!}\alpha_{n!})^{n+1} \\ &= \beta_{(n+1)!}^-\sigma_{n+1}\sigma_n^-\tau_{n+1}\tau_n^-(\beta_{n!}\alpha_{n!})^{n+1}. \end{aligned}$$

(b) In this case $\sigma_{n+1}\sigma_n^- = (n\ n{-}1\ \ldots\ 1)$ and $\tau_{n+1}\tau_n^- = (n{+}1\ n\ 0)$, and we have $\beta_{(n+1)!}\alpha_{(n+1)!} = (n{+}1\ n\ \ldots\ 0)$ by induction. Therefore $\alpha_{jn!+m} = \beta_{n!}^{-j}\alpha_m(n\ \ldots\ 0)^j$ for $0 \le j \le n$ and $0 \le m < n!$. All permutations of $\{0,\ldots,n\}$ are achieved because $\beta_{n!}^{-j}\alpha_m$ fixes n and $(n\ \ldots\ 0)^j$ takes $n \mapsto n-j$.

56. If we set $\sigma_k = (k{-}1\ k{-}2)(k{-}3\ k{-}4)\ldots$ in the previous exercise, we find by induction that $\beta_{n!}\alpha_{n!}$ is the $(n+1)$-cycle $(0\ n\ n{-}1\ n{-}3\ \ldots$ (2 or 1) (1 or 2) $\ldots\ n{-}4\ n{-}2)$.

57. Arguing as in answer 5, we obtain $\sum_{k=2}^{n-1}[k\ \text{odd}]/k! - (\lfloor n/2 \rfloor - 1)/n! = \sinh 1 - 1 - O(1/(n-1)!)$.

58. True. By the formulas of exercise 55 we have $\alpha_{n!-1} = (0\ n)\beta_{n!}^-(n\ \ldots\ 0)$, and this takes $0 \mapsto n-1$ because $\beta_{n!}$ fixes n. (Consequently Algorithm E will define a Hamiltonian *circuit* on the graph of exercise 66 if and only if $\beta_{n!} = (n{-}1\ \ldots\ 2\ 1)$, and this holds if and only if the length of every cycle of $\beta_{(n-1)!}$ is a divisor of n. The latter is true for $n = 2$, 3, 4, 6, 12, 20, and 40, but for no other $n \le 250{,}000$.)

59. The Cayley graph with generators $(\alpha_1,\ldots,\alpha_k)$ in the text's definition is isomorphic to the Cayley graph with generators $(\alpha_1^-,\ldots,\alpha_k^-)$ in the alternative definition, since $\pi \to \alpha_j\pi$ in the former if and only if $\pi^- \to \pi^-\alpha_j^-$ in the latter.

60. There are 88 delta sequences, which reduce to four classes: $P = (32131231)^3$ (plain changes, represented by 8 different delta sequences); $Q = (32121232)^3$ (a doubly Gray variant of plain changes, with 8 representatives); $R = (121232321232)^2$ (a doubly Gray code with 24 representatives); $S = 2\alpha3\alpha^R$, $\alpha = 12321312121$ (48 representatives). Classes P and Q are cyclic shifts of their complements; classes P, Q, and S are shifts of their reversals; class R is a shifted reversal of its complement. [See A. L. Leigh Silver, *Math. Gazette* **48** (1964), 1–16.]

61. There are respectively $(26, 36, 20, 26, 28, 40, 40, 20, 26, 28, 28, 26)$ such paths ending at $(1243, 1324, 1432, 2134, 2341, 2413, 3142, 3214, 3421, 4123, 4231, 4312)$.

62. There are only two paths when $n = 3$, ending respectively at 132 and 213. But when $n \geq 4$ there are Gray paths leading from $12\ldots n$ to any odd permutation $a_1a_2\ldots a_n$. Exercise 61 establishes this when $n = 4$, and we can prove it by induction for $n > 4$ as follows.

Let $A(j)$ be the set of all permutations that begin with j, and let $A(j,k)$ be those that begin with jk. If $(\alpha_0, \alpha_1, \ldots, \alpha_n)$ are any odd permutations such that $\alpha_j \in A(x_j, x_{j+1})$, then $(1\,2)\,\alpha_j$ is an even permutation in $A(x_{j+1}, x_j)$. Consequently, if $x_1x_2\ldots x_n$ is a permutation of $\{1, 2, \ldots, n\}$, there is at least one Hamiltonian path of the form

$$(1\,2)\,\alpha_0 \mathrel{—} \cdots \mathrel{—} \alpha_1 \mathrel{—} (1\,2)\,\alpha_1 \mathrel{—} \cdots \mathrel{—} \alpha_2 \mathrel{—} \cdots \mathrel{—} (1\,2)\,\alpha_{n-1} \mathrel{—} \cdots \mathrel{—} \alpha_n;$$

the subpath from $(1\,2)\,\alpha_{j-1}$ to α_j includes all elements of $A(x_j)$.

This construction solves the problem in at least $(n-2)!^n/2^{n-1}$ distinct ways when $a_1 \neq 1$, because we can take $\alpha_0 = 2\,1\ldots n$ and $\alpha_n = a_1a_2\ldots a_n$; there are $(n-2)!$ ways to choose $x_2\ldots x_{n-1}$, and $(n-2)!/2$ ways to choose each of $\alpha_1, \ldots, \alpha_{n-1}$.

Finally, if $a_1 = 1$, take any path $12\ldots n \mathrel{—} \cdots \mathrel{—} a_1a_2\ldots a_n$ that runs through all of $A(1)$, and choose any step $\alpha \mathrel{—} \alpha'$ with $\alpha \in A(1,j)$ and $\alpha' \in A(1,j')$ for some $j \neq j'$. Replace that step by

$$\alpha \mathrel{—} (1\,2)\,\alpha_1 \mathrel{—} \cdots \mathrel{—} \alpha_2 \mathrel{—} \cdots \mathrel{—} (1\,2)\,\alpha_{n-1} \mathrel{—} \cdots \mathrel{—} \alpha_n \mathrel{—} \alpha',$$

using a construction like the Hamiltonian path above but now with $\alpha_1 = \alpha$, $\alpha_n = (1\,2)\,\alpha'$, $x_1 = 1$, $x_2 = j$, $x_n = j'$, and $x_{n+1} = 1$. (In this case the permutations α_1, $\ldots$, α_n might all be even.)

63. Monte Carlo estimates using the techniques of Section 7.2.3 suggest that the total number of equivalence classes will be roughly 1.2×10^{21}; most of those classes will contain 480 Gray codes.

64. Exactly 2,005,200 delta sequences have the doubly Gray property; they belong to 4206 equivalence classes under cyclic shift, reversal, and/or complementation. Nine classes, such as the code $2\alpha2\alpha^R$ where

$$\alpha = 123432343212321212323212321212343432121234321234321212323 21,$$

are shifts of their reversal; 48 classes are composed of repeated 60-cycles. One of the most interesting of the latter type is $\alpha\alpha$ where

$$\alpha = \beta2\beta4\beta4\beta4\beta4, \qquad \beta = 32121232123.$$

65. Such a path exists for any given $N \leq n!$: Let the Nth permutation be $\alpha = a_1\ldots a_n$, and let $j = a_1$. Also let Π_k be the set of all permutations $\beta = b_1\ldots b_n$ for which $b_1 = k$ and $\beta \leq \alpha$. By induction on N there is a Gray path P_1 for Π_j. We can then construct Gray paths P_k for $\Pi_j \cup \Pi_1 \cup \cdots \cup \Pi_{k-1}$ for $2 \leq k \leq j$, successively combining P_{k-1} with a Gray code for Π_{k-1}. (See the "absorption" construction of answer 62. In fact, P_j will be a Gray *code* when N is a multiple of 6.)

66. Defining the delta sequence by the rule $\pi_{(k+1) \bmod n!} = (1\,\delta_k)\,\pi_k$, we find exactly 36 such sequences, all of which are cyclic shifts of a pattern like $(xyzyzyxzyzyz)^2$. (The next case, $n = 5$, probably has about 10^{18} solutions that are inequivalent with respect to cyclic shifting, reversal, and permutation of coordinates, thus about 6×10^{21} different

delta sequences.) Incidentally, Igor Pak has shown that the Cayley graph generated by star transpositions is an $(n-2)$-dimensional torus in general.

67. If we let π be equivalent to $\pi(12345)$, we get a reduced graph on 24 vertices that has 40768 Hamiltonian circuits, 240 of which lead to delta sequences of the form α^5 in which α uses each transposition 6 times (for example, $\alpha = 354232534234532454352452$). The total number of solutions to this problem is probably about 10^{16}.

68. If A isn't connected, neither is G. If A is connected, we can assume that it is a free tree. Moreover, in this case we can prove a generalization of the result in exercise 62: For $n \geq 4$ there is a Hamiltonian path in G from the identity permutation to any odd permutation. For we can assume without loss of generality that A contains the edge $1 \mathrel{—} 2$ where 1 is a leaf of the tree, and a proof like that of exercise 62 applies.

[This elegant construction is due to M. Tchuente, *Ars Combinatoria* **14** (1982), 115–122. Extensive generalizations have been discussed by Ruskey and Savage in *SIAM J. Discrete Math.* **6** (1993), 152–166. See also the original Russian publication in *Kibernetika* **11**, 3 (1975), 17–25; English translation, *Cybernetics* **11** (1975), 362–366.]

69. Following the hint, the modified algorithm behaves like this when $n = 5$:

$\underset{\downarrow}{12\underset{}{34}}$	$\underset{\uparrow}{12\underset{}{43}}$	$\underset{\downarrow}{1\underset{}{423}}$	$\underset{\uparrow}{41\underset{}{23}}$	$\underset{\downarrow}{4\underset{}{132}}$	$\underset{\uparrow}{14\underset{}{32}}$	$\underset{\downarrow}{13\underset{}{42}}$	$\underset{\uparrow}{1\underset{}{324}}$	$\underset{\downarrow}{31\underset{}{24}}$	$\underset{\uparrow}{3\underset{}{142}}$	$\underset{\downarrow}{34\underset{}{12}}$	$\underset{\uparrow}{4312}$
54321	24351	24153	54123	14523	14325	24315	24513	54213	14253	14352	54312
12345	15342	35142	32145	32541	52341	51342	31542	31245	35241	25341	21345
15342	12435	32415	35412$\leftarrow$	31452	51432	52431	32451$\leftarrow$	35421	31425	21435	25431
23451	53421	51423	21453$\rightarrow$	25413	23415	13425	15423$\rightarrow$	12453	52413	53412	13452
21543	51243	53241	23541	23145	25143	15243	13245	13542	53142	52143	12543
34512	34215	14235	14532	54132	34152	34251	54231	24531	24135	34125	34521
32154$\rightarrow$	35124	15324$\rightarrow$	12354	52314	32514$\leftarrow$	31524	51324	21354$\rightarrow$	25314	35214$\rightarrow$	31254
45123$\leftarrow$	42153	42351$\leftarrow$	45321	41325	41523$\rightarrow$	42513	42315	45312$\leftarrow$	41352	41253$\leftarrow$	45213
43215	43512$\leftarrow$	41532	41235	45231$\rightarrow$	43251	43152$\rightarrow$	45132	42135	42531$\leftarrow$	43521	43125
51234	21534$\rightarrow$	23514	53214	13254$\leftarrow$	15234	25134$\leftarrow$	23154	53124	13524$\rightarrow$	12534	52134
$\downarrow$	$\uparrow$	$\downarrow$	$\uparrow$	$\downarrow$	$\uparrow$	$\downarrow$	$\uparrow$	$\downarrow$	$\uparrow$	$\downarrow$	$\uparrow$

Here the columns represent sets of permutations that are cyclically rotated and/or reflected in all $2n$ ways; therefore each column contains exactly one "rosary permutation" (exercise 18). We can use Algorithm P to run through the rosary permutations systematically, knowing that the pair xy will occur before yx in its column, at which time τ' instead of ρ' will move us to the right or to the left. Step Z2 omits the interchange $a_1 \leftrightarrow a_2$, thereby causing the permutations $a_1 \ldots a_{n-1}$ to repeat themselves going backwards. (We implicitly use the fact that $t[k] = t[n! - k]$ in the output of Algorithm T.)

Now if we replace $1 \ldots n$ by $24 \ldots 31$ and change $A_1 \ldots A_n$ to $A_1 A_n A_2 A_{n-1} \ldots$, we get the unmodified algorithm whose results are shown in Fig. 22(b).

This method was inspired by a (nonconstructive) theorem of E. S. Rapoport, *Scripta Math.* **24** (1959), 51–58. It illustrates a more general fact observed by Carla Savage in 1989, namely that the Cayley graph for *any* group generated by three involutions ρ, σ, τ has a Hamiltonian circuit when $\rho\tau = \tau\rho$ [see I. Pak and R. Radoičić, "Hamiltonian expanders," to appear.]

70. No; the longest cycle in that digraph has length 358. But there do exist pairs of disjoint 180-cycles from which a Hamiltonian path of length 720 can be derived. For

example, consider the cycles $\alpha\sigma\beta\sigma$ and $\gamma\sigma\sigma$ where

$$\alpha = \tau\sigma^5\tau\sigma^5\tau\sigma^3\tau\sigma^2\tau\sigma^5\tau\sigma^3\tau\sigma^2\tau\sigma^5\tau\sigma^5\tau\sigma^2\tau\sigma^3\tau\sigma^1\tau\sigma^5\tau\sigma^5\tau\sigma^5\tau\sigma^3\tau\sigma^1\tau\sigma^1\tau\sigma^3\tau\sigma^2\tau\sigma^1\tau\sigma^1;$$
$$\beta = \sigma^3\tau\sigma^5\tau\sigma^2\tau\sigma^2\tau\sigma^5\tau\sigma^2\tau\sigma^3\tau\sigma^1\tau\sigma^1\tau\sigma^5\tau\sigma^1\tau\sigma^3\tau\sigma^5\tau\sigma^5\tau\sigma^3\tau\sigma^2\tau\sigma^1\tau\sigma^2\tau\sigma^3\tau\sigma^1\tau\sigma^1\tau\sigma^3\tau\sigma^2\tau\sigma^4;$$
$$\gamma = \sigma\tau\sigma^5\tau\sigma^5\tau\sigma^3\tau\sigma^1\tau\sigma^1\tau\sigma^3\tau\sigma^2\tau\sigma^5\tau\sigma^2\tau\sigma^3\tau\sigma^5\tau\sigma^1\tau\sigma^5\tau\sigma^3\tau\sigma^2\tau\sigma^1\tau\sigma^2\tau\sigma^3\tau\sigma^1\tau\sigma^1\tau\sigma^3\tau\sigma^2$$
$$\tau\sigma^5\tau\sigma^5\tau\sigma^5\tau\sigma^3\tau\sigma^5\tau\sigma^2\tau\sigma^5\tau\sigma^2\tau\sigma^3\tau\sigma^1\tau\sigma^1\tau\sigma^5\tau\sigma^1\tau\sigma^3\tau\sigma^3\tau\sigma^5\tau\sigma^5\tau\sigma^1\tau\sigma^5\tau\sigma^2\tau\sigma^3\tau\sigma^1\tau\sigma^2.$$

If we start with 134526 and follow $\alpha\sigma\beta\tau$ we reach 163452; then follow $\gamma\sigma\tau$ and reach 126345; then follow $\sigma\gamma\tau$ and reach 152634; then follow $\beta\sigma\alpha$, ending at 415263.

71. Any Hamiltonian path includes $(n-1)!$ vertices that take $y \mapsto x$, each of which (if not the last) is followed by a vertex that takes $x \mapsto x$. So one must be last; otherwise $(n-1)!+1$ vertices would take $x \mapsto x$.

72. (a) Assume first that β is the identity permutation $()$. Then every cycle of α that contains an element of A lies entirely within A. Hence the cycles of σ are obtained by omitting all cycles of α that contain no element of A. All remaining cycles have odd length, so σ is an even permutation.

If β is not the identity, we apply this argument to $\alpha' = \alpha\beta^-$, $\beta' = ()$, and $\sigma' = \sigma\beta^-$, concluding that σ' is an even permutation; thus σ and β have the same sign.

Similarly, σ and α have the same sign, because $\beta\alpha^- = (\alpha\beta^-)^-$ has the same order as $\alpha\beta^-$.

(b) Let X be the vertices of the Cayley graph in Theorem R, and let α be the permutation of X that takes a vertex π into $\alpha\pi$; this permutation has g/a cycles of length a. Define the permutation β similarly. Then $\alpha\beta^-$ has g/c cycles of length c. If c is odd, any Hamiltonian circuit in the graph defines a cycle σ that contains all the vertices and satisfies the hypotheses of (a). Therefore α and β have an odd number of cycles, because the sign of a permutation on n elements with r cycles is $(-1)^{n-r}$ (see exercise 5.2.2–2).

[This proof, which shows that X cannot be the union of any odd number of cycles, was presented by Rankin in *Proc. Cambridge Phil. Soc.* **62** (1966), 15–16.]

73. The representation $\beta^j\gamma^k$ is unique if we require $0 \le j < g/c$ and $0 \le k < c$. For if we had $\beta^j = \gamma^k$ for some j with $0 < j < g/c$, the group would have at most jc elements. It follows that $\beta^{g/c} = \gamma^t$ for some t.

Let σ be a Hamiltonian circuit, as in the previous answer. If $x\sigma = x\alpha$ then $x\gamma\sigma$ must be $x\gamma\alpha$, because $x\gamma\beta = \alpha$. And if $x\sigma = x\beta$ then $x\gamma\sigma$ cannot be $x\gamma\alpha$, because that would imply $x\gamma^c\sigma = x\gamma^c\alpha$. Thus the elements $x\gamma^k$ all have equivalent behavior with respect to their successors in σ.

Notice that if $j \ge 0$ there is a $k \le j$ such that $x\sigma^j = x\alpha^k\beta^{j-k} = x\beta^j\gamma^k$. Therefore $x\sigma^{g/c} = x\gamma^{t+k}$ is equivalent to x, and the same behavior will repeat. We return to x for the first time in g steps if and only if $t+k$ is relatively prime to c.

74. Apply the previous exercise with $g = mn$, $a = m$, $b = n$, $c = mn/d$. The number t satisfies $t \equiv 0$ (modulo m), $t + d \equiv 0$ (modulo n); and it follows that $k + t \perp c$ if and only if $(d-k)m/d \perp kn/d$.

Notes: The modular Gray path of exercise 7.2.1.1–78 is a Hamiltonian path from $(0,0)$ to $\big(m-1, (-m) \bmod n\big)$, so it is a Hamiltonian circuit if and only if m is a multiple of n. It is natural to conjecture (falsely) that at least one Hamiltonian circuit exists whenever $d > 1$. But P. Erdős and W. T. Trotter have observed [*J. Graph Theory* **2** (1978), 137–142] that if p and $2p+1$ are odd prime numbers, no suitable k exists when $m = p(2p+1)(3p+1)$ and $n = (3p+1)\prod_{q=1}^{3p} q^{[q \text{ is prime}][q \ne p][q \ne 2p+1]}$.

See J. A. Gallian, *Mathematical Intelligencer* **13**, 3 (Summer 1991), 40–43, for interesting facts about other kinds of cycles in $C_m \times C_n$.

75. We may assume that the tour begins in the lower left corner. There are no solutions when m and n are both divisible by 3, because 2/3 of the cells are unreachable in that case. Otherwise, letting $d = \gcd(m, n)$ and arguing as in the previous exercise but with $(x, y)\alpha = ((x+2) \bmod m, (y+1) \bmod n)$ and $(x, y)\beta = ((x+1) \bmod m, (y+2) \bmod n)$, we find the answer

$$\sum_{k=1}^{d-1} \binom{d}{k} \big[\gcd\big((2d-k)\,m, (k+d)\,n\big) = d \text{ or } \big(mn \perp 3 \text{ and } \gcd\big((2d-k)\,m, (k+d)\,n\big) = 3d\big)\big].$$

76.

```
01  * Permutation generator \`a la Heap
02  N     IS    10              The value of n (3 or more, not large)
03  t     IS    $255
04  j     IS    $0              8j
05  k     IS    $1              8k
06  ak    IS    $2
07  aj    IS    $3
08        LOC   Data_Segment
09  a     GREG  @               Base address for a_0 ... a_{n-1}
10  A0    IS    @
11  A1    IS    @+8
12  A2    IS    @+16
13        LOC   @+8*N           Space for a_0 ... a_{n-1}
14  c     GREG  @-8*3           Location of 8c_0
15        LOC   @-8*3+8*N       8c_3 ... 8c_{n-1}, initially zero
16        OCTA  -1              8c_n = -1, a convenient sentinel
17  u     GREG  0               Contents of a_0, except in inner loop
18  v     GREG  0               Contents of a_1, except in inner loop
19  w     GREG  0               Contents of a_2, except in inner loop
20        LOC   #100
21  1H    STCO  0,c,k           B-A   c_k <- 0.
22        INCL  k,8             B-A   k <- k+1.
23  0H    LDO   j,c,k           B     j <- c_k.
24        CMP   t,j,k           B
25        BZ    t,1B            B     Loop if c_k = k.
26        BN    j,Done          A     Terminate if c_k < 0 (k = n).
27        LDO   ak,a,k          A-1   Fetch a_k.
28        ADD   t,j,8           A-1
29        STO   t,c,k           A-1   c_k <- j+1.
30        AND   t,k,#8          A-1
31        CSZ   j,t,0           A-1   Set j <- 0 if k is even.
32        LDO   aj,a,j          A-1   Fetch a_j.
33        STO   ak,a,j          A-1   Replace it by a_k.
34        CSZ   u,j,ak          A-1   Set u <- a_k if j = 0.
35        SUB   j,j,8           A-1   j <- j-1.
36        CSZ   v,j,ak          A-1   Set v <- a_k if j = 0.
37        SUB   j,j,8           A-1   j <- j-1.
38        CSZ   w,j,ak          A-1   Set w <- a_k if j = 0.
39        STO   aj,a,k          A-1   Replace a_k by what was a_j.
```

```
40  Inner PUSHJ  0,Visit      A
          ...                          (See (42))
55        PUSHJ  0,Visit      A
56        SET    t,u          A        Swap u ↔ w.
57        SET    u,w          A
58        SET    w,t          A
59        SET    k,8*3        A        k ← 3.
60        JMP    0B           A
61  Main  LDO    u,A0         1
62        LDO    v,A1         1
63        LDO    w,A2         1
64        JMP    Inner        1        ∎
```

77. Lines 31–38 become $2r-1$ instructions, lines 61–63 become r, and lines 56–58 become $3+(r-2)[r \text{ even}]$ instructions (see $\omega(r-1)$ in answer 40). The total running time is therefore $\big((2r!+2)A+2B+r-5\big)\mu+\big((2r!+2r+7+(r-2)[r \text{ even}])A+7B-r-4\big)\upsilon$, where $A=n!/r!$ and $B=n!(1/r!+\cdots+1/n!)$.

78. `SLU u,[#f],t; SLU t,a,4; XOR t,t,a; AND t,t,u; SRU u,t,4; OR t,t,u; XOR a,a,t`; here, as in the answer to exercise 1.3.1′–34, the notation '`[#f]`' denotes a register that contains the constant value `#f`.

79. `SLU u,a,t; MXOR u,[#8844221188442211],u; AND u,u,[#ff000000]; SRU u,u,t; XOR a,a,u`. This cheats, since it transforms `#12345678` to `#13245678` when $t=4$, but (45) still works.

Even faster and trickier would be a routine analogous to (42): Consider

```
PUSHJ 0,Visit; MXOR a,a,c1; PUSHJ 0,Visit; ... MXOR a,a,c5; PUSHJ 0,Visit
```

where `c1`, ..., `c5` are constants that would cause `#12345678` to become successively `#12783456`, `#12567834`, `#12563478`, `#12785634`, `#12347856`. Other instructions, executed only 1/6 or 1/24 as often, can take care of shuffling nybbles within and between bytes. Very clever, but it doesn't beat (46) in view of the `PUSHJ/POP` overhead.

80.
```
t  IS   $255 ;k IS $0 ;kk IS $1 ;c IS $2 ;d IS $3
   SET  k,1      k ← 1.
3H SRU  d,a,60   d ← leftmost nybble.
   SLU  a,a,4    a ← 16a mod 16^16.
   CMP  c,d,k
   SLU  kk,k,2
   SLU  d,d,kk
   OR   t,t,d    t ← t + 16^k d.
   PBNZ c,1B     Return to main loop if d ≠ k.
   INCL k,1      k ← k + 1.
   PBNZ a,3B     Return to second loop if k < n.   ∎
```

81. $\mu+\big(5n!+11A-(n-1)!+6\big)\upsilon=\big((5+10/n)\upsilon+O(n^{-2})\big)n!$, plus the visiting time, where $A=\sum_{k=1}^{n-1}k!$ is the number of times the loop at `3H` is used.

82. With suitable initialization and a 13-octabyte table, only about a dozen `MMIX` instructions are needed:

```
magic  GREG #8844221188442211
0H     ⟨Visit register a⟩
       PBN  c,Sigma
Tau    MXOR t,magic,a; ANDNL t,#ffff; JMP 1F
Sigma  SRU  t,a,20; SLU a,a,4; ANDNML a,#f00
1H     XOR  a,a,t; SLU c,c,1
2H     PBNZ c,0B; INCL p,8
3H     LDOU c,p,0; PBNZ c,0B                    ∎
```

83. Assuming that the processors all have essentially the same speed, we can let the kth processor generate all permutations of rank r for $(k-1)n!/p \le r < kn!/p$, using any method based on control tables $c_1 \ldots c_n$. The starting and ending control tables are easily computed by converting their ranks to mixed-radix notation.

84. We can use a technique like that of Algorithm 3.4.2P: To compute $k = r(\alpha)$, first set $a'_{a_j} \leftarrow j$ for $1 \le j \le n$ (the inverse permutation). Then set $k \leftarrow 0$, and for $j = n$, $n-1$, ..., 2 (in this order) set $t \leftarrow a'_j$, $k \leftarrow kj + t - 1$, $a_t \leftarrow a_j$, $a'_{a_j} \leftarrow t$. To compute $r^{[-1]}(k)$, start with $a_1 \leftarrow 1$. Then for $j = 2$, ..., $n-1$, n (in this order) set $t \leftarrow (k \bmod j) + 1$, $a_j \leftarrow a_t$, $a_t \leftarrow j$, $k \leftarrow \lfloor k/j \rfloor$. [See S. Pleszczyński, *Inf. Proc. Letters* **3** (1975), 180–183; W. Myrvold and F. Ruskey, *Inf. Proc. Letters* **79** (2001), 281–284.]

85. If $x \prec y$ and $y \prec z$, the algorithm will never move y to the left of x, nor z to the left of y, so it will never test x versus z.

86. They appear in lexicographic order; Algorithm P used a reflected Gray order.

87. Generate inverse permutations with $a'_0 < a'_1 < a'_2$, $a'_3 < a'_4 < a'_5$, $a'_6 < a'_7$, $a'_8 < a'_9$, $a'_0 < a'_3$, $a'_6 < a'_8$.

88. (a) Let $d_k = \max\{j \mid 0 \le j \le k \text{ and } j \text{ is nontrivial}\}$, where 0 is considered nontrivial. This table is easily precomputed, because j is trivial if and only if it must follow $\{1, \ldots, j-1\}$. Set $k \leftarrow d_n$ in step V2 and $k \leftarrow d_{k-1}$ in step V5.

(b) Now $M = \sum_{j=1}^{n} t_j\,[j \text{ is nontrivial}]$.

(c) There are at least two topological sorts $a_j \ldots a_k$ of the set $\{j, \ldots, k\}$, and either of them can be placed after any topological sort $a_1 \ldots a_{j-1}$ of $\{1, \ldots, j-1\}$.

(d) Algorithm 2.2.3T repeatedly outputs minimal elements (elements with no predecessors), removing them from the relation graph. We use it in reverse, repeatedly removing and giving the highest labels to *maximal* elements (elements with no successors). If only one maximal element exists, it is trivial. If j and k are both maximal, they both are output before any element x with $x \prec j$ or $x \prec k$, because steps T5 and T7 keep maximal elements in a queue (not a stack). Thus if k is nontrivial and output first, the next nontrivial element will not be output before j; and k is unrelated to j.

(e) Let the nontrivial t's be $s_1 < s_2 < \cdots < s_r = N$. Then we have $s_j \ge 2s_{j-2}$, by (c). Consequently $M = s_1 + \cdots + s_r \le s_r(1 + \frac{1}{2} + \frac{1}{4} + \cdots) + s_{r-1}(1 + \frac{1}{2} + \frac{1}{4} + \cdots) < 4s_r$. (A sharper estimate is probably possible, but exercise 89 shows that M can be larger than $2.6N$.)

89. The number N of such permutations is F_{n+1} by exercise 5.2.1–25. Therefore $M = F_{n+1} + \cdots + F_1 = F_{n+3} - 1 \approx \phi^2 N$. Notice incidentally that all such permutations satisfy $a_1 \ldots a_n = a'_1 \ldots a'_n$. They can be arranged in a Gray path (exercise 7.2.1.1–89).

90. Since $t_j = (j-1)(j-3)\ldots(2 \text{ or } 1)$, we find $M = \big(1 + 2/\sqrt{\pi n} + O(1/n)\big)N$.

Note: The inversion tables $c_1 \ldots c_n$ for permutations satisfying (49) are characterized by the conditions $c_1 = 0$, $0 \le c_{2k} \le c_{2k-1}$, $0 \le c_{2k+1} \le c_{2k-1} + 1$.

91. The total number of pairs (R, S), where R is a partial ordering and S is a linear ordering that includes R, is equal to P_n times the expected number of topological sorts; it is also Q_n times $n!$. So the answer is $n!\,Q_n/P_n$.

We will discuss the computation of P_n and Q_n in Section 7.2.3. For $1 \le n \le 12$ the expectation turns out to be approximately

$$(1,\ 1.33,\ 2.21,\ 4.38,\ 10.1,\ 26.7,\ 79.3,\ 262,\ 950,\ 3760,\ 16200,\ 74800).$$

Asymptotic values as $n \to \infty$ have been deduced by Brightwell, Prömel, and Steger [*J. Combinatorial Theory* **A73** (1996), 193–206], but the limiting behavior is quite different from what happens when n is in a practical range. The values of Q_n were first determined for $n \le 5$ by S. P. Avann [*Aequationes Math.* **8** (1972), 95–102].

92. The basic idea is to introduce dummy elements $n+1$ and $n+2$ with $j \prec n+1$ and $j \prec n+2$ for $1 \le j \le n$, and to find all topological sorts of such an extended relation via adjacent interchanges; then take every *second* permutation, suppressing the dummy elements. An algorithm similar to Algorithm V can be used, but with a recursion that reduces n to $n-2$ by inserting $n-1$ and n among $a_1 \ldots a_{n-2}$ in all possible ways, assuming that $n-1 \not\prec n$, occasionally swapping $n+1$ with $n+2$. [See G. Pruesse and F. Ruskey, *SICOMP* **23** (1994), 373–386. A loopless implementation has been described by Canfield and Williamson, *Order* **12** (1995), 57–75.]

93. The case $n = 3$ illustrates the general idea of a pattern that begins with $1 \ldots (2n)$ and ends with $1(2n)2(2n{-}1)\ldots n(n{+}1)$: 123456, 123546, 123645, 132645, 132546, 132456, 142356, 142536, 142635, 152634, 152436, 152346, 162345, 162435, 162534.

Matchings can also be regarded as involutions of $\{1, \ldots, 2n\}$ that have n cycles. With that representation this pattern involves two transpositions per step.

Notice that the C inversion tables of the permutations just listed are respectively 000000, 000100, 000200, 010200, 010100, 010000, 020000, 020100, 020200, 030200, 030100, 030000, 040000, 040100, 040200. In general, $C_1 = C_3 = \cdots = C_{2n-1} = 0$ and the n-tuples $(C_2, C_4, \ldots, C_{2n})$ run through a reflected Gray code on the radices $(2n-1, 2n-3, \ldots, 1)$. Thus the generation process can easily be made loopless if desired. [See Timothy Walsh, *J. Combinatorial Math. and Combinatorial Computing* **36** (2001), 95–118, Section 1.]

94. Generate inverse permutations with $a'_1 < a'_n > a'_2 < a'_{n-1} > \cdots$, using Algorithm V. (See exercise 5.1.4–23 for the number of solutions.)

95. For example, we can start with $a_1 \ldots a_{n-1}a_n = 2 \ldots n1$ and $b_1 b_2 \ldots b_n b_{n+1} = 12 \ldots n1$, and use Algorithm P to generate the $(n-1)!$ permutations $b_2 \ldots b_n$ of $\{2, \ldots, n\}$. Just after that algorithm swaps $b_i \leftrightarrow b_{i+1}$, we set $a_{b_{i-1}} \leftarrow b_i$, $a_{b_i} \leftarrow b_{i+1}$, $a_{b_{i+1}} \leftarrow b_{i+2}$, and visit $a_1 \ldots a_n$.

96. Use Algorithm X, with $t_k(a_1, \ldots, a_k) =$ '$a_k \ne k$'.

97. Using the notation of exercise 47, we have $N_k = \sum \binom{k}{j}(-1)^j (n-j)^{\underline{k-j}}$ by the method of inclusion and exclusion (exercise 1.3.3–26). If $k = O(\log n)$ then $N_{n-k} = (n!\,e^{-1}/k!)(1 + O(\log n)^2/n)$; hence $A/n! \approx (e-1)/e$ and $B/n! \approx 1$. The number of memory references, under the assumptions of answer 48, is therefore $\approx A + B + 3A + B - N_n + 3A \approx n!(9 - \frac{8}{e}) \approx 6.06\,n!$, about 16.5 per derangement. [See S. G. Akl, *BIT* **20** (1980), 2–7, for a similar method.]

98. Suppose L_n generates $D_n \cup D_{n-1}$, beginning with $(1\ 2\ \ldots\ n)$, then $(2\ 1\ \ldots\ n)$, and ending with $(1\ \ldots\ n{-}1)$; for example, $L_3 = (1\ 2\ 3), (2\ 1\ 3), (1\ 2)$. Then we can generate

D_{n+1} as K_{nn}, ..., K_{n2}, K_{n1}, where $K_{nk} = (1\ 2\ \ldots\ n)^{-k}(n\ n{+}1)L_n(1\ 2\ \ldots\ n)^k$; for example, D_4 is

$$(1\,2\,3\,4),\ (2\,1\,3\,4),\ (1\,2)(3\,4),\ (3\,1\,2\,4),\ (1\,3\,2\,4),\ (3\,1)(2\,4),\ (2\,3\,1\,4),\ (3\,2\,1\,4),\ (2\,3)(1\,4).$$

Notice that K_{nk} begins with the cycle $(k{+}1\ \ldots\ n\ 1\ \ldots\ k\ n{+}1)$ and ends with $(k{+}1\ \ldots\ n\ 1\ \ldots\ k{-}1)(k\ n{+}1)$; so premultiplication by $(k{-}1\ k)$ takes us from K_{nk} to $K_{n(k-1)}$. Also, premultiplication by $(1\ n)$ will return from the last element of D_{n+1} to the first. Premultiplication by $(1\ 2\ n{+}1)$ takes us from the last element of D_{n+1} to $(2\ 1\ 3\ \ldots\ n)$, from which we can return to $(1\ 2\ \ldots\ n)$ by following the cycle for D_n backwards, thereby completing the list L_{n+1} as desired.

99. Use Algorithm X, with $t_k(a_1, \ldots, a_k) =$ '$p > 0$ or $l[q] \neq k+1$'.

Notes: The number of indecomposable permutations is $[z^n]\left(1{-}1/\sum_{k=0}^{\infty} k!\,z^k\right)$; see L. Comtet, *Comptes Rendus Acad. Sci. Paris* **A275** (1972), 569–572. It appears likely that the indecomposable permutations can be generated by adjacent transpositions; for example, when $n = 4$ they are 3142, 3412, 3421, 3241, 2341, 2431, 4231, 4321, 4312, 4132, 4123, 4213, 2413.

100. Here is a lexicographic involution generator analogous to Algorithm X.

Y1. [Initialize.] Set $a_k \leftarrow k$ and $l_{k-1} \leftarrow k$ for $1 \le k \le n$. Then set $l_n \leftarrow 0$, $k \leftarrow 1$.

Y2. [Enter level k.] If $k > n$, visit $a_1 \ldots a_n$ and go to Y3. Otherwise set $p \leftarrow l_0$, $u_k \leftarrow p$, $l_0 \leftarrow l_p$, $k \leftarrow k+1$, and repeat this step. (We have decided to let $a_p = p$.)

Y3. [Decrease k.] Set $k \leftarrow k-1$, and terminate if $k = 0$. Otherwise set $q \leftarrow u_k$ and $p \leftarrow a_q$. If $p = q$, set $l_0 \leftarrow q$, $q \leftarrow 0$, $r \leftarrow l_p$, and $k \leftarrow k+1$ (preparing to make $a_p > p$). Otherwise set $l_{u_{k-1}} \leftarrow q$, $r \leftarrow l_q$ (preparing to make $a_p > q$).

Y4. [Increase a_p.] If $r = 0$ go to Y5. Otherwise set $l_q \leftarrow l_r$, $u_{k-1} \leftarrow q$, $u_k \leftarrow r$, $a_p \leftarrow r$, $a_q \leftarrow q$, $a_r \leftarrow p$, $k \leftarrow k+1$, and go to Y2.

Y5. [Restore a_p.] Set $l_0 \leftarrow p$, $a_p \leftarrow p$, $a_q \leftarrow q$, $k \leftarrow k-1$, and return to Y3. ▮

Let $t_{n+1} = t_n + nt_{n-1}$, $a_{n+1} = 1 + a_n + na_{n-1}$, $t_0 = t_1 = 1$, $a_0 = 0$, $a_1 = 1$. (See Eq. 5.1.4–(40).) Step Y2 is performed t_n times with $k > n$ and a_n times with $k \le n$. Step Y3 is performed a_n times with $p = q$ and $a_n + t_n$ times altogether. Step Y4 is performed $t_n - 1$ times; step Y5, a_n times. The total number of mems for all t_n outputs is therefore approximately $11a_n + 12t_n$, where $a_n < 1.25331414t_n$. (Optimizations are clearly possible if speed is essential.)

101. We construct a list L_n that begins with () and ends with $(n{-}1\ n)$, starting with $L_3 = (),\ (1\ 2),\ (1\ 3),\ (2\ 3)$. If n is odd, L_{n+1} is L_n, K^R_{n1}, K_{n2}, ..., K^R_{nn}, where $K_{nk} = (k\ \ldots\ n)^- L_{n-1}(k\ \ldots n)(k\ n{+}1)$. For example,

$$L_4 = (),\ (1\,2),\ (1\,3),\ (2\,3),\ (2\,3)(1\,4),\ (1\,4),\ (2\,4),\ (1\,3)(2\,4),\ (1\,2)(3\,4),\ (3\,4).$$

If n is even, L_{n+1} is L_n, $K_{n(n-1)}$, $K^R_{n(n-2)}$, ..., K_{n1}, $(1\ n{-}2)L^R_{n-1}(1\ n{-}2)(n\ n{+}1)$.

For further developments, see the article by Walsh cited in answer 93.

102. The following elegant solution by Carla Savage needs only $n-2$ different operations ρ_j, for $1 < j < n$, where ρ_j replaces $a_{j-1}a_ja_{j+1}$ by $a_{j+1}a_{j-1}a_j$ when j is even, $a_ja_{j+1}a_{j-1}$ when j is odd. We may assume that $n \ge 4$; let $A_4 = (\rho_3\rho_2\rho_2\rho_3)^3$. In general A_n will begin and end with ρ_{n-1}, and it will contain $2n-2$ occurrences of ρ_{n-1} altogether. To get A_{n+1}, replace the kth ρ_{n-1} of A_n by $\rho_nA'_n\rho_n$, where $k = 1, 2, 4, \ldots, 2n-2$ if n is even and $k = 1, 3, \ldots, 2n-3, 2n-2$ if n is

odd, and where A'_n is A_n with its first or last element deleted. Then, if we begin with $a_1 \ldots a_n = 1 \ldots n$, the operations ρ_{n-1} of A_n will cause position a_n to run through the successive values $n \to p_1 \to n \to p_2 \to \cdots \to p_{n-1} \to n$, where $p_1 \ldots p_{n-1} = (n{-}1 - [n \text{ even}]) \ldots 4213 \ldots (n{-}1 - [n \text{ odd}])$; the final permutation will again be $1 \ldots n$.

103. (a) A well-balanced permutation has $\sum_{k=1}^n k a_k = n(n+1)^2/4$, an integer.

(b) Replace k by a_k when summing over k.

(c) A fairly fast way to count, when n is not too large, can be based on the streamlined plain-change algorithm of exercise 16, because the quantity $\sum k a_k$ changes in a simple way with each adjacent interchange, and because $n-1$ of every n steps are "hunts" that can be done rapidly. We can save half the work by considering only permutations in which 1 precedes 2. The values for $1 \le n \le 15$ are 0, 0, 0, 2, 6, 0, 184, 936, 6688, 0, 420480, 4298664, 44405142, 0, 6732621476.

104. (a) For each permutation $a_1 \ldots a_n$, insert $\prec$ between a_j and a_{j+1} if $a_j > a_{j+1}$; insert either $\equiv$ or $\prec$ between them if $a_j < a_{j+1}$. (A permutation with k "ascents" therefore yields 2^k weak orders. Weak orders are sometimes called "preferential arrangements; exercise 5.3.1–4 shows that there are approximately $n!/(2(\ln 2)^{n+1})$ of them.)

(b) Start with $a_0 a_1 \ldots a_n a_{n+1} = 01 \ldots 11$. Perform Algorithm L until it stops with $j = 0$. Find k such that $a_1 > \cdots > a_k = a_{k+1}$, and terminate if $k = n$. Otherwise set $a_l \leftarrow a_{k+1} + 1$ for $1 \le l \le k$ and go to step L4. [See M. Mor and A. S. Fraenkel, *Discrete Math.* **48** (1984), 101–112.]

105. All weak ordering sequences can be obtained by a sequence of elementary operations $a_i \leftrightarrow a_j$ or $a_i \leftarrow a_j$. (Perhaps one could actually restrict the transformations further, allowing only $a_j \leftrightarrow a_{j+1}$ or $a_j \leftarrow a_{j+1}$ for $1 \le j < n$.)

106. Every step increases the quantity $\sum_{k=1}^n 2^k [a_k = k]$, as noted by H. S. Wilf, so the game must terminate. At least three approaches to the solution are plausible: one bad, one good, and one better.

The bad one is to play the game on all 13! shuffles and to record the longest. This method does produce the correct answer; but 13! is 6,227,020,800, and the average game lasts ≈ 8.728 steps.

The good one [A. Pepperdine, *Math. Gazette* **73** (1989), 131–133] is to play backwards, starting with the final position $1{*}\ldots{*}$ where $*$ denotes a card that is face down; we will turn a card up only when its value becomes relevant. To move backward from a given position $a_1 \ldots a_n$, consider all $k > 1$ such that either $a_k = k$ or $a_k = *$ and k has not yet turned up. Thus the next-to-last positions are $21{*}\ldots{*}$, $3{*}1{*}\ldots{*}$, $\ldots$, $n{*}\ldots{*}1$. Some positions (like $6{*}{*}213$ for $n = 6$) have no predecessors, even though we haven't turned all the cards up. It is easy to explore the tree of potential backwards games systematically, and one can in fact show that the number of nodes with t $*$'s is exactly $(n-1)!/t!$. Hence the total number of nodes considered is exactly $\lfloor (n-1)!\, e \rfloor$. When $n = 13$ this is 1,302,061,345.

The better one is to play forwards, starting with initial position $*\ldots*$ and turning over the top card when it is face down, running through all $(n-1)!$ permutations of $\{2, \ldots, n\}$ as cards are turned. If the bottom $n-m$ cards are known to be equal to $(m{+}1)(m{+}2)\ldots n$, in that order, at most $f(m)$ further moves are possible; thus we need not pursue a line of play any further if it cannot last long enough to be interesting. A permutation generator like Algorithm X allows us to share the computation for all

permutations with the same prefix and to reject unimportant prefixes. The card in position j need not take the value j when it is turned. When $n = 13$ this method needs to consider only respectively $(1, 11, 940, 6960, 44745, 245083, 1118216, 4112676, 11798207, 26541611, 44380227, 37417359)$ branches at levels $(1, 2, \ldots, 12)$ and to make a total of only 482,663,902 forward moves. Although it repeats some lines of play, the early cutoffs of unprofitable branches make it run more than 11 times faster than the backward method when $n = 13$.

The unique way to attain length 80 is to start with 2 9 4 5 11 12 10 1 8 13 3 6 7.

107. This result holds for any game in which

$$a_1 \ldots a_n \to a_k a_{p(k,2)} \ldots a_{p(k,k-1)} a_1 a_{k+1} \ldots a_n$$

when $a_1 = k$, where $p(k, 2) \ldots p(k, k-1)$ is an arbitrary permutation of $\{2, \ldots, k-1\}$. Suppose a_1 takes on exactly m distinct values $d(1) < \cdots < d(m)$ during a play of the game; we will prove that at most F_{m+1} permutations occur, including the initial shuffle. This assertion is obvious when $m = 1$.

Let $d(j)$ be the initial value of $a_{d(m)}$, where $j < m$, and suppose $a_{d(m)}$ changes on step r. If $d(j) = 1$, the number of permutations is $r + 1 \le F_m + 1 \le F_{m+1}$. Otherwise $r \le F_{m-1}$, and at most F_m further permutations follow step r. [*SIAM Review* **19** (1977), 239–241.]

The values of $f(n)$ for $1 \le n \le 16$ are (0, 1, 2, 4, 7, 10, 16, 22, 30, 38, 51, 65, 80, 101, 113, 139), and they are attainable in respectively (1, 1, 2, 2, 1, 5, 2, 1, 1, 1, 1, 1, 1, 4, 6, 1) ways. The unique longest-winded permutation for $n = 16$ is

$$9\ 12\ 6\ 7\ 2\ 14\ 8\ 1\ 11\ 13\ 5\ 4\ 15\ 16\ 10\ 3.$$

108. The forward method of answer 106 suggests that $f(n)$ probably grows at least as fast as $n \log n$ (by comparison with coupon collecting).

109. For $0 \le j \le 9$ construct the bit vectors $A_j = [a_j \in S_1] \ldots [a_j \in S_m]$ and $B_j = [j \in S_1] \ldots [j \in S_m]$. Then the number of j such that $A_j = v$ must equal the number of k such that $B_k = v$, for all bit vectors v. And if so, the values $\{a_j \mid A_j = v\}$ should be assigned to permutations of $\{k \mid B_k = v\}$ in all possible ways.

For example, the bit vectors in the given problem are

$$(A_0, \ldots, A_9) = (\mathtt{9}, \mathtt{6}, \mathtt{8}, \mathtt{b}, \mathtt{5}, \mathtt{4}, \mathtt{0}, \mathtt{a}, \mathtt{2}, \mathtt{0}), \qquad (B_0, \ldots, B_9) = (\mathtt{5}, \mathtt{0}, \mathtt{8}, \mathtt{6}, \mathtt{2}, \mathtt{a}, \mathtt{4}, \mathtt{b}, \mathtt{9}, \mathtt{0}),$$

in hexadecimal notation; hence $a_0 \ldots a_9 = 8327061549$ or 8327069541.

In a larger problem we would keep the bit vectors in a hash table. It would be better to give the answer in terms of equivalence classes, not permutations; indeed, this problem has comparatively little to do with permutations.

110. In the directed graph with $n!/2$ vertices $a_1 \ldots a_{n-2}$ and $n!$ arcs $a_1 \ldots a_{n-2} \to a_2 \ldots a_{n-1}$ (one for each permutation $a_1 \ldots a_n$), each vertex has in-degree 2 and out-degree 2. Furthermore, from paths like $a_1 \ldots a_{n-2} \to a_2 \ldots a_{n-1} \to a_3 \ldots a_n \to a_4 \ldots a_n a_2 \to a_5 \ldots a_n a_2 a_1 \to \cdots \to a_2 a_1 a_3 \ldots a_{n-2}$, we can see that any vertex is reachable from any other. Therefore an Eulerian circuit exists by Theorem 2.3.4.2D, and such a circuit clearly is equivalent to a universal cycle of permutations. The lexicographically smallest example when $n = 4$ is (123124132134214324314234).

Mark Cooke has pointed out that a universal cycle of permutations is also equivalent to a *Hamiltonian* circuit on the Cayley graph with generators $\sigma = (1\ 2\ \ldots\ n)$ and $\rho = (1\ 2\ \ldots\ n{-}1)$. For example, the cycle just given for $n = 4$ corresponds to $\sigma^3\rho^2\sigma\rho\sigma^2\rho^2\sigma^3\rho\sigma^2\rho^2\sigma\rho\sigma^2\rho$.

111. By exercise 2.3.4.2–22 it suffices to count the oriented trees rooted at $12\ldots(n-2)$, in the digraph of the preceding answer. For $n = 2$, 3, 4, 5, 6, the numbers can be calculated by exercise 2.3.4.2–19, and they turn out to be tantalizingly simple: 1, 3, $2^7 \cdot 3$, $2^{33} \cdot 3^8 \cdot 5^3$, $2^{190} \cdot 3^{49} \cdot 5^{33}$. (Here we consider (121323) to be the same cycle as (213231), but different from (131232).) The graph has symmetries related to Young tableaux in beautiful ways, so the answer to this problem should be quite instructive.

At least one of these cycles must almost surely be easy to describe and to compute, as we did for de Bruijn cycles in Section 7.2.1.1. But no simple construction has yet been found.

INDEX AND GLOSSARY

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information. An answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.

THE ART OF COMPUTER PROGRAMMING

PRE-FASCICLE 2C

A DRAFT OF SECTION 7.2.1.3: GENERATING ALL COMBINATIONS

DONALD E. KNUTH *Stanford University*

ADDISON–WESLEY

Internet page `http://www-cs-faculty.stanford.edu/~knuth/taocp.html` contains current information about this book and related books.

See also `http://www-cs-faculty.stanford.edu/~knuth/sgb.html` for information about *The Stanford GraphBase*, including downloadable software for dealing with the graphs used in many of the examples in Chapter 7.

See also `http://www-cs-faculty.stanford.edu/~knuth/mmixware.html` for downloadable software to simulate the `MMIX` computer.

Zeroth printing (revision 1), 16 June 2002

PREFACE

[The Art of Combinations] has a relation
to almost every species of useful knowledge
that the mind of man can be employed upon.
— JACQUES BERNOULLI, *Ars Conjectandi* (1713)

THIS BOOKLET contains draft material that I'm circulating to experts in the field, in hopes that they can help remove its most egregious errors before too many other people see it. I am also, however, posting it on the Internet for courageous and/or random readers who don't mind the risk of reading a few pages that have not yet reached a very mature state. *Beware:* This material has not yet been proofread as thoroughly as the manuscripts of Volumes 1, 2, and 3 were at the time of their first printings. Those volumes, alas, were subsequently found to contain thousands of mistakes.

Given this caveat, I hope that my errors this time will not be so numerous and/or obtrusive that you will be discouraged from reading the material carefully. I did try to make it both interesting and authoritative, as far as it goes. But the field is so vast, I cannot hope to have surrounded it enough to corral it completely. Therefore I beg you to let me know about any deficiencies you discover.

To put the material in context, this is Section 7.2.1.3 of a long, long chapter on combinatorial algorithms. Chapter 7 will eventually fill three volumes (namely Volumes 4A, 4B, and 4C), assuming that I'm able to remain healthy. It will begin with a short review of graph theory, with emphasis on some highlights of significant graphs in The Stanford GraphBase, from which I will be drawing many examples. Then comes Section 7.1, which deals with the topic of bitwise manipulations. (I drafted about 60 pages about that subject in 1977, but those pages need extensive revision; meanwhile I've decided to work for awhile on the material that follows it, so that I can get a better feel for how much to cut.) Section 7.2 is about generating all possibilities, and it begins with Section 7.2.1: Generating Basic Combinatorial Patterns—which, in turn, begins with Section 7.2.1.1, "Generating all n-tuples," and Section 7.2.1.2, "Generating all permutations." (Readers of the present booklet should have already looked at those sections, drafts of which are available as Prefascicles 2A and 2B.) The stage is now set for the main contents of this booklet, Section 7.2.1.3: "Generating all combinations." Then will come Section 7.2.1.4 (about partitions), etc. Section 7.2.2 will deal with backtracking in general. And so it will go on, if all goes well; an outline of the entire Chapter 7 as currently envisaged appears on the `taocp` webpage that is cited on page ii.

Even the apparently lowly topic of combination generation turns out to be surprisingly rich, with ties to Sections 1.2.1, 1.2.4, 1.2.6, 2.3.2, 2.3.4.2, 4.3.2, 4.6.1, 4.6.2, 5.1.2, 5.4.1, 5.4.2, 6.1, and 6.3 of the first three volumes. I strongly believe in building up a firm foundation, so I have discussed this topic much more thoroughly than I will be able to do with material that is newer or less basic. To my surprise, I came up with 109 exercises, even though — believe it or not — I had to eliminate quite a bit of the interesting material that appears in my files.

Some of the things presented are new, to the best of my knowledge, although I will not be at all surprised to learn that my own little "discoveries" have been discovered before. Please look, for example, at the exercises that I've classed as research problems (rated with difficulty level 46 or higher), namely exercises 53, 55, 66, and 82; I've also implicitly posed additional unsolved questions in the answers to exercises 62, 100, 104, and 108. Are those problems still open? Please let me know if you know of a solution to any of these intriguing questions. And of course if no solution is known today but you do make progress on any of them in the future, I hope you'll let me know.

I urgently need your help also with respect to some exercises that I made up as I was preparing this material. I certainly don't like to get credit for things that have already been published by others, and most of these results are quite natural "fruits" that were just waiting to be "plucked." Therefore please tell me if you know who I should have credited, with respect to the ideas found in exercises 9, 18, 19, 20, 26, 27, 28, 30, 31, 32, 33, 34, 35, 36, 37, 41, 42, 43, 44, 45, 48, 49, 51, 58, 61, 62, 63, 64, 65, 68, 78, 81(b–f), 84, 85, 86, 92, and/or 109.

I shall happily pay a finder's fee of \$2.56 for each error in this draft when it is first reported to me, whether that error be typographical, technical, or historical. The same reward holds for items that I forgot to put in the index. And valuable suggestions for improvements to the text are worth 32¢ each. (Furthermore, if you find a better solution to an exercise, I'll actually reward you with immortal glory instead of mere money, by publishing your name in the eventual book:–)

Happy reading!

Stanford, California D. E. K.
13 June 2002

7.2.1.3. Generating all combinations. Combinatorial mathematics is often described as "the study of permutations, combinations, etc.," so we turn our attention now to combinations. A *combination of n things, taken t at a time*, often called simply a t-combination of n things, is a way to select a subset of size t from a given set of size n. We know from Eq. 1.2.6–(2) that there are exactly $\binom{n}{t}$ ways to do this; and we learned in Section 3.4.2 how to choose t-combinations at random.

Selecting t of n objects is equivalent to choosing the $n-t$ elements not selected. We will emphasize this symmetry by letting

$$n \;=\; s \,+\, t \tag{1}$$

throughout our discussion, and we will often refer to a t-combination of n things as an "(s,t)-combination." Thus, an (s,t)-combination is a way to subdivide $s+t$ objects into two collections of sizes s and t.

> *If I ask how many combinations of 21 can be taken out of 25,*
> *I do in effect ask how many combinations of 4 may be taken.*
> *For there are just as many ways of taking 21 as there are of leaving 4.*
> — AUGUSTUS DE MORGAN, *An Essay on Probabilities* (1838)

There are two main ways to represent (s,t)-combinations: We can list the elements $c_t \ldots c_2 c_1$ that have been selected, or we can work with binary strings $a_{n-1} \ldots a_1 a_0$ for which

$$a_{n-1} + \cdots + a_1 + a_0 \;=\; t. \tag{2}$$

The latter representation has s 0s and t 1s, corresponding to elements that are unselected or selected. The list representation $c_t \ldots c_2 c_1$ tends to work out best if we represent the objects as subsets of the set $\{0, 1, \ldots, n-1\}$ and if we list them in *decreasing* order:

$$n > c_t > \cdots > c_2 > c_1 \geq 0. \tag{3}$$

Binary notation connects these two representations nicely, because the item list $c_t \ldots c_2 c_1$ corresponds to the sum

$$2^{c_t} + \cdots + 2^{c_2} + 2^{c_1} \;=\; \sum_{k=0}^{n-1} a_k 2^k \;=\; (a_{n-1} \ldots a_1 a_0)_2. \tag{4}$$

Of course we could also list the positions $b_s \dots b_2 b_1$ of the 0s in $a_{n-1} \dots a_1 a_0$, where

$$n > b_s > \cdots > b_2 > b_1 \geq 0. \tag{5}$$

Combinations are important not only because subsets are omnipresent in mathematics but also because they are equivalent to many other configurations. For example, every (s,t)-combination corresponds to a combination of $s+1$ things taken t at a time *with repetitions permitted*, also called a *multicombination*, namely a sequence of integers $d_t \dots d_2 d_1$ with

$$s \geq d_t \geq \cdots \geq d_2 \geq d_1 \geq 0. \tag{6}$$

One reason is that $d_t \dots d_2 d_1$ solves (6) if and only if $c_t \dots c_2 c_1$ solves (3), where

$$c_t = d_t + t - 1, \quad \dots, \quad c_2 = d_2 + 1, \quad c_1 = d_1 \tag{7}$$

(see exercise 1.2.6–60). And there is another useful way to relate combinations with repetition to ordinary combinations, suggested by Solomon Golomb [*AMM* **75** (1968), 530–531], namely to define

$$e_j = \begin{cases} c_j, & \text{if } c_j \leq s; \\ e_{c_j - s}, & \text{if } c_j > s. \end{cases} \tag{8}$$

In this form the numbers $e_t \dots e_1$ don't necessarily appear in descending order, but the multiset $\{e_1, e_2, \dots, e_t\}$ is equal to $\{c_1, c_2, \dots, c_t\}$ if and only if $\{e_1, e_2, \dots, e_t\}$ is a set. (See Table 1 and exercise 1.)

An (s,t)-combination is also equivalent to a *composition* of $n+1$ into $t+1$ parts, namely an ordered sum

$$n+1 = p_t + \cdots + p_1 + p_0, \qquad \text{where } p_t, \dots, p_1, p_0 \geq 1. \tag{9}$$

The connection with (3) is now

$$p_t = n - c_t, \quad p_{t-1} = c_t - c_{t-1}, \quad \dots, \quad p_1 = c_2 - c_1, \quad p_0 = c_1 + 1. \tag{10}$$

Equivalently, if $q_j = p_j - 1$, we have

$$s = q_t + \cdots + q_1 + q_0, \qquad \text{where } q_t, \dots, q_1, q_0 \geq 0, \tag{11}$$

a composition of s into $t+1$ *nonnegative* parts, related to (6) by setting

$$q_t = s - d_t, \quad q_{t-1} = d_t - d_{t-1}, \quad \dots, \quad q_1 = d_2 - d_1, \quad q_0 = d_1. \tag{12}$$

Furthermore it is easy to see that an (s,t)-combination is equivalent to a path of length $s+t$ from corner to corner of an $s \times t$ grid, because such a path contains s vertical steps and t horizontal steps. Thus, combinations can be studied in at least eight different guises. Table 1 illustrates all $\binom{6}{3} = 20$ possibilities in the case $s = t = 3$.

These cousins of combinations might seem rather bewildering at first glance, but most of them can be understood directly from the binary representation $a_{n-1} \dots a_1 a_0$. Consider, for example, the "random" bit string

$$a_{23} \dots a_1 a_0 = 01100100100000111111101101, \tag{13}$$

Table 1

THE $(3,3)$-COMBINATIONS AND THEIR EQUIVALENTS

$a_5a_4a_3a_2a_1a_0$	$b_3b_2b_1$	$c_3c_2c_1$	$d_3d_2d_1$	$e_3e_2e_1$	$p_3p_2p_1p_0$	$q_3q_2q_1q_0$	path
000111	543	210	000	210	4111	3000	
001011	542	310	100	310	3211	2100	
001101	541	320	110	320	3121	2010	
001110	540	321	111	321	3112	2001	
010011	532	410	200	010	2311	1200	
010101	531	420	210	020	2221	1110	
010110	530	421	211	121	2212	1101	
011001	521	430	220	030	2131	1020	
011010	520	431	221	131	2122	1011	
011100	510	432	222	232	2113	1002	
100011	432	510	300	110	1411	0300	
100101	431	520	310	220	1321	0210	
100110	430	521	311	221	1312	0201	
101001	421	530	320	330	1231	0120	
101010	420	531	321	331	1222	0111	
101100	410	532	322	332	1213	0102	
110001	321	540	330	000	1141	0030	
110010	320	541	331	111	1132	0021	
110100	310	542	332	222	1123	0012	
111000	210	543	333	333	1114	0003	

which has $s = 11$ zeros and $t = 13$ ones, hence $n = 24$. The dual combination $b_s \ldots b_1$ lists the positions of the zeros, namely

$$23\ 20\ 19\ 17\ 16\ 14\ 13\ 12\ 11\ 4\ 1,$$

because the leftmost position is $n-1$ and the rightmost is 0. The primal combination $c_t \ldots c_1$ lists the positions of the ones, namely

$$22\ 21\ 18\ 15\ 10\ 9\ 8\ 7\ 6\ 5\ 3\ 2\ 0.$$

The corresponding multicombination $d_t \ldots d_1$ lists the number of 0s to the right of each 1:

$$10\ 10\ 8\ 6\ 2\ 2\ 2\ 2\ 2\ 2\ 1\ 1\ 0.$$

The composition $p_t \ldots p_0$ lists the distances between consecutive 1s, if we imagine additional 1s at the left and the right:

$$2\ 1\ 3\ 3\ 5\ 1\ 1\ 1\ 1\ 1\ 2\ 1\ 2\ 1.$$

And the nonnegative composition $q_t \ldots q_0$ counts how many 0s appear between "fenceposts" represented by 1s:

$$1\ 0\ 2\ 2\ 4\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1\ 0;$$

thus we have

$$a_{n-1} \ldots a_1 a_0 = 0^{q_t} 1 0^{q_{t-1}} 1 \ldots 1 0^{q_1} 1 0^{q_0}. \tag{14}$$

The paths in Table 1 also have a simple interpretation (see exercise 2).

Lexicographic generation. Table 1 shows combinations $a_{n-1}\dots a_1a_0$ and $c_t\dots c_1$ in lexicographic order, which is also the lexicographic order of $d_t\dots d_1$. Notice that the dual combinations $b_s\dots b_1$ and the corresponding compositions $p_t\dots p_0$, $q_t\dots q_0$ then appear in *reverse* lexicographic order.

Lexicographic order usually suggests the most convenient way to generate combinatorial configurations. Indeed, Algorithm 7.2.1.2L already solves the problem for combinations in the form $a_{n-1}\dots a_1a_0$, since (s,t)-combinations in bitstring form are the same as permutations of the multiset $\{s\cdot 0, t\cdot 1\}$. That general-purpose algorithm can be streamlined in obvious ways when it is applied to this special case. (See also exercise 7.1–00, which presents a remarkable sequence of seven bitwise operations that will convert any given binary number $(a_{n-1}\dots a_1a_0)_2$ to the lexicographically next t-combination, assuming that n does not exceed the computer's word length.)

Let's focus, however, on generating combinations in the other principal form $c_t\dots c_2c_1$, which is more directly relevant to the ways in which combinations are often needed, and which is more compact than the bit strings when t is small compared to n. In the first place we should keep in mind that a simple sequence of nested loops will do the job nicely when t is very small. For example, when $t=3$ the following instructions suffice:

$$\begin{aligned}&\text{For } c_3 = 2,\ 3,\ \dots,\ n-1 \text{ (in this order) do the following:}\\ &\quad\text{For } c_2 = 1,\ 2,\ \dots,\ c_3-1 \text{ (in this order) do the following:}\\ &\qquad\text{For } c_1 = 0,\ 1,\ \dots,\ c_2-1 \text{ (in this order) do the following:}\\ &\qquad\quad\text{Visit the combination } c_3c_2c_1.\end{aligned}\tag{15}$$

(See the analogous situation in 7.2.1.1–(3).)

On the other hand when t is variable or not so small, we can generate combinations lexicographically by following the general recipe discussed after Algorithm 7.2.1.2L, namely to find the rightmost element c_j that can be increased and then to set the subsequent elements $c_{j-1}\dots c_1$ to their smallest possible values:

Algorithm L (*Lexicographic combinations*). This algorithm generates all t-combinations $c_t\dots c_2c_1$ of the n numbers $\{0,1,\dots,n-1\}$, given $n\ge t\ge 0$. Additional variables c_{t+1} and c_{t+2} are used as sentinels.

L1. [Initialize.] Set $c_j\leftarrow j-1$ for $1\le j\le t$; also set $c_{t+1}\leftarrow n$ and $c_{t+2}\leftarrow 0$.

L2. [Visit.] Visit the combination $c_t\dots c_2c_1$.

L3. [Find j.] Set $j\leftarrow 1$. Then, while $c_j+1=c_{j+1}$, set $c_j\leftarrow j-1$ and $j\leftarrow j+1$; repeat until $c_j+1\ne c_{j+1}$.

L4. [Done?] Terminate the algorithm if $j>t$.

L5. [Increase c_j.] Set $c_j\leftarrow c_j+1$ and return to L2. ∎

The running time of this algorithm is not difficult to analyze. Step L3 sets $c_j\leftarrow j-1$ just after visiting a combination for which $c_{j+1}=c_1+j$, and the number of such combinations is the number of solutions to the inequalities

$$n>c_t>\cdots>c_{j+1}\ge j;\tag{16}$$

but this formula is equivalent to a $(t-j)$-combination of the $n-j$ objects $\{n-1,\ldots,j\}$, so the assignment $c_j \leftarrow j-1$ occurs exactly $\binom{n-j}{t-j}$ times. Summing for $1 \le j \le t$ tells us that the loop in step L3 is performed

$$\binom{n-1}{t-1}+\binom{n-2}{t-2}+\cdots+\binom{n-t}{0} = \binom{n-1}{s}+\binom{n-2}{s}+\cdots+\binom{s}{s} = \binom{n}{s+1} \quad (17)$$

times altogether, or an average of

$$\binom{n}{s+1}\Big/\binom{n}{t} = \frac{n!}{(s+1)!\,(t-1)!}\Big/\frac{n!}{s!\,t!} = \frac{t}{s+1} \quad (18)$$

times per visit. This ratio is less than 1 when $t \le s$, so Algorithm L is quite efficient in such cases.

But the quantity $t/(s+1)$ can be embarrassingly large when t is near n and s is small. Indeed, Algorithm L occasionally sets $c_j \leftarrow j-1$ needlessly, at times when c_j already equals $j-1$. Further scrutiny reveals that we need not always search for the index j that is needed in steps L4 and L5, since the correct value of j can often be predicted from the actions just taken. For example, after we have increased c_4 and reset $c_3c_2c_1$ to their starting values 210, the next combination will inevitably increase c_3. These observations lead to a tuned-up version of the algorithm:

Algorithm T (*Lexicographic combinations*). This algorithm is like Algorithm L, but faster. It also assumes, for convenience, that $t < n$.

T1. [Initialize.] Set $c_j \leftarrow j-1$ for $1 \le j \le t$; then set $c_{t+1} \leftarrow n$, $c_{t+2} \leftarrow 0$, and $j \leftarrow t$.

T2. [Visit.] (At this point j is the smallest index such that $c_{j+1} > j$.) Visit the combination $c_t \ldots c_2c_1$. Then, if $j > 0$, set $x \leftarrow j$ and go to step T6.

T3. [Easy case?] If $c_1 + 1 < c_2$, set $c_1 \leftarrow c_1 + 1$ and return to T2. Otherwise set $j \leftarrow 2$.

T4. [Find j.] Set $c_{j-1} \leftarrow j-1$ and $x \leftarrow c_j + 1$. If $x = c_{j+1}$, set $j \leftarrow j+1$ and repeat this step until $x < c_{j+1}$.

T5. [Done?] Terminate the algorithm if $j > t$.

T6. [Increase c_j.] Set $c_j \leftarrow x$, $j \leftarrow j-1$, and return to T2. ∎

Now $j = 0$ in step T2 if and only if $c_1 > 0$, so the assignments in step T4 are never redundant. Exercise 6 carries out a complete analysis of Algorithm T.

Notice that the parameter n appears only in the initialization steps L1 and T1, not in the principal parts of Algorithms L and T. Thus we can think of the process as generating the first $\binom{n}{t}$ combinations of an *infinite* list, which depends only on t. This simplification arises because the list of t-combinations for $n+1$ things begins with the list for n things, under our conventions; we have been using lexicographic order on the decreasing sequences $c_t \ldots c_1$ for this very reason, instead of working with the increasing sequences $c_1 \ldots c_t$.

Derrick Lehmer noticed another pleasant property of Algorithms L and T [*Applied Combinatorial Mathematics*, edited by E. F. Beckenbach (1964), 27–30]:

Theorem L. *The combination $c_t \ldots c_2 c_1$ is visited after exactly*

$$\binom{c_t}{t} + \cdots + \binom{c_2}{2} + \binom{c_1}{1} \tag{19}$$

other combinations have been visited.

Proof. There are $\binom{c_k}{k}$ combinations $c'_t \ldots c'_2 c'_1$ with $c'_j = c_j$ for $t \ge j > k$ and $c'_k < c_k$, namely $c_t \ldots c_{k+1}$ followed by the k-combinations of $\{0, \ldots, c_k - 1\}$. ∎

When $t = 3$, for example, the numbers

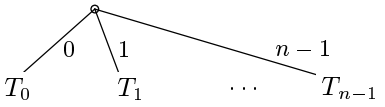
$$\tbinom{2}{3} + \tbinom{1}{2} + \tbinom{0}{1},\ \tbinom{3}{3} + \tbinom{1}{2} + \tbinom{0}{1},\ \tbinom{3}{3} + \tbinom{2}{2} + \tbinom{0}{1},\ \ldots,\ \tbinom{5}{3} + \tbinom{4}{2} + \tbinom{3}{1}$$

that correspond to the combinations $c_3 c_2 c_1$ in Table 1 simply run through the sequence 0, 1, 2, ..., 19. Theorem L gives us a nice way to understand the *combinatorial number system* of degree t, which represents every nonnegative integer N uniquely in the form

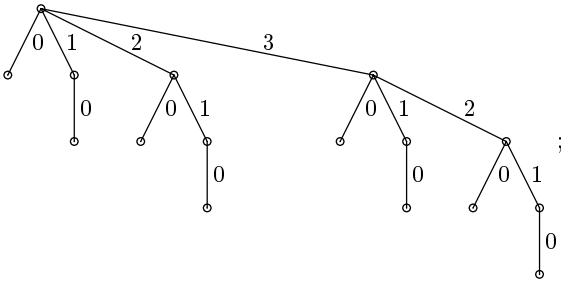
$$N = \binom{n_t}{t} + \cdots + \binom{n_2}{2} + \binom{n_1}{1}, \qquad n_t > \cdots > n_2 > n_1 \ge 0. \tag{20}$$

Binomial trees. The family of trees T_n defined by

$$T_0 = \circ\ , \qquad T_n = \quad \text{for } n > 0, \tag{21}$$



arises in several important contexts and sheds further light on combination generation. For example, T_4 is



; (22)

and T_5, rendered more artistically, appears as the frontispiece to Volume 1 of this series of books.

Notice that T_n is like T_{n-1}, except for an additional copy of T_{n-1}; therefore T_n has 2^n nodes altogether. Furthermore, the number of nodes on level t is the binomial coefficient $\binom{n}{t}$; this fact accounts for the name "binomial tree." Indeed, the sequence of labels encountered on the path from the root to each node on level t defines a combination $c_t \ldots c_1$, and all combinations occur in lexicographic order from left to right. Thus, Algorithms L and T can be regarded as procedures to traverse the nodes on level t of the binomial tree T_n.

The infinite binomial tree T_∞ is obtained by letting $n \to \infty$ in (21). The root of this tree has infinitely many branches, but every other node is the root

of a finite binomial subtree. All possible t-combinations appear in lexicographic order on level t of T_∞.

Let's get more familiar with binomial trees by considering all possible ways to pack a rucksack. More precisely, suppose we have n items that take up respectively $w_{n-1}, \ldots, w_1, w_0$ units of capacity, where

$$w_{n-1} \geq \cdots \geq w_1 \geq w_0; \tag{23}$$

we want to generate all binary vectors $a_{n-1} \ldots a_1 a_0$ such that

$$a \cdot w = a_{n-1}w_{n-1} + \cdots + a_1 w_1 + a_0 w_0 \leq N, \tag{24}$$

where N is the total capacity of a rucksack. Equivalently, we want to find all subsets C of $\{0, 1, \ldots, n-1\}$ such that $w(C) = \sum_{c\in C} w_c \leq N$; such subsets will be called *feasible*. We will write a feasible subset as $c_1 \ldots c_t$, where $c_1 > \cdots > c_t \geq 0$, numbering the subscripts differently from the convention of (3) above because t is variable in this problem.

Every feasible subset corresponds to a node of T_n, and our goal is to visit each feasible node. Clearly the parent of every feasible node is feasible, and so is the left sibling, if any; therefore a simple tree exploration procedure solves the problem.

Algorithm F (*Filling a rucksack*). This algorithm generates all feasible ways $c_1 \ldots c_t$ to fill a rucksack, given $w_{n-1}, \ldots, w_1, w_0$, and N. We let $\delta_j = w_j - w_{j-1}$ for $1 \leq j < n$.

F1. [Initialize.] Set $t \leftarrow 0$, $c_0 \leftarrow n$, and $r \leftarrow N$.

F2. [Visit.] Visit the combination $c_1 \ldots c_t$, which uses $N - r$ units of capacity.

F3. [Try to add w_0.] If $c_t > 0$ and $r \geq w_0$, set $t \leftarrow t + 1$, $c_t \leftarrow 0$, $r \leftarrow r - w_0$, and return to F2.

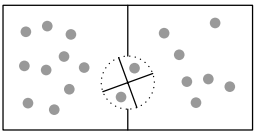
F4. [Try to increase c_t.] Terminate if $t = 0$. Otherwise, if $c_{t-1} > c_t + 1$ and $r \geq \delta_{c_t+1}$, set $c_t \leftarrow c_t + 1$, $r \leftarrow r - \delta_{c_t}$, and return to F2.

F5. [Remove c_t.] Set $r \leftarrow r + w_{c_t}$, $t \leftarrow t - 1$, and return to F4. ∎

Notice that the algorithm implicitly visits nodes of T_n in preorder, skipping over unfeasible subtrees. An element $c > 0$ is never placed in the rucksack until after the procedure has explored all possibilities using element $c-1$. The running time is proportional to the number of feasible combinations visited (see exercise 20).

Incidentally, the classical "knapsack problem" of operations research is different: It asks for a feasible subset C such that $v(C) = \sum_{c\in C} v(c)$ is maximum, where each item c has been assigned a value $v(c)$. Algorithm F is not a particularly good way to solve that problem, because it often considers cases that could be ruled out. For example, if C and C' are subsets of $\{1, \ldots, n-1\}$ with $w(C) \leq w(C') \leq N - w_0$ and $v(C) \geq v(C')$, Algorithm F will examine both $C \cup \{0\}$ and $C' \cup \{0\}$, but the latter subset will never improve the maximum. We will consider methods for the classical knapsack problem later; Algorithm F is intended only for situations when *all* of the feasible possibilities are potentially relevant.

Gray codes for combinations. Instead of merely generating all combinations, we often prefer to visit them in such a way that each one is obtained by making only a small change to its predecessor.



For example, we can ask for what Nijenhuis and Wilf have called a "revolving door algorithm": Imagine two rooms that contain respectively s and t people, with a revolving door between them. Whenever a person goes into the opposite room, somebody else comes out. Can we devise a sequence of moves so that each (s,t)-combination occurs exactly once?

The answer is yes, and in fact a huge number of such patterns exist. For example, it turns out that if we examine all n-bit strings $a_{n-1}\ldots a_1a_0$ in the well-known order of Gray binary code (Section 7.2.1.1), but select only those that have exactly s 0s and t 1s, the resulting strings form a revolving door code.

Here's the proof: Gray binary code is defined by the recurrence $\Gamma_n = 0\Gamma_{n-1}$, $1\Gamma_{n-1}^R$ of 7.2.1.1–(5), so its (s,t) subset satisfies the recurrence

$$\Gamma_{st} \;=\; 0\Gamma_{(s-1)t},\; 1\Gamma_{s(t-1)}^R \tag{25}$$

when $st > 0$. We also have $\Gamma_{s0} = 0^s$ and $\Gamma_{0t} = 1^t$. Therefore it is clear by induction that Γ_{st} begins with 0^s1^t and ends with 10^s1^{t-1} when $st > 0$. The transition at the comma in (25) is from the last element of $0\Gamma_{(s-1)t}$ to the last element of $1\Gamma_{s(t-1)}$, namely from $010^{s-1}1^{t-1} = 010^{s-1}11^{t-2}$ to $110^s1^{t-2} = 110^{s-1}01^{t-2}$ when $t \ge 2$, and this satisfies the revolving-door constraint. The case $t = 1$ also checks out. For example, Γ_{33} is given by the columns of

000111	011010	110001	101010
001101	011100	110010	101100
001110	010101	110100	100101
001011	010110	111000	100110
011001	010011	101001	100011

(26)

and Γ_{23} can be found in the first two columns of this array. One more turn of the door takes the last element into the first. [These properties of Γ_{st} were discovered by D. T. Tang and C. N. Liu, *IEEE Trans.* **C-22** (1973), 176–180; a loopless implementation was presented by J. R. Bitner, G. Ehrlich, and E. M. Reingold, *CACM* **19** (1976), 517–521.]

When we convert the bitstrings $a_5a_4a_3a_2a_1a_0$ in (26) to the corresponding index-list forms $c_3c_2c_1$, a striking pattern becomes evident:

210	431	540	531
320	432	541	532
321	420	542	520
310	421	543	521
430	410	530	510

(27)

The first components c_3 occur in increasing order; but for each fixed value of c_3, the values of c_2 occur in *decreasing* order. And for fixed c_3c_2, the values of c_1 are again increasing. The same is true in general: *All combinations* $c_t\ldots c_2c_1$

appear in lexicographic order of

$$(c_t,\ -c_{t-1},\ c_{t-2},\ \ldots,\ (-1)^t c_1) \tag{28}$$

in the revolving-door Gray code Γ_{st}. This property follows by induction, because (25) becomes

$$\Gamma_{st} \ = \ \Gamma_{(s-1)t},\ (s+t-1)\Gamma^R_{s(t-1)} \tag{29}$$

for $st > 0$ when we use index-list notation instead of bitstring notation. Consequently the sequence can be generated efficiently by the following algorithm due to W. H. Payne [see *ACM Trans. Math. Software* **5** (1979), 163–172]:

Algorithm R (*Revolving-door combinations*). This algorithm generates all t-combinations $c_t \ldots c_2 c_1$ of $\{0, 1, \ldots, n-1\}$ in lexicographic order of the alternating sequence (28), assuming that $n > t > 1$. Step R3 has two variants, depending on whether t is even or odd.

R1. [Initialize.] Set $c_j \leftarrow j-1$ for $t \geq j \geq 1$, and $c_{t+1} \leftarrow n$.

R2. [Visit.] Visit the combination $c_t \ldots c_2 c_1$.

R3. [Easy case?] If t is odd: If $c_1 + 1 < c_2$, increase c_1 by 1 and return to R2, otherwise set $j \leftarrow 2$ and go to R4. If t is even: If $c_1 > 0$, decrease c_1 by 1 and return to R2, otherwise set $j \leftarrow 2$ and go to R5.

R4. [Try to decrease c_j.] (At this point $c_j = c_{j-1} + 1$.) If $c_j \geq j$, set $c_j \leftarrow c_{j-1}$, $c_{j-1} \leftarrow j-2$, and return to R2. Otherwise increase j by 1.

R5. [Try to increase c_j.] (At this point $c_{j-1} = j-2$.) If $c_j + 1 < c_{j+1}$, set $c_{j-1} \leftarrow c_j$, $c_j \leftarrow c_j + 1$, and return to R2. Otherwise increase j by 1, and go to R4 if $j \leq t$. ∎

Exercises 21–25 explore further properties of this interesting sequence. One of them is a nice companion to Theorem L: *The combination* $c_t c_{t-1} \ldots c_2 c_1$ *is visited by Algorithm R after exactly*

$$N = \binom{c_t+1}{t} - \binom{c_{t-1}+1}{t-1} + \cdots + (-1)^t \binom{c_2+1}{2} - (-1)^t \binom{c_1+1}{1} - [t \text{ odd}] \tag{30}$$

other combinations have been visited. We may call this the representation of N in the "alternating combinatorial number system" of degree t; one consequence, for example, is that every positive integer has a unique representation of the form $N = \binom{a}{3} - \binom{b}{2} + \binom{c}{1}$ with $a > b > c > 0$. Algorithm R tells us how to add 1 to N in this system.

Although the strings of (26) and (27) are not in lexicographic order, they are examples of a more general concept called *genlex order*, a name coined by Timothy Walsh. A sequence of strings $\alpha_1, \ldots, \alpha_N$ is said to be in genlex order when all strings with a common prefix occur consecutively. For example, all 3-combinations that begin with 53 appear together in (27).

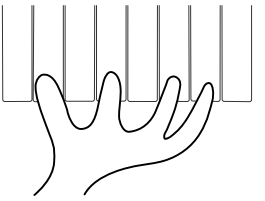
Genlex order means that the strings can be arranged in a trie structure, as in Fig. 31 of Section 6.3, but with the children of each node ordered arbitrarily. When a trie is traversed in any order such that each node is visited just before or just after its descendants, all nodes with a common prefix—that is, all nodes of

a subtrie — appear consecutively. This principle makes genlex order convenient, because it corresponds to recursive generation schemes. Many of the algorithms we have seen for generating n-tuples have therefore produced their results in some version of genlex order; similarly, the method of "plain changes" (Algorithm 7.2.1.2P) visits permutations in a genlex order of the corresponding inversion tables.

The revolving-door method of Algorithm R is a genlex routine that changes only one element of the combination at each step. But it isn't totally satisfactory, because it frequently must change two of the indices c_j simultaneously, in order to preserve the condition $c_t > \cdots > c_2 > c_1$. For example, Algorithm R changes 210 into 320, and (27) includes nine such "crossing" moves.

The source of this defect can be traced to our proof that (25) satisfies the revolving-door property: We observed that the string $010^{s-1}11^{t-2}$ is followed by $110^{s-1}01^{t-2}$ when $t \geq 2$. Hence the recursive construction Γ_{st} involves transitions of the form $110^a0 \leftrightarrow 010^a1$, when a substring like 11000 is changed to 01001 or vice versa; the two 1s cross each other.

A Gray path for combinations is said to be *homogeneous* if it changes only one of the indices c_j at each step. A homogeneous scheme is characterized in bitstring form by having only transitions of the forms $10^a \leftrightarrow 0^a1$ within strings, for $a \geq 1$, when we pass from one string to the next. With a homogeneous scheme we can, for example, play all t-note chords on an n-note keyboard by moving only one finger at a time.



A slight modification of (25) yields a genlex scheme for (s,t)-combinations that is pleasantly homogeneous. The basic idea is to construct a sequence that begins with 0^s1^t and ends with 1^t0^s, and the following recursion suggests itself almost immediately: Let $K_{s0} = 0^s$, $K_{0t} = 1^t$, $K_{s(-1)} = \emptyset$, and

$$K_{st} \;=\; 0K_{(s-1)t},\; 10K^R_{(s-1)(t-1)},\; 11K_{s(t-2)} \quad \text{for } st > 0. \tag{31}$$

At the commas of this sequence we have 01^t0^{s-1} followed by $101^{t-1}0^{s-1}$, and 10^s1^{t-1} followed by 110^s1^{t-2}; both of these transitions are homogeneous, although the second one requires the 1 to jump across s 0s. The combinations K_{33} for $s = t = 3$ are

000111	010101	101100	100011
001011	010011	101001	110001
001101	011001	101010	110010
001110	011010	100110	110100
010110	011100	100101	111000

(32)

in bitstring form, and the corresponding "finger patterns" are

210	420	532	510
310	410	530	540
320	430	531	541
321	431	521	542
421	432	520	543

(33)

When a homogeneous scheme for ordinary combinations $c_t \dots c_1$ is converted to the corresponding scheme (6) for combinations with repetitions $d_t \dots d_1$, it retains the property that only one of the indices d_j changes at each step. And when it is converted to the corresponding schemes (9) or (11) for compositions $p_t \dots p_0$ or $q_t \dots q_0$, only two (adjacent) parts change when c_j changes.

Near-perfect schemes. But we can do even better! All (s,t)-combinations can be generated by a sequence of strongly homogeneous transitions that are either $01 \leftrightarrow 10$ or $001 \leftrightarrow 100$. In other words, we can insist that each step causes a single index c_j to change by at most 2. Let's call such generation schemes *near-perfect*.

Imposing such strong conditions actually makes it easier to discover near-perfect schemes, because comparatively few choices are available. Indeed, if we restrict ourselves to genlex methods that are near-perfect on n-bit strings, T. A. Jenkyns and D. McCarthy observed that all such methods can be easily characterized [*Ars Combinatoria* **40** (1995), 153–159]:

Theorem N. *If $st > 0$, there are exactly $2s$ near-perfect ways to list all (s,t)-combinations in a genlex order. In fact, when $1 \le a \le s$, there is exactly one such listing, N_{sta}, that begins with 1^t0^s and ends with $0^a1^t0^{s-a}$; the other s possibilities are the reverse lists, N^R_{sta}.*

Proof. The result certainly holds when $s = t = 1$; otherwise we use induction on $s + t$. The listing N_{sta}, if it exists, must have the form $1X_{s(t-1)}$, $0Y_{(s-1)t}$ for some near-perfect genlex listings $X_{s(t-1)}$ and $Y_{(s-1)t}$. If $t = 1$, $X_{s(t-1)}$ is the single string 0^s; hence $Y_{(s-1)t}$ must be $N_{(s-1)1(a-1)}$ if $a > 1$, $N_{(s-1)11}$ if $a = 1$. On the other hand if $t > 1$, the near-perfect condition implies that the last string of $X_{s(t-1)}$ cannot begin with 1; hence $X_{s(t-1)} = N_{s(t-1)b}$ for some b. If $a > 1$, $Y_{(s-1)t}$ must be $N_{(s-1)t(a-1)}$, hence b must be 1; similarly, b must be 1 if $s = 1$. Otherwise we have $a = 1 < s$, and this forces $Y_{(s-1)t} = N^R_{(s-1)tc}$ for some c. The transition from $10^b1^{t-1}0^{s-b}$ to $0^{c+1}1^t0^{s-1-c}$ is near-perfect only if $c = 1$ and $b = 2$. ∎

The proof of Theorem N yields the following recursive formulas when $st > 0$:

$$N_{sta} = \begin{cases} 1N_{s(t-1)1},\ 0N_{(s-1)t(a-1)}, & \text{if } 1 < a \le s; \\ 1N_{s(t-1)2},\ 0N^R_{(s-1)t1}, & \text{if } 1 = a < s; \\ 1N_{1(t-1)1},\ 01^t, & \text{if } 1 = a = s. \end{cases} \tag{34}$$

Also, of course, $N_{s0a} = 0^s$.

Let us set $A_{st} = N_{st1}$ and $B_{st} = N_{st2}$. These near-perfect listings, discovered by Phillip J. Chase in 1976, have the net effect of shifting a leftmost block of 1s to the right by one or two positions, respectively, and they satisfy the following mutual recursions:

$$A_{st} = 1B_{s(t-1)},\ 0A^R_{(s-1)t}; \qquad B_{st} = 1A_{s(t-1)},\ 0A_{(s-1)t}. \tag{35}$$

"To take one step forward, take two steps forward, then one step backward; to take two steps forward, take one step forward, then another." These equations

Table 2

CHASE'S SEQUENCES FOR (3, 3)-COMBINATIONS

$A_{33} = \widehat{C}_{33}^R$				$B_{33} = C_{33}$			
543	531	321	420	543	520	432	410
541	530	320	421	542	510	430	210
540	510	310	431	540	530	431	310
542	520	210	430	541	531	421	320
532	521	410	432	521	532	420	321

hold for all integer values of s and t, if we define A_{st} and B_{st} to be $\emptyset$ when s or t is negative, except that $A_{00} = B_{00} = \epsilon$ (the empty string). Thus A_{st} actually takes $\min(s, 1)$ forward steps, and B_{st} actually takes $\min(s, 2)$. For example, Table 2 shows the relevant listings for $s = t = 3$, using an equivalent index-list form $c_3c_2c_1$ instead of the bit strings $a_5a_4a_3a_2a_1a_0$.

Chase noticed that a computer implementation of these sequences becomes simpler if we define

$$C_{st} = \begin{cases} A_{st}, & \text{if } s+t \text{ is odd;} \\ B_{st}, & \text{if } s+t \text{ is even;} \end{cases} \qquad \widehat{C}_{st} = \begin{cases} A_{st}^R, & \text{if } s+t \text{ is even;} \\ B_{st}^R, & \text{if } s+t \text{ is odd.} \end{cases} \tag{36}$$

[See *Congressus Numerantium* **69** (1989), 215–242.] Then we have

$$C_{st} = \begin{cases} 1C_{s(t-1)},\ 0\widehat{C}_{(s-1)t}, & \text{if } s+t \text{ is odd;} \\ 1C_{s(t-1)},\ 0C_{(s-1)t}, & \text{if } s+t \text{ is even;} \end{cases} \tag{37}$$

$$\widehat{C}_{st} = \begin{cases} 0C_{(s-1)t},\ 1\widehat{C}_{s(t-1)}, & \text{if } s+t \text{ is even;} \\ 0\widehat{C}_{(s-1)t},\ 1\widehat{C}_{s(t-1)}, & \text{if } s+t \text{ is odd.} \end{cases} \tag{38}$$

When bit a_j is ready to change, we can tell where we are in the recursion by testing whether j is even or odd.

Indeed, the sequence C_{st} can be generated by a surprisingly simple algorithm, based on general ideas that apply to *any* genlex scheme. Let us say that bit a_j is *active* in a genlex algorithm if it is supposed to change before anything to its left is altered. (The node for an active bit in the corresponding trie is not the rightmost child of its parent.) Suppose we have an auxiliary table $w_n \ldots w_1 w_0$, where $w_j = 1$ if and only if either a_j is active or $j < r$, where r is the least subscript such that $a_r \neq a_0$; we also let $w_n = 1$. Then the following method will find the successor of $a_{n-1} \ldots a_1 a_0$:

> Set $j \leftarrow r$. If $w_j = 0$, set $w_j \leftarrow 1$, $j \leftarrow j+1$, and repeat until $w_j = 1$. Terminate if $j = n$; otherwise set $w_j \leftarrow 0$. Change a_j to $1 - a_j$, and make any other changes to $a_{j-1} \ldots a_0$ and r that apply to the particular genlex scheme being used. (39)

The beauty of this approach comes from the fact that the loop is guaranteed to be efficient: We can prove that the operation $j \leftarrow j+1$ will be performed less than once per generation step, on the average (see exercise 36).

By analyzing the transitions that occur when bits change in (37) and (38), we can readily flesh out the remaining details:

Algorithm C (*Chase's sequence*). This algorithm visits all (s,t)-combinations $a_{n-1}\ldots a_1a_0$, where $n=s+t$, in the near-perfect order of Chase's sequence C_{st}.

C1. [Initialize.] Set $a_j\leftarrow 0$ for $0\le j<s$, $a_j\leftarrow 1$ for $s\le j<n$, and $w_j\leftarrow 1$ for $0\le j\le n$. If $s>0$, set $r\leftarrow s$; otherwise set $r\leftarrow t$.

C2. [Visit.] Visit the combination $a_{n-1}\ldots a_1a_0$.

C3. [Find j and branch.] Set $j\leftarrow r$. If $w_j=0$, set $w_j\leftarrow 1$, $j\leftarrow j+1$, and repeat until $w_j=1$. Terminate if $j=n$; otherwise set $w_j\leftarrow 0$ and make a four-way branch: Go to C4 if j is odd and $a_j\neq 0$, to C5 if j is even and $a_j\neq 0$, to C6 if j is even and $a_j=0$, to C7 if j is odd and $a_j=0$.

C4. [Move right one.] Set $a_{j-1}\leftarrow 1$, $a_j\leftarrow 0$. If $r=j>1$, set $r\leftarrow j-1$; otherwise if $r=j-1$ set $r\leftarrow j$. Return to C2.

C5. [Move right two.] If $a_{j-2}\neq 0$, go to C4. Otherwise set $a_{j-2}\leftarrow 1$, $a_j\leftarrow 0$. If $r=j$, set $r\leftarrow\max(j-2,1)$; otherwise if $r=j-2$, set $r\leftarrow j-1$. Return to C2.

C6. [Move left one.] Set $a_j\leftarrow 1$, $a_{j-1}\leftarrow 0$. If $r=j>1$, set $r\leftarrow j-1$; otherwise if $r=j-2$ set $r\leftarrow j-1$. Return to C2.

C7. [Move left two.] If $a_{j-1}\neq 0$, go to C6. Otherwise set $a_j\leftarrow 1$, $a_{j-2}\leftarrow 0$. If $r=j-2$, set $r\leftarrow j$; otherwise if $r=j-1$, set $r\leftarrow j-2$. Return to C2. ∎

***Analysis of Chase's sequence.** The magical properties of Algorithm C cry out for further exploration, and a closer look turns out to be quite instructive. Given a bit string $a_{n-1}\ldots a_1a_0$, let us define $a_n=1$, $u_n=n \bmod 2$, and

$$u_j=a_{j+1}(1-u_{j+1}),\quad v_j=(u_j+j)\bmod 2,\quad w_j=(v_j+a_j)\bmod 2, \tag{40}$$

for $n>j\ge 0$. For example, we might have $n=26$ and

$$\begin{aligned} a_{25}\ldots a_1a_0&=11001001000011111101101010,\\ u_{25}\ldots u_1u_0&=10100100100001010100100101,\\ v_{25}\ldots v_1v_0&=00001110001011111110001111,\\ w_{25}\ldots w_1w_0&=11000111001000000011100101. \end{aligned} \tag{41}$$

With these definitions we can prove by induction that $v_j=0$ if and only if bit a_j is being "controlled" by C rather than by $\widehat{C}$ in the recursions (37)–(38) that generate $a_{n-1}\ldots a_1a_0$, except when a_j is part of the final run of 0s or 1s at the right end. Therefore w_j agrees with the value computed by Algorithm C at the moment when $a_{n-1}\ldots a_1a_0$ is visited, for $r\le j<n$. These formulas can be used to determine exactly where a given combination appears in Chase's sequence (see exercise 39).

If we want to work with the index-list form $c_t\ldots c_2c_1$ instead of the bit strings $a_{n-1}\ldots a_1a_0$, it is convenient to change the notation slightly, writing

$C_t(n)$ for C_{st} and $\widehat{C}_t(n)$ for $\widehat{C}_{st}$ when $s+t=n$. Then $C_0(n)=\widehat{C}_0(n)=\epsilon$, and the recursions for $t\ge 0$ take the form

$$C_{t+1}(n+1)=\begin{cases} nC_t(n),\ \widehat{C}_{t+1}(n), & \text{if } n \text{ is even;}\\ nC_t(n),\ C_{t+1}(n), & \text{if } n \text{ is odd;}\end{cases} \tag{42}$$

$$\widehat{C}_{t+1}(n+1)=\begin{cases} C_{t+1}(n),\ n\widehat{C}_t(n), & \text{if } n \text{ is odd;}\\ \widehat{C}_{t+1}(n),\ n\widehat{C}_t(n), & \text{if } n \text{ is even.}\end{cases} \tag{43}$$

These new equations can be expanded to tell us, for example, that

$$\begin{aligned} C_{t+1}(9) &= 8C_t(8),\ 6C_t(6),\ 4C_t(4),\ \ldots,\ 3\widehat{C}_t(3),\ 5\widehat{C}_t(5),\ 7\widehat{C}_t(7);\\ C_{t+1}(8) &= 7C_t(7),\ 6C_t(6),\ 4C_t(4),\ \ldots,\ 3\widehat{C}_t(3),\ 5\widehat{C}_t(5);\\ \widehat{C}_{t+1}(9) &= \quad 6C_t(6),\ 4C_t(4),\ \ldots,\ 3\widehat{C}_t(3),\ 5\widehat{C}_t(5),\ 7\widehat{C}_t(7),\ 8\widehat{C}_t(8);\\ \widehat{C}_{t+1}(8) &= \quad 6C_t(6),\ 4C_t(4),\ \ldots,\ 3\widehat{C}_t(3),\ 5\widehat{C}_t(5),\ 7\widehat{C}_t(7); \end{aligned} \tag{44}$$

notice that the same pattern predominates in all four sequences. The meaning of "..." in the middle depends on the value of t: We simply omit all terms $nC_t(n)$ and $n\widehat{C}_t(n)$ where $n<t$.

Except for edge effects at the very beginning or end, all of the expansions in (44) are based on the infinite progression

$$\ldots,\ 10,\ 8,\ 6,\ 4,\ 2,\ 0,\ 1,\ 3,\ 5,\ 7,\ 9,\ \ldots, \tag{45}$$

which is a natural way to arrange the nonnegative integers into a doubly infinite sequence. If we omit all terms of (45) that are $<t$, given any integer $t\ge 0$, the remaining terms retain the property that adjacent elements differ by either 1 or 2. Richard Stanley has suggested the name *endo-order* for this sequence, because we can remember it by thinking "even numbers decreasing, odd ...". (Notice that if we retain only the terms less than N and complement with respect to N, endo-order becomes organ-pipe order; see exercise 6.1–18.)

We could program the recursions of (42) and (43) directly, but it is interesting to unwind them using (44), thus obtaining an iterative algorithm analogous to Algorithm C. The result needs only $O(t)$ memory locations, and it is especially efficient when t is relatively small compared to n. Exercise 45 contains the details.

***Near-perfect multiset permutations.** Chase's sequences lead in a natural way to an algorithm that will generate permutations of any multiset $\{s_0\cdot 0, s_1\cdot 1, \ldots, s_d\cdot d\}$ in a near-perfect manner, meaning that

i) every transition is either $a_{j+1}a_j \leftrightarrow a_ja_{j+1}$ or $a_{j+2}a_{j+1}a_j \leftrightarrow a_ja_{j+1}a_{j+2}$;

ii) transitions of the second kind have $a_{j+1}=\min(a_j,a_{j+2})$.

Algorithm C tells us how to do this when $d=1$, and we can extend it to larger values of d by the following recursive construction [*CACM* **13** (1970), 368–369, 376]: Suppose

$$\alpha_0,\ \alpha_1,\ \ldots,\ \alpha_{N-1}$$

is any near-perfect listing of the permutations of $\{s_1 \cdot 1, \ldots, s_d \cdot d\}$. Then Algorithm C, with $s = s_0$ and $t = s_1 + \cdots + s_d$, tells us how to generate a listing

$$\Lambda_j \;=\; \alpha_j 0^s,\; \ldots,\; 0^a \alpha_j 0^{s-a} \tag{46}$$

in which all transitions are $0x \leftrightarrow x0$ or $00x \leftrightarrow x00$; the final entry has $a = 1$ or 2 leading zeros, depending on s and t. Therefore all transitions of the sequence

$$\Lambda_0,\; \Lambda_1^R,\; \Lambda_2,\; \ldots,\; (\Lambda_{N-1} \text{ or } \Lambda_{N-1}^R) \tag{47}$$

are near-perfect; and this list clearly contains all the permutations.

For example, the permutations of $\{0, 0, 0, 1, 1, 2\}$ generated in this way are

211000, 210100, 210001, 210010, 200110, 200101, 200011, 201001, 201010, 201100,
021100, 021001, 021010, 020110, 020101, 020011, 000211, 002011, 002101, 002110,
001210, 001201, 001021, 000121, 010021, 010201, 010210, 012010, 012001, 012100,
102100, 102010, 102001, 100021, 100201, 100210, 120010, 120001, 120100, 121000,
112000, 110200, 110002, 110020, 100120, 100102, 100012, 101002, 101020, 101200,
011200, 011002, 011020, 010120, 010102, 010012, 000112, 001012, 001102, 001120.

***Perfect schemes.** Why should we settle for a near-perfect generator like C_{st}, instead of insisting that all transitions have the simplest possible form $01 \leftrightarrow 10$?

One reason is that perfect schemes don't always exist. For example, we observed in 7.2.1.2–(2) that there is no way to generate all six permutations of $\{1, 1, 2, 2\}$ with adjacent interchanges; thus there is no perfect scheme for $(2, 2)$-combinations. In fact, our chances of achieving perfection are only about 1 in 4:

Theorem P. *The generation of all (s, t)-combinations $a_{s+t-1} \ldots a_1 a_0$ by adjacent interchanges $01 \leftrightarrow 10$ is possible if and only if $s \le 1$ or $t \le 1$ or st is odd.*

Proof. Consider all permutations of the multiset $\{s \cdot 0, t \cdot 1\}$. We learned in exercise 5.1.2–16 that the number m_k of such permutations having k inversions is the coefficient of z^k in the z-nomial coefficient

$$\binom{s+t}{t}_z \;=\; \prod_{k=s+1}^{s+t} (1 + z + \cdots + z^{k-1}) \Big/ \prod_{k=1}^{t} (1 + z + \cdots + z^{k-1}). \tag{48}$$

Every adjacent interchange changes the number of inversions by ± 1, so a perfect generation scheme is possible only if approximately half of all the permutations have an odd number of inversions. More precisely, the value of $\binom{s+t}{t}_{-1} = m_0 - m_1 + m_2 - \cdots$ must be 0 or ± 1. But exercise 49 shows that

$$\binom{s+t}{t}_{-1} \;=\; \binom{\lfloor (s+t)/2 \rfloor}{\lfloor t/2 \rfloor} [st \text{ is even}], \tag{49}$$

and this quantity exceeds 1 unless $s \le 1$ or $t \le 1$ or st is odd.

Conversely, perfect schemes are easy with $s \le 1$ or $t \le 1$, and they turn out to be possible also whenever st is odd. The first nontrivial case occurs for $s = t = 3$, when there are four essentially different solutions; the most symmetrical of these is

$$\begin{aligned} &210 — 310 — 410 — 510 — 520 — 521 — 531 — 532 — 432 — 431 — \\ &\quad 421 — 321 — 320 — 420 — 430 — 530 — 540 — 541 — 542 — 543 \end{aligned} \tag{50}$$

(see exercise 51). Several authors have constructed Hamiltonian paths in the relevant graph for arbitrary odd numbers s and t, most notably Eades, Hickey, and Read [*JACM* **31** (1984), 19–29], whose method makes an interesting exercise in programming with recursive coroutines. Unfortunately, however, none of the known constructions are sufficiently simple to describe in a short space, or to implement with reasonable efficiency. Perfect combination generators have therefore not yet proved to be of practical importance. ▮

In summary, then, we have seen that the study of (s,t)-combinations leads to many fascinating patterns, some of which are of great practical importance and some of which are merely elegant and/or beautiful. Fig. 26 illustrates the principal options that are available in the case $s = t = 5$, when $\binom{10}{5} = 252$ combinations arise. Lexicographic order (Algorithm L), the revolving-door Gray code (Algorithm R), the homogeneous scheme K_{55} of (31), and Chase's near-perfect scheme (Algorithm C) are shown in parts (a), (b), (c), and (d) of the illustration. Part (e) shows the near-perfect scheme that is as close to perfection as possible while still being in genlex order (see exercise 34), while part (f) is the perfect scheme of Eades, Hickey, and Read. Finally, Fig. 26(g) is a listing that proceeds by swapping $a_j \leftrightarrow a_0$, akin to Algorithm 7.2.1.2E (see exercise 55).

***Combinations of a multiset.** If multisets can have permutations, they can have combinations too. For example, consider the multiset $\{b,b,b,b,g,g,g,r,r,r,w,w\}$, representing a sack that contains four blue balls and three that are green, three red, two white. There are 37 ways to choose five balls from this sack; in lexicographic order (but descending in each combination) they are

gbbbb, *ggbbb*, *gggbb*, *rbbbb*, *rgbbb*, *rggbb*, *rgggb*, *rrbbb*, *rrgbb*, *rrggb*,
rrggg, *rrrbb*, *rrrgb*, *rrrgg*, *wbbbb*, *wgbbb*, *wggbb*, *wgggb*, *wrbbb*, *wrgbb*,
wrggb, *wrggg*, *wrrbb*, *wrrgb*, *wrrgg*, *wrrrb*, *wrrrg*, *wwbbb*, *wwgbb*, *wwggb*,
wwggg, *wwrbb*, *wwrgb*, *wwrgg*, *wwrrb*, *wwrrg*, *wwrrr*. (51)

This fact might seem frivolous and/or esoteric, yet we will see in Theorem W below that the lexicographic generation of multiset combinations yields optimal solutions to significant combinatorial problems.

James Bernoulli observed in his *Ars Conjectandi* (1713), 119–123, that we can count the number of such combinations by looking at the coefficient of z^5 in $(1+z+z^2)(1+z+z^2+z^3)^2(1+z+z^2+z^3+z^4)$. Indeed, his observation is easy to understand, because we get all possible selections from the sack if we multiply out the polynomials

$$(1+w+ww)(1+r+rr+rrr)(1+g+gg+ggg)(1+b+bb+bbb+bbbb).$$

Multiset combinations are also equivalent to *bounded compositions*, namely to compositions in which the individual parts are bounded. For example, the 37 multicombinations listed in (51) correspond to 37 solutions of

$$5 = r_3 + r_2 + r_1 + r_0, \quad 0 \le r_3 \le 2, \quad 0 \le r_2, r_1 \le 3, \quad 0 \le r_0 \le 4,$$

namely $5 = 0{+}0{+}1{+}4 = 0{+}0{+}2{+}3 = 0{+}0{+}3{+}2 = 0{+}1{+}0{+}4 = \cdots = 2{+}3{+}0{+}0$.

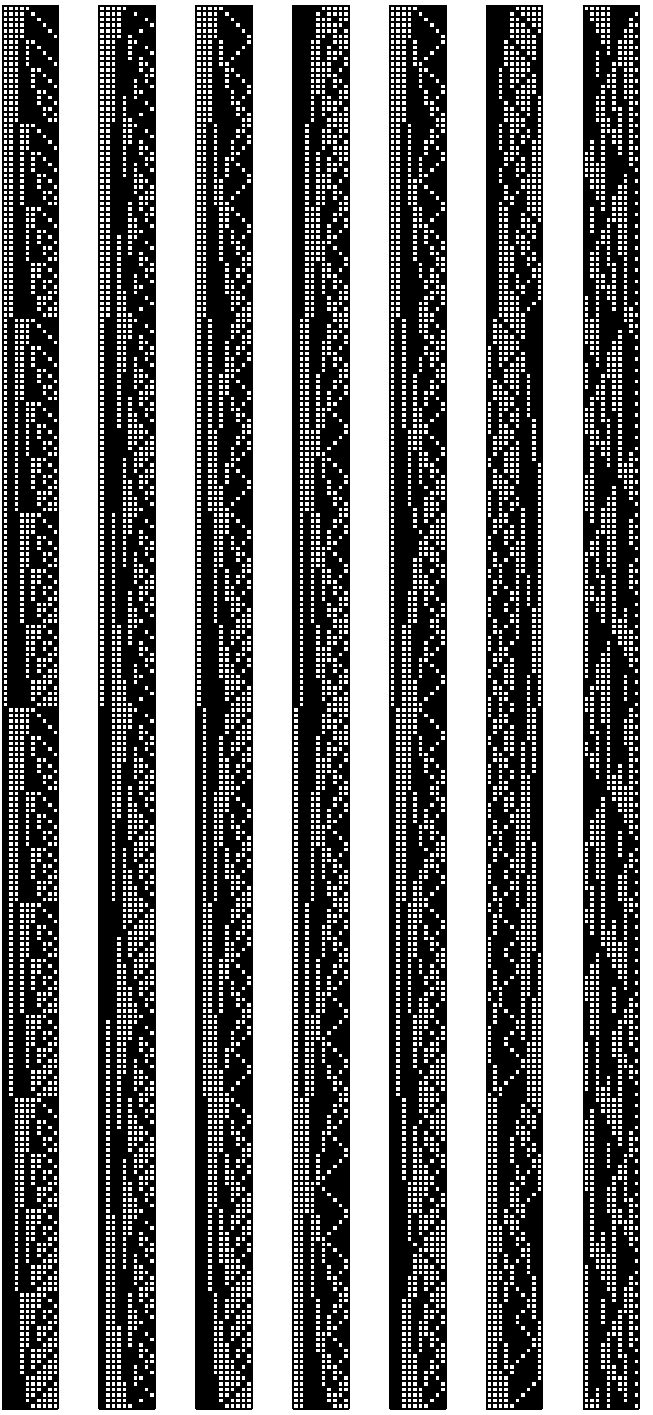


Fig. 26. Examples of $(5,5)$-combinations:
a) lexicographic;
b) revolving-door;
c) homogeneous;
d) near-perfect;
e) nearer-perfect;
f) perfect;
g) right-swapped.

Bounded compositions, in turn, are special cases of *contingency tables*, which are of great importance in statistics. And all of these combinatorial configurations can be generated with Graylike codes as well as in lexicographic order. Exercises 59–62 explore some of the basic ideas involved.

***Shadows.** Sets of combinations appear frequently in mathematics. For example, a set of 2-combinations (namely a set of pairs) is essentially a graph, and a set of t-combinations for general t is called a uniform hypergraph. If the vertices of a convex polyhedron are perturbed slightly, so that no three are collinear, no four lie in a plane, and in general no $t+1$ lie in a $(t-1)$-dimensional hyperplane, the resulting $(t-1)$-dimensional faces are "simplexes" whose vertices have great significance in computer applications. Researchers have learned that such sets of combinations have important properties related to lexicographic generation.

If α is any t-combination $c_t \dots c_2 c_1$, its *shadow* $\partial\alpha$ is the set of all its $(t-1)$-element subsets $c_{t-1} \dots c_2 c_1$, $\dots$, $c_t \dots c_3 c_1$, $c_t \dots c_3 c_2$. For example, $\partial 5310 = \{310, 510, 530, 531\}$. We can also represent a t-combination as a bit string $a_{n-1} \dots a_1 a_0$, in which case $\partial\alpha$ is the set of all strings obtained by changing a 1 to a 0: $\partial 101011 = \{001011, 100011, 101001, 101010\}$. If A is any set of t-combinations, we define its shadow

$$\partial A \;=\; \{\, \partial\alpha \mid \alpha \in A \,\} \tag{52}$$

to be the set of all $(t-1)$-combinations in the shadows of its members. For example, $\partial\partial 5310 = \{10, 30, 31, 50, 51, 53\}$.

These definitions apply also to combinations with repetitions, namely to multicombinations: $\partial 5330 = \{330, 530, 533\}$ and $\partial\partial 5330 = \{30, 33, 50, 53\}$. In general, when A is a set of t-element multisets, ∂A is a set of $(t-1)$-element multisets. Notice, however, that ∂A never has repeated elements itself.

The *upper shadow* $\partial\alpha$ with respect to a universe U is defined similarly, but it goes from t-combinations to $(t+1)$-combinations:

$$\partial\alpha \;=\; \{\, \beta \in U \mid \alpha \in \partial\beta \,\}, \qquad \text{for } \alpha \in U; \tag{53}$$

$$\partial A \;=\; \{\, \partial\alpha \mid \alpha \in A \,\}, \qquad \text{for } A \subseteq U. \tag{54}$$

If, for example, $U = \{0, 1, 2, 3, 4, 5, 6\}$, we have $\partial 5310 = \{53210, 54310, 65310\}$; on the other hand, if $U = \{\infty\cdot 0, \infty\cdot 1, \dots, \infty\cdot 6\}$, we have $\partial 5310 = \{53100, 53110, 53210, 53310, 54310, 55310, 65310\}$.

The following fundamental theorems, which have many applications in various branches of mathematics and computer science, tell us how small a set's shadows can be:

Theorem K. *If A is a set of N t-combinations contained in $U = \{0, 1, \dots, n-1\}$, then*

$$\|\partial A\| \ge \|\partial P_{Nt}\| \qquad \textit{and} \qquad \|\partial A\| \ge \|\partial Q_{Nnt}\|, \tag{55}$$

where P_{Nt} denotes the first N combinations generated by Algorithm L, namely the N lexicographically smallest combinations $c_t \dots c_2 c_1$ that satisfy (3), and Q_{Nnt} denotes the N lexicographically largest. ∎

Theorem M. *If A is a set of N t-multicombinations contained in the multiset $U = \{\infty \cdot 0, \infty \cdot 1, \ldots, \infty \cdot s\}$, then*

$$\|\partial A\| \ge \|\partial \widehat{P}_{Nt}\| \qquad \text{and} \qquad \|\varrho A\| \ge \|\varrho \widehat{Q}_{Nst}\|, \tag{56}$$

where $\widehat{P}_{Nt}$ denotes the N lexicographically smallest multicombinations $d_t \ldots d_2 d_1$ that satisfy (6)*, and $\widehat{Q}_{Nst}$ denotes the N lexicographically largest.* ∎

Both of these theorems are consequences of a stronger result that we shall prove later. Theorem K is generally called the Kruskal–Katona theorem, because it was discovered by J. B. Kruskal [*Math. Optimization Techniques*, edited by R. Bellman (1963), 251–278] and rediscovered by G. Katona [*Theory of Graphs*, Tihany 1966, edited by Erdős and Katona, (Academic Press, 1968), 187–207]; M. P. Schützenberger had previously stated it in a less-well-known publication, with incomplete proof [*RLE Quarterly Progress Report* **55** (1959), 117–118]. Theorem M goes back to F. S. Macaulay, many years earlier [*Proc. London Math. Soc.* (2) **26** (1927), 531–555].

Before proving (55) and (56), let's take a closer look at what those formulas mean. We know from Theorem L that the first N combinations visited by Algorithm L are those that precede $n_t \ldots n_2 n_1$, where

$$N = \binom{n_t}{t} + \cdots + \binom{n_2}{2} + \binom{n_1}{1}, \qquad n_t > \cdots > n_2 > n_1 \ge 0$$

is the degree-t combinatorial representation of N. Sometimes this representation has fewer than t nonzero terms, because n_j can be equal to $j-1$; let's suppress the zeros, and write

$$N = \binom{n_t}{t} + \binom{n_{t-1}}{t-1} + \cdots + \binom{n_v}{v}, \qquad n_t > n_{t-1} > \cdots > n_v \ge v \ge 1. \tag{57}$$

Now the first $\binom{n_t}{t}$ combinations $c_t \ldots c_1$ are the t-combinations of $\{0, \ldots, n_t-1\}$; the next $\binom{n_{t-1}}{t-1}$ are those in which $c_t = n_t$ and $c_{t-1} \ldots c_1$ is a $(t-1)$-combination of $\{0, \ldots, n_{t-1}-1\}$; and so on. For example, if $t = 5$ and $N = \binom{9}{5} + \binom{7}{4} + \binom{4}{3}$, the first N combinations are

$$P_{N5} = \{43210, \ldots, 87654\} \cup \{93210, \ldots, 96543\} \cup \{97210, \ldots, 97321\}. \tag{58}$$

The shadow of this set P_{N5} is, fortunately, easy to understand: It is

$$\partial P_{N5} = \{3210, \ldots, 8765\} \cup \{9210, \ldots, 9654\} \cup \{9710, \ldots, 9732\}, \tag{59}$$

namely the first $\binom{9}{4} + \binom{7}{3} + \binom{4}{2}$ combinations in lexicographic order when $t = 4$.

In other words, if we define Kruskal's function κ_t by the formula

$$\kappa_t N = \binom{n_t}{t-1} + \binom{n_{t-1}}{t-2} + \cdots + \binom{n_v}{v-1} \tag{60}$$

when N has the unique representation (57), we have

$$\partial P_{Nt} \;=\; P_{(\kappa_t N)(t-1)}. \tag{61}$$

Theorem K tells us, for example, that a graph with a million edges can contain at most

$$\binom{1414}{3}+\binom{1009}{2} = 470{,}700{,}300$$

triangles, that is, at most 470,700,300 sets of vertices $\{u,v,w\}$ with u — v — w — u. The reason is that $1000000=\binom{1414}{2}+\binom{1009}{1}$ by exercise 17, and the edges $P_{(1000000)_2}$ do support $\binom{1414}{3}+\binom{1009}{2}$ triangles; but if there were more, the graph would necessarily have at least $\kappa_3 470700301 = \binom{1414}{2}+\binom{1009}{1}+\binom{1}{0} = 1000001$ edges in their shadow.

Kruskal defined the companion function

$$\lambda_t N = \binom{n_t}{t+1}+\binom{n_{t-1}}{t}+\cdots+\binom{n_v}{v+1} \tag{62}$$

to deal with questions such as this. The κ and λ functions are related by an interesting law proved in exercise 71:

$$M+N=\binom{s+t}{t} \quad\text{implies}\quad \kappa_s M+\lambda_t N = \binom{s+t}{t+1}, \quad \text{if } st>0. \tag{63}$$

Turning to Theorem M, the sizes of $\partial\widehat{P}_{Nt}$ and $\varrho\widehat{Q}_{Nst}$ turn out to be

$$\|\partial\widehat{P}_{Nt}\| = \mu_t N \qquad\text{and}\qquad \|\varrho\widehat{Q}_{Nst}\| = N+\kappa_s N \tag{64}$$

(see exercise 80), where the function μ_t satisfies

$$\mu_t N = \binom{n_t-1}{t-1}+\binom{n_{t-1}-1}{t-2}+\cdots+\binom{n_v-1}{v-1} \tag{65}$$

when N has the combinatorial representation (57).

Table 3 shows how these functions $\kappa_t N$, $\lambda_t N$, and $\mu_t N$ behave for small values of t and N. When t and N are large, they can be well approximated in terms of a remarkable function $\tau(x)$ introduced by Teiji Takagi in 1903; see Fig. 27 and exercises 81–84.

Theorems K and M are corollaries of a much more general theorem of discrete geometry, discovered by Da-Lun Wang and Ping Wang [*SIAM J. Applied Math.* **33** (1977), 55–59], which we shall now proceed to investigate. Consider the *discrete n-dimensional torus* $T(m_1,\ldots,m_n)$ whose elements are integer vectors $x=(x_1,\ldots,x_n)$ with $0\le x_1<m_1$, …, $0\le x_n<m_n$. We define the sum and difference of two such vectors x and y as in Eqs. 4.3.2–(2) and 4.3.2–(3):

$$x+y = \big((x_1+y_1) \bmod m_1,\ldots,(x_n+y_n) \bmod m_n\big), \tag{66}$$
$$x-y = \big((x_1-y_1) \bmod m_1,\ldots,(x_n-y_n) \bmod m_n\big). \tag{67}$$

We also define the so-called *cross order* on such vectors by saying that $x\preceq y$ if and only if

$$\nu x<\nu y \quad\text{or}\quad (\nu x=\nu y \text{ and } x\ge y \text{ lexicographically}); \tag{68}$$

here, as usual, $\nu(x_1,\ldots,x_n)=x_1+\cdots+x_n$. For example, when $m_1=m_2=2$ and $m_3=3$, the 12 vectors $x_1x_2x_3$ in cross order are

$$000,\ 100,\ 010,\ 001,\ 110,\ 101,\ 011,\ 002,\ 111,\ 102,\ 012,\ 112, \tag{69}$$

Table 3

EXAMPLES OF THE KRUSKAL–MACAULAY FUNCTIONS κ, λ, AND μ

$N=$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
$\kappa_1 N=$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\kappa_2 N=$	0	2	3	3	4	4	4	5	5	5	5	6	6	6	6	6	7	7	7	7	7
$\kappa_3 N=$	0	3	5	6	6	8	9	9	10	10	10	12	13	13	14	14	14	15	15	15	15
$\kappa_4 N=$	0	4	7	9	10	10	13	15	16	16	18	19	19	20	20	20	23	25	26	26	28
$\kappa_5 N=$	0	5	9	12	14	15	15	19	22	24	25	25	28	30	31	31	33	34	34	35	35
$\lambda_1 N=$	0	0	1	3	6	10	15	21	28	36	45	55	66	78	91	105	120	136	153	171	190
$\lambda_2 N=$	0	0	0	1	1	2	4	4	5	7	10	10	11	13	16	20	20	21	23	26	30
$\lambda_3 N=$	0	0	0	0	1	1	1	2	2	3	5	5	5	6	6	7	9	9	10	12	15
$\lambda_4 N=$	0	0	0	0	0	1	1	1	1	2	2	2	3	3	4	6	6	6	6	7	7
$\lambda_5 N=$	0	0	0	0	0	0	1	1	1	1	1	2	2	2	2	3	3	3	4	4	5
$\mu_1 N=$	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$\mu_2 N=$	0	1	2	2	3	3	3	4	4	4	4	5	5	5	5	5	6	6	6	6	6
$\mu_3 N=$	0	1	2	3	3	4	5	5	6	6	6	7	8	8	9	9	9	10	10	10	10
$\mu_4 N=$	0	1	2	3	4	4	5	6	7	7	8	9	9	10	10	10	11	12	13	13	14
$\mu_5 N=$	0	1	2	3	4	5	5	6	7	8	9	9	10	11	12	12	13	14	14	15	15

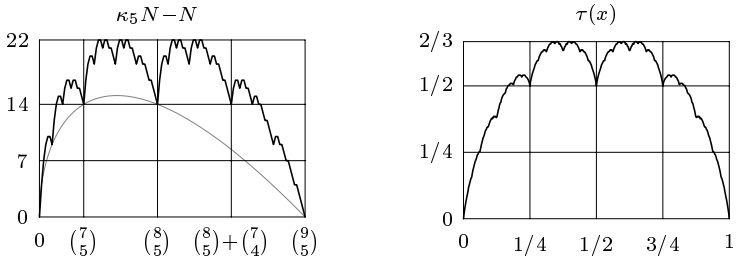


Fig. 27. Approximating a Kruskal function with the Takagi function. (The smooth curve in the left-hand graph is the lower bound $\underline{\kappa}_5 N - N$ of exercise 79.)

omitting parentheses and commas for convenience. The *complement* of a vector in $T(m_1, \ldots, m_n)$ is

$$\overline{x} \;=\; (m_1 - 1 - x_1, \ldots, m_n - 1 - x_n). \tag{70}$$

Notice that $x \preceq y$ holds if and only if $\overline{x} \succeq \overline{y}$. Therefore we have

$$\operatorname{rank}(x) + \operatorname{rank}(\overline{x}) \;=\; T - 1, \qquad \text{where } T = m_1 \ldots m_n, \tag{71}$$

if $\operatorname{rank}(x)$ denotes the number of vectors that precede x in cross order.

We will find it convenient to call the vectors "points" and to name the points e_0, e_1, $\ldots$, e_{T-1} in increasing cross order. Thus we have $e_7 = 002$ in (69), and $\overline{e}_r = e_{T-1-r}$ in general. Notice that

$$e_1 = 100\ldots00, \quad e_2 = 010\ldots00, \quad \ldots, \quad e_n = 000\ldots01; \tag{72}$$

these are the so-called *unit vectors*. The set

$$S_N = \{e_0, e_1, \ldots, e_{N-1}\} \tag{73}$$

consisting of the smallest N points is called a *standard set*, and in the special case $N = n + 1$ we write

$$E = \{e_0, e_1, \ldots, e_n\} = \{000\ldots00, 100\ldots00, 010\ldots00, \ldots, 000\ldots01\}. \tag{74}$$

Any set of points X has a *spread* X^+, a *core* X°, and a *dual* $X^\sim$, defined by the rules

$$X^+ = \{\, x \in S_T \mid x \in X \text{ or } x - e_1 \in X \text{ or } \cdots \text{ or } x - e_n \in X \,\}; \tag{75}$$

$$X^\circ = \{\, x \in S_T \mid x \in X \text{ and } x + e_1 \in X \text{ and } \cdots \text{ and } x + e_n \in X \,\}; \tag{76}$$

$$X^\sim = \{\, x \in S_T \mid \overline{x} \notin X \,\}. \tag{77}$$

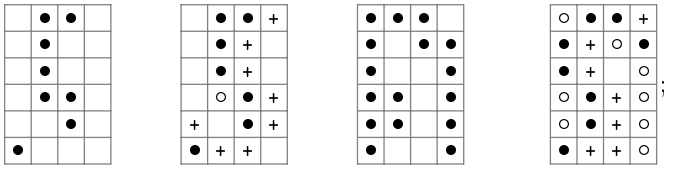
We can also define the spread of X algebraically, writing

$$X^+ = X + E, \tag{78}$$

where $X + Y$ denotes $\{\, x + y \mid x \in X \text{ and } y \in Y \,\}$. Clearly

$$X^+ \subseteq Y \qquad \text{if and only if} \qquad X \subseteq Y^\circ. \tag{79}$$

These notions can be illustrated in the two-dimensional case $m_1 = 4$, $m_2 = 6$, by the more-or-less random toroidal arrangements



X $\quad$ X° and X^+ $\quad$ $X^\sim$ $\quad$ $X^{\sim\circ}$ and $X^{\sim+}$; (80)

here X consists of points marked $\bullet$ or $\circ$, X° comprises just the $\circ$s, and X^+ consists of $+$s plus $\bullet$s plus $\circ$s. Notice that if we rotate the diagram for $X^{\sim\circ}$ and $X^{\sim+}$ by 180°, we obtain the diagram for X° and X^+, but with $(\bullet, \circ, +, \ )$ respectively changed to $(+, \ , \bullet, \circ)$; and in fact the identities

$$X^\circ = X^{\sim+\sim}, \qquad X^+ = X^{\sim\circ\sim} \tag{81}$$

hold in general (see exercise 85).

Now we are ready to state the theorem of Wang and Wang:

Theorem W. *Let X be any set of N points in the discrete torus $T(m_1, \ldots, m_n)$, where $m_1 \le \cdots \le m_n$. Then $\|X^+\| \ge \|S_N^+\|$ and $\|X^\circ\| \le \|S_N^\circ\|$.*

In other words, the standard sets S_N have the smallest spread and largest core, among all N-point sets. We will prove this result by following a general approach first used by F. W. J. Whipple to prove Theorem M [*Proc. London Math. Soc.* (2) **28** (1928), 431–437]. The first step is to prove that the spread and the core of standard sets are standard:

Lemma S. *There are functions α and β such that $S_N^+ = S_{\alpha N}$ and $S_N^\circ = S_{\beta N}$.*

Proof. We may assume that $N > 0$. Let r be maximum with $e_r \in S_N^+$, and let $\alpha N = r + 1$; we must prove that $e_q \in S_N^+$ for $0 \le q < r$. Suppose $e_q = x = (x_1, \ldots, x_n)$ and $e_r = y = (y_1, \ldots, y_n)$, and let k be the largest subscript with $x_k > 0$. Since $y \in S_N^+$, there is a subscript j such that $y - e_j \in S_N$. It suffices to prove that $x - e_k \preceq y - e_j$, and exercise 87 does this.

The second part follows from (81), with $\beta N = T - \alpha(T - N)$, because $S_N^\sim = S_{T-N}$. ∎

Theorem W is obviously true when $n = 1$, so we assume by induction that it has been proved in $n - 1$ dimensions. The next step is to *compress* the given set X in the kth coordinate position, by partitioning it into disjoint sets

$$X_k(a) \;=\; \{\, x \in X \mid x_k = a \,\} \tag{82}$$

for $0 \le a < m_k$ and replacing each $X_k(a)$ by

$$X'_k(a) \;=\; \{\, (s_1, \ldots, s_{k-1}, a, s_k, \ldots, s_{n-1}) \mid (s_1, \ldots, s_{n-1}) \in S_{\|X_k(a)\|} \,\}, \tag{83}$$

a set with the same number of elements. The sets S used in (83) are standard in the $(n-1)$-dimensional torus $T(m_1, \ldots, m_{k-1}, m_{k+1}, \ldots, m_n)$. Notice that we have $(x_1, \ldots, x_{k-1}, a, x_{k+1}, \ldots, x_n) \preceq (y_1, \ldots, y_{k-1}, a, y_{k+1}, \ldots, y_n)$ if and only if $(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n) \preceq (y_1, \ldots, y_{k-1}, y_{k+1}, \ldots, y_n)$; therefore $X'_k(a) = X_k(a)$ if and only if the $(n-1)$-dimensional points $(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n)$ with $(x_1, \ldots, x_{k-1}, a, x_{k+1}, \ldots, x_n) \in X$ are as small as possible when projected onto the $(n-1)$-dimensional torus. We let

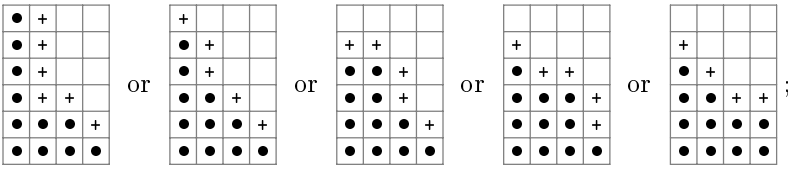
$$C_k X \;=\; X'_k(0) \cup X'_k(1) \cup \cdots \cup X'_k(m_k - 1) \tag{84}$$

be the compression of X in position k. Exercise 89 proves the basic fact that compression does not increase the size of the spread:

$$\|X^+\| \;\ge\; \|(C_k\, X)^+\|, \qquad \text{for } 1 \le k \le n. \tag{85}$$

Furthermore, if compression changes X, it replaces some of the elements by other elements of lower rank. Therefore we need only prove Theorem W for sets X that are totally compressed, having $X = C_k\, X$ for all k.

Consider, for example, the case $n = 2$. A totally compressed set in two dimensions has all points moved to the left of their rows and the bottom of their columns, as in



the rightmost of these is standard, and has the smallest spread. Exercise 90 completes the proof of Theorem W in two dimensions.

When $n > 2$, suppose $x = (x_1, \ldots, x_n) \in X$ and $x_j > 0$. The condition $C_k X = X$ implies that, if $0 \le i < j$ and $i \ne k \ne j$, we have $x + e_i - e_j \in X$. Applying this fact for three values of k tells us that $x + e_i - e_j \in X$ whenever $0 \le i < j$. Consequently

$$X_n(a) + E_n(0) \;\subseteq\; X_n(a-1) + e_n \quad \text{for } 0 < a < m, \tag{86}$$

where $m = m_n$ and $E_n(0)$ is a clever abbreviation for the set $\{e_0, \ldots, e_{n-1}\}$.

Let $X_n(a)$ have N_a elements, so that $N = \|X\| = N_0 + N_1 + \cdots + N_{m-1}$, and let $Y = X^+$. Then

$$Y_n(a) \;=\; \big(X_n\big((a-1) \bmod m\big) + e_n\big) \cup \big(X_n(a) + E_n(0)\big)$$

is standard in $n-1$ dimensions, and (86) tells us that

$$N_{m-1} \le \beta N_{m-2} \le N_{m-2} \le \cdots \le N_1 \le \beta N_0 \le N_0 \le \alpha N_0,$$

where α and β refer to coordinates 1 through $n-1$. Therefore

$$\begin{aligned} \|Y\| &= \|Y_n(0)\| + \|Y_n(1)\| + \|Y_n(2)\| + \cdots + \|Y_n(m-1)\| \\ &= \alpha N_0 + N_0 + N_1 + \cdots + N_{m-2} = \alpha N_0 + N - N_{m-1}. \end{aligned}$$

The proof of Theorem W now has a beautiful conclusion. Let $Z = S_N$, and suppose $\|Z_n(a)\| = M_a$. We want to prove that $\|Y\| \ge \|Z^+\|$, namely that

$$\alpha N_0 + N - N_{m-1} \;\ge\; \alpha M_0 + N - M_{m-1}, \tag{87}$$

because the arguments of the previous paragraph apply to Z as well as to X. We will prove (87) by showing that $N_{m-1} \le M_{m-1}$ and $N_0 \ge M_0$.

Using the $(n-1)$-dimensional α and β functions, let us define

$$N'_{m-1} = N_{m-1},\; N'_{m-2} = \alpha N'_{m-1},\; \ldots,\; N'_1 = \alpha N'_2,\; N'_0 = \alpha N'_1; \tag{88}$$

$$N''_0 = N_0,\; N''_1 = \beta N''_0,\; N''_2 = \beta N''_1,\; \ldots,\; N''_{m-1} = \beta N''_{m-2}. \tag{89}$$

Then we have $N'_a \le N_a \le N''_a$ for $0 \le a < m$, and it follows that

$$N' = N'_0 + N'_1 + \cdots + N'_{m-1} \;\le\; N \;\le\; N'' = N''_0 + N''_1 + \cdots + N''_{m-1}. \tag{89}$$

Exercise 91 proves that the standard set $Z'' = S_{N''}$ has exactly N''_a elements with nth coordinate equal to a, for each a; and by the duality between α and β, the standard set $Z' = S_{N'}$ likewise has exactly N'_a elements with nth coordinate a. Finally, therefore,

$$\begin{aligned} M_{m-1} = \|Z_n(m-1)\| &\ge \|Z'_n(m-1)\| = N_{m-1}, \\ M_0 = \|Z_n(0)\| &\le \|Z''_n(0)\| = N_0, \end{aligned}$$

because $Z' \subseteq Z \subseteq Z''$ by (89). ∎

Now we are ready to prove the claimed results about shadows, which are in fact special cases of a more general theorem of Clements and Lindström that applies to arbitrary multisets [*J. Combinatorial Theory* **7** (1969), 230–238]:

Corollary C. *If A is a set of N t-multicombinations contained in the multiset $U = \{s_0 \cdot 0, s_1 \cdot 1, \ldots, s_d \cdot d\}$, where $s_0 \geq s_1 \geq \cdots \geq s_d$, then*

$$\|\partial A\| \geq \|\partial P_{Nt}\| \qquad \text{and} \qquad \|\varrho A\| \geq \|\varrho Q_{Nt}\|, \tag{90}$$

where P_{Nt} denotes the N lexicographically smallest multicombinations $c_t \ldots c_2 c_1$ of U, and Q_{Nt} denotes the N lexicographically largest.

Proof. Multicombinations of U can be represented as points $x_1 \ldots x_n$ of the torus $T(m_1, \ldots, m_n)$, where $n = d + 1$ and $m_j = s_{n-j} + 1$; we let x_j be the number of occurrences of $n - j$. This correspondence preserves lexicographic order. For example, if $U = \{0, 0, 0, 1, 1, 2, 3\}$, its 3-multicombinations are

$$000,\ 100,\ 110,\ 200,\ 210,\ 211,\ 300,\ 310,\ 311,\ 321, \tag{91}$$

in lexicographic order, and the corresponding points $x_1 x_2 x_3 x_4$ are

$$0003, 0012, 0021, 0102, 0111, 0120, 1002, 1011, 1020, 1110. \tag{92}$$

Let T_w be the points of the torus that have weight $x_1 + \cdots + x_n = w$. Then every allowable set A of t-multicombinations is a subset of T_t. Furthermore—and this is the main point—the spread of $T_0 \cup T_1 \cup \cdots \cup T_{t-1} \cup A$ is

$$\begin{aligned} (T_0 \cup T_1 \cup \cdots \cup T_{t-1} \cup A)^+ &= T_0^+ \cup T_1^+ \cup \cdots \cup T_{t-1}^+ \cup A^+ \\ &= T_0 \cup T_1 \cup \cdots \cup T_t \cup \varrho A. \end{aligned} \tag{93}$$

Thus the upper shadow ϱA is simply $(T_0 \cup T_1 \cup \cdots \cup T_{t-1} \cup A)^+ \cap T_{t+1}$, and Theorem W tells us that $\|A\| = N$ implies $\|\varrho A\| \geq \|\varrho(S_{M+N} \cap T_t)\|$, where $M = \|T_0 \cup \cdots \cup T_{t-1}\|$. Hence, by the definition of cross order, $S_{M+N} \cup T_t$ consists of the lexicographically largest N t-multicombinations, namely Q_{Nt}.

The proof that $\|\partial A\| \geq \|\partial P_{Nt}\|$ follows by complementation (see exercise 93). ∎

EXERCISES

1. [*M23*] Explain why Golomb's rule (8) makes all sets $\{c_1, \ldots, c_t\} \subseteq \{0, \ldots, n-1\}$ correspond uniquely to multisets $\{e_1, \ldots, e_t\} \subseteq \{\infty \cdot 0, \ldots, \infty \cdot n - t\}$.

2. [*16*] What path in an 11×13 grid corresponds to the bit string (113)?

▸ **3.** [*21*] (R. R. Fenichel, 1968.) Show that the compositions $q_t + \cdots + q_1 + q_0$ of s into $t + 1$ nonnegative parts can be generated in lexicographic order by a simple loopless algorithm.

4. [*16*] Show that every composition $q_t \ldots q_0$ of s into $t+1$ nonnegative parts corresponds to a composition $r_s \ldots r_0$ of t into $s+1$ nonnegative parts. What composition corresponds to 10224000001010 under this correspondence?

▸ **5.** [*20*] What is a good way to generate all of the integer solutions to the following systems of inequalities?

a) $n > x_t \geq x_{t-1} > x_{t-2} \geq x_{t-3} > \cdots > x_1 \geq 0$, when t is odd.

b) $n \gg x_t \gg x_{t-1} \gg \cdots \gg x_2 \gg x_1 \gg 0$, where $a \gg b$ means $a \geq b + 2$.

6. [*M22*] How often is each step of Algorithm T performed?

7. [*22*] Design an algorithm that runs through the "dual" combinations $b_s \dots b_2 b_1$ in *decreasing* lexicographic order (see (5) and Table 1). Like Algorithm T, your algorithm should avoid redundant assignments and unnecessary searching.

8. [*M23*] Design an algorithm that generates all (s,t)-combinations $a_{n-1} \dots a_1 a_0$ lexicographically in bitstring form. The total running time should be $O\binom{n}{t}$, assuming that $st > 0$.

9. [*M26*] When all (s,t)-combinations $a_{n-1} \dots a_1 a_0$ are listed in lexicographic order, let $2A_{st}$ be the total number of bit changes between adjacent strings. For example, $A_{33} = 25$ because there are respectively

$$2+2+2+4+2+2+4+2+2+6+2+2+4+2+2+4+2+2+2 = 50$$

bit changes between the 20 strings in Table 1.

a) Show that $A_{st} = \min(s,t) + A_{(s-1)t} + A_{s(t-1)}$ when $st > 0$; $A_{st} = 0$ when $st = 0$.

b) Prove that $A_{st} < 2\binom{s+t}{t}$.

▸ **10.** [*21*] The "World Series" of baseball is traditionally a competition in which the American League champion (A) plays the National League champion (N) until one of them has beaten the other four times. What is a good way to list all possible scenarios AAAA, AAANA, AAANNA, ..., NNNN? What is a simple way to assign consecutive integers to those scenarios?

11. [*19*] Which of the scenarios in exercise 10 occurred most often during the 1900s? Which of them never occurred? [*Hint:* World Series scores are easily found on the Internet.]

12. [*HM32*] A set V of n-bit vectors that is closed under addition modulo 2 is called a *binary vector space*.

a) Prove that every such V contains 2^t elements, for some integer t, and can be represented as the set $\{x_1\alpha_1 \oplus \cdots \oplus x_t\alpha_t \mid 0 \le x_1, \dots, x_t \le 1\}$ where the vectors $\alpha_1, \dots, \alpha_t$ form a "canonical basis" with the following property: There is a t-combination $c_t \dots c_2 c_1$ of $\{0, 1, \dots, n-1\}$ such that, if α_k is the binary vector $a_{k(n-1)} \dots a_{k1} a_{k0}$, we have

$$a_{kc_j} = [j = k] \quad \text{for } 1 \le j,k \le t; \qquad\qquad a_{kl} = 0 \quad \text{for } 0 \le l < c_k,\ 1 \le k \le t.$$

For example, the canonical bases with $n = 9$, $t = 4$, and $c_4c_3c_2c_1 = 7641$ have the general form

$$\begin{aligned} \alpha_1 &= *\,0\,0\,*\,0\,*\,*\,1\,0, \\ \alpha_2 &= *\,0\,0\,*\,1\,0\,0\,0\,0, \\ \alpha_3 &= *\,0\,1\,0\,0\,0\,0\,0\,0, \\ \alpha_4 &= *\,1\,0\,0\,0\,0\,0\,0\,0; \end{aligned}$$

there are 2^8 ways to replace the eight asterisks by 0s and/or 1s, and each of these defines a canonical basis. We call t the dimension of V.

b) How many t-dimensional spaces are possible with n-bit vectors?

c) Design an algorithm to generate all canonical bases $(\alpha_1, \dots, \alpha_t)$ of dimension t. *Hint:* Let the associated combinations $c_t \dots c_1$ increase lexicographically as in Algorithm L.

d) What is the 1000000th basis visited by your algorithm when $n = 9$ and $t = 4$?

13. [*25*] A one-dimensional *Ising configuration* of length n, weight t, and energy r, is a binary string $a_{n-1} \dots a_0$ such that $\sum_{j=0}^{n-1} a_j = t$ and $\sum_{j=1}^{n-1} b_j = r$, where $b_j =$

$a_j \oplus a_{j-1}$. For example, $a_{12} \ldots a_0 = 1100100100011$ has weight 6 and energy 6, since $b_{12} \ldots b_1 = 010110110010$.

Design an algorithm to generate all such configurations, given n, t, and r.

14. [*26*] When the binary strings $a_{n-1} \ldots a_1 a_0$ of (s,t)-combinations are generated in lexicographic order, we sometimes need to change $2\min(s,t)$ bits to get from one combination to the next. For example, 011100 is followed by 100011 in Table 1. Therefore we apparently cannot hope to generate all combinations with a loopless algorithm unless we visit them in some other order.

Show, however, that there actually is a way to compute the lexicographic successor of a given combination in $O(1)$ steps, if each combination is represented indirectly in a doubly linked list as follows: There are arrays $l[0], \ldots, l[n]$ and $r[0], \ldots, r[n]$ such that $l[r[j]] = j$ for $0 \le j \le n$. If $x_0 = l[0]$ and $x_j = l[x_{j-1}]$ for $0 < j < n$, then $a_j = [x_j > s]$ for $0 \le j < n$.

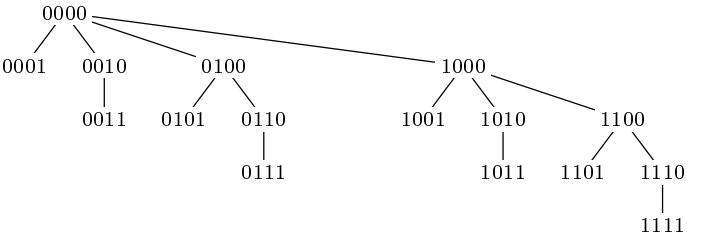
15. [*M22*] Use the fact that dual combinations $b_s \ldots b_2 b_1$ occur in reverse lexicographic order to prove that the sum $\binom{b_s}{s} + \cdots + \binom{b_2}{2} + \binom{b_1}{1}$ has a simple relation to the sum $\binom{c_t}{t} + \cdots + \binom{c_2}{2} + \binom{c_1}{1}$.

16. [*M21*] What is the millionth combination generated by Algorithm L when t is (a) 2? (b) 3? (c) 4? (d) 5? (e) 1000000?

17. [*HM25*] Given N and t, what is a good way to compute the combinatorial representation (20)?

▸ **18.** [*20*] What binary tree do we get when the binomial tree T_n is represented by "right child" and "left sibling" pointers as in exercise 2.3.2–5?

19. [*21*] Instead of labeling the branches of the binomial tree T_4 as shown in (22), we could label each node with the bit string of its corresponding combination:



If T_∞ has been labeled in this way, suppressing leading zeros, preorder is the same as the ordinary increasing order of binary notation; so the millionth node turns out to be 11111010000010001111111. But what is the millionth node of T_∞ in *postorder*?

20. [*M20*] Find generating functions g and h such that Algorithm F finds exactly $[z^N]\, g(z)$ feasible combinations and sets $t \leftarrow t+1$ exactly $[z^N]\, h(z)$ times.

21. [*M22*] Prove the alternating combination law (30).

22. [*M23*] What is the millionth revolving-door combination visited by Algorithm R when t is (a) 2? (b) 3? (c) 4? (d) 5? (e) 1000000?

23. [*M23*] Suppose we augment Algorithm R by setting $j \leftarrow t+1$ in step R1, and $j \leftarrow 1$ if R3 goes directly to R2. Find the probability distribution of j, and its average value. What does this imply about the running time of the algorithm?

▸ **24.** [*M25*] (W. H. Payne, 1974.) Continuing the previous exercise, let j_k be the value of j on the kth visit by Algorithm R. Show that $|j_{k+1} - j_k| \le 2$, and explain how to make the algorithm loopless by exploiting this property.

25. [*M35*] Let $c_t \dots c_2 c_1$ and $c'_t \dots c'_2 c'_1$ be the Nth and N'th combinations generated by the revolving-door method, Algorithm R. If the set $C = \{c_t, \dots, c_2, c_1\}$ has m elements not in $C' = \{c'_t, \dots, c'_2, c'_1\}$, prove that $|N - N'| > \sum_{k=1}^{m-1} \binom{2k}{k-1}$.

26. [*26*] Do elements of the *ternary* reflected Gray path have properties similar to the revolving-door Gray code Γ_{st}, if we extract only the n-tuples $a_{n-1} \dots a_1 a_0$ such that (a) $a_{n-1} + \cdots + a_1 + a_0 = t$? (b) $\{a_{n-1}, \dots, a_1, a_0\} = \{r \cdot 0, s \cdot 1, t \cdot 2\}$?

▸ **27.** [*25*] Show that there is a simple way to generate all combinations of *at most* t elements of $\{0, 1, \dots, n-1\}$, using only Gray-code-like transitions $0 \leftrightarrow 1$ and $01 \leftrightarrow 10$. (In other words, each step should either insert a new element, delete an element, or shift an element by ± 1.) For example,

0000, 0001, 0011, 0010, 0110, 0101, 0100, 1100, 1010, 1001, 1000

is one such sequence when $n = 4$ and $t = 2$. *Hint:* Think of Chinese rings.

28. [*M21*] True or false: A listing of (s,t)-combinations $a_{n-1} \dots a_1 a_0$ in bitstring form is in genlex order if and only if the corresponding index-form listings $b_s \dots b_2 b_1$ (for the 0s) and $c_t \dots c_2 c_1$ (for the 1s) are both in genlex order.

▸ **29.** [*M28*] (P. J. Chase.) Given a string on the symbols `+`, `-`, and `0`, say that an *R-block* is a substring of the form $\texttt{-}^{k+1}$ that is preceded by `0` and not followed by `-`; an *L-block* is a substring of the form $\texttt{+-}^{k}$ that is followed by `0`; in both cases $k \ge 0$. For example, the string `+00++-++-000-` has two L-blocks and one R-block, shown in gray. Notice that blocks cannot overlap.

We form the *successor* of such a string as follows, whenever at least one block is present: Replace the rightmost $\texttt{0-}^{k+1}$ by $\texttt{-+}^{k}\texttt{0}$, if the rightmost block is an R-block; otherwise replace the rightmost $\texttt{+-}^{k}\texttt{0}$ by $\texttt{0+}^{k+1}$. Also negate the first sign, if any, that appears to the right of the block that has been changed. For example,

$$\texttt{-+00++-} \to \texttt{-0+0-+-} \to \texttt{-0+--0--} \to \texttt{-0+--+0} \to \texttt{-0+--0+} \to \texttt{-00+++-},$$

where the notation $\alpha \to \beta$ means that β is the successor of α.

a) What strings have no blocks (and therefore no successor)?
b) Can there be a cycle of strings with $\alpha_0 \to \alpha_1 \to \cdots \to \alpha_{k-1} \to \alpha_0$?
c) Prove that if $\alpha \to \beta$ then $-\beta \to -\alpha$, where "$-$" means "negate all the signs." (Therefore every string has at most one predecessor.)
d) Show that if $\alpha_0 \to \alpha_1 \to \cdots \to \alpha_k$ and $k > 0$, the strings α_0 and α_k do not have all their `0`s in the same positions. (Therefore, if α_0 has s signs and t zeros, k must be less than $\binom{s+t}{t}$.)
e) Prove that every string α with s signs and t zeros belongs to exactly one chain $\alpha_0 \to \alpha_1 \to \cdots \to \alpha_{\binom{s+t}{t}-1}$.

30. [*M32*] The previous exercise defines 2^s ways to generate all combinations of s 0s and t 1s, via the mapping `+` $\mapsto$ 0, `-` $\mapsto$ 0, and `0` $\mapsto$ 1. Show that each of these ways is a homogeneous genlex sequence, definable by an appropriate recurrence. Is Chase's sequence (37) a special case of this general construction?

31. [*M23*] How many genlex listings of (s,t)-combinations are possible in (a) bitstring form $a_{n-1} \dots a_1 a_0$? (b) index-list form $c_t \dots c_2 c_1$?

▸ **32.** [*M30*] How many of the genlex listings of (s,t)-combination strings $a_{n-1}\ldots a_1a_0$ (a) have the revolving-door property? (b) are homogeneous?

33. [*HM31*] How many of the genlex listings in exercise 31(b) are near-perfect?

34. [*M30*] Continuing exercise 33, explain how to find such schemes that are as near as possible to perfection, in the sense that the number of "imperfect" transitions $c_j \leftarrow c_j \pm 2$ is minimized, when s and t are not too large.

35. [*M26*] How many steps of Chase's sequence C_{st} use an imperfect transition?

▸ **36.** [*M21*] Prove that method (39) performs the operation $j \leftarrow j+1$ a total of exactly $\binom{s+t}{t}-1$ times as it generates all (s,t)-combinations $a_{n-1}\ldots a_1a_0$, given any genlex scheme for combinations in bitstring form.

▸ **37.** [*27*] What algorithm results when the general genlex method (39) is used to produce (s,t)-combinations $a_{n-1}\ldots a_1a_0$ in (a) lexicographic order? (b) the revolving-door order of Algorithm R? (c) the homogeneous order of (31)?

38. [*26*] Design a genlex algorithm like Algorithm C for the *reverse* sequence C^R_{st}.

39. [*M21*] When $s=12$ and $t=14$, how many combinations precede the bit string 11001001000011111101101010 in Chase's sequence C_{st}? (See (41).)

40. [*M22*] What is the millionth combination in Chase's sequence C_{st}, when $s=12$ and $t=14$?

41. [*M27*] Show that there is a permutation $c(0)$, $c(1)$, $c(2)$, ... of the nonnegative integers such that the elements of Chase's sequence C_{st} are obtained by complementing the least significant $s+t$ bits of the elements $c(k)$ for $0\le k<2^{s+t}$ that have weight $\nu(c(k))=s$. (Thus the sequence $\bar c(0)$, ..., $\bar c(2^n-1)$ contains, as subsequences, all of the C_{st} for which $s+t=n$, just as Gray binary code $g(0)$, ..., $g(2^n-1)$ contains all the revolving-door sequences Γ_{st}.) Explain how to compute the binary representation $c(k)=(\ldots a_2a_1a_0)_2$ from the binary representation $k=(\ldots b_2b_1b_0)_2$.

42. [*HM34*] Use generating functions of the form $\sum_{s,t} g_{st}w^sz^t$ to analyze each step of Algorithm C.

43. [*20*] Prove or disprove: If $s(x)$ and $p(x)$ denote respectively the successor and predecessor of x in endo-order, then $s(x+1)=p(x)+1$.

▸ **44.** [*M21*] Let $C_t(n)-1$ denote the sequence obtained from $C_t(n)$ by striking out all combinations with $c_1=0$, then replacing $c_t\ldots c_1$ by $(c_t-1)\ldots(c_1-1)$ in the combinations that remain. Show that $C_t(n)-1$ is near-perfect.

45. [*32*] Exploit endo-order and the expansions sketched in (44) to generate the combinations $c_t\ldots c_2c_1$ of Chase's sequence $C_t(n)$ with a nonrecursive procedure.

▸ **46.** [*33*] Construct a nonrecursive algorithm for the dual combinations $b_s\ldots b_2b_1$ of Chase's sequence C_{st}, namely the positions of the zeros in $a_{n-1}\ldots a_1a_0$.

47. [*26*] Implement the near-perfect multiset permutation method of (46) and (47).

48. [*M21*] Suppose α_0, α_1, ..., α_{N-1} is any listing of the permutations of the multiset $\{s_1\cdot 1,\ldots,s_d\cdot d\}$, where α_k differs from α_{k+1} by the interchange of two elements. Let β_0, ..., β_{M-1} be any revolving-door listing for (s,t)-combinations, where $s=s_0$, $t=s_1+\cdots+s_d$, and $M=\binom{s+t}{t}$. Then let Λ_j be the list of M elements obtained by starting with $\alpha_j\uparrow\beta_0$ and applying the revolving-door exchanges; here $\alpha\uparrow\beta$ denotes the string obtained by substituting the elements of α for the 1s in β, preserving left-right order. For example, if β_0, ..., β_{M-1} is 0110, 0101, 1100, 1001, 0011, 1010, and if $\alpha_j=12$,

then Λ_j is 0120, 0102, 1200, 1002, 0012, 1020. (The revolving-door listing need *not* be homogeneous.)

Prove that the list (47) contains all permutations of $\{s_0 \cdot 0, s_1 \cdot 1, \ldots, s_d \cdot d\}$, and that adjacent permutations differ from each other by the interchange of two elements.

49. [*HM23*] If q is a primitive mth root of unity, such as $e^{2\pi i/m}$, show that

$$\binom{n}{k}_q = \binom{\lfloor n/m \rfloor}{\lfloor k/m \rfloor} \binom{n \bmod m}{k \bmod m}_q.$$

▸ **50.** [*HM25*] Extend the formula of the previous exercise to *q-multinomial* coefficients

$$\binom{n_1 + \cdots + n_t}{n_1, \ldots, n_t}_q.$$

51. [*25*] Find all Hamiltonian paths in the graph whose vertices are permutations of $\{0, 0, 0, 1, 1, 1\}$ related by adjacent transposition. Which of those paths are equivalent under the operations of interchanging 0s with 1s and/or left-right reflection?

52. [*M37*] Generalizing Theorem P, find a necessary and sufficient condition that all permutations of the multiset $\{s_0 \cdot 0, \ldots, s_d \cdot d\}$ can be generated by adjacent transpositions $a_j a_{j-1} \leftrightarrow a_{j-1} a_j$.

53. [*M46*] (D. H. Lehmer, 1965.) Suppose the N permutations of $\{s_0 \cdot 0, \ldots, s_d \cdot d\}$ cannot be generated by a perfect scheme, because $(N + x)/2$ of them have an even number of inversions, where $x \geq 2$. Is it possible to generate them all with a sequence of $N + x - 2$ adjacent interchanges $a_{\delta_k} \leftrightarrow a_{\delta_k - 1}$ for $1 \leq k < N + x - 1$, where $x - 1$ cases are "spurs" with $\delta_k = \delta_{k-1}$ that take us back to the permutation we've just seen? For example, a suitable sequence $\delta_1 \ldots \delta_{94}$ for the 90 permutations of $\{0, 0, 1, 1, 2, 2\}$, where $x = \binom{2+2+2}{2,2,2}_{-1} = 6$, is $234535432523451\alpha42\alpha^R51\alpha42\alpha^R51\alpha4$, where $\alpha = 45352542345355$, if we start with $a_5 a_4 a_3 a_2 a_1 a_0 = 221100$.

54. [*M40*] For what values of s and t can all (s, t)-combinations be generated if we allow end-around swaps $a_{n-1} \leftrightarrow a_0$ in addition to adjacent interchanges $a_j \leftrightarrow a_{j-1}$?

55. [*M49*] (Buck and Wiedemann, 1984.) Can all (t, t)-combinations $a_{2t-1} \ldots a_1 a_0$ be generated by repeatedly swapping a_0 with some other element?

▸ **56.** [*22*] (Frank Ruskey.) Can a piano player run through all possible 4-note chords that span at most one octave, changing only one finger at a time? This is the problem of generating all combinations $c_t \ldots c_1$ such that $n > c_t > \cdots > c_1 \geq 0$ and $c_t - c_1 < m$, where $t = 4$ and (a) $m = 8$, $n = 52$ if we consider only the white notes of a piano keyboard; (b) $m = 13$, $n = 88$ if we consider also the black notes.

57. [*20*] Consider the piano player's problem of exercise 56 with the additional condition that the chords don't involve adjacent notes. (In other words, $c_{j+1} > c_j + 1$ for $t > j \geq 1$.)

58. [*M25*] Is there a *perfect* solution to the piano player's problem, in which each step moves a finger to an *adjacent* key?

59. [*23*] Design an algorithm to generate all *bounded* compositions

$$t = r_s + \cdots + r_1 + r_0, \qquad \text{where } 0 \leq r_j \leq m_j \text{ for } s \geq j \geq 0.$$

60. [*32*] Show that all bounded compositions can be generated by changing only two of the parts at each step.

▸ **61.** [*M27*] A *contingency table* is an $m \times n$ matrix of nonnegative integers (a_{ij}) having given row sums $r_i = \sum_{j=1}^{n} a_{ij}$ and column sums $c_j = \sum_{i=1}^{m} a_{ij}$, where $r_1 + \cdots + r_m = c_1 + \cdots + c_n$.

a) Show that $2 \times n$ contingency tables are equivalent to bounded compositions.

b) What is the lexicographically largest contingency table for $(r_1, \ldots, r_m; c_1, \ldots, c_n)$, when matrix entries are read row-wise from left to right and top to bottom, namely in the order $(a_{11}, a_{12}, \ldots, a_{1n}, a_{21}, \ldots, a_{mn})$?

c) What is the lexicographically largest contingency table for $(r_1, \ldots, r_m; c_1, \ldots, c_n)$, when matrix entries are read column-wise from top to bottom and left to right, namely in the order $(a_{11}, a_{21}, \ldots, a_{m1}, a_{12}, \ldots, a_{mn})$?

d) What is the lexicographically smallest contingency table for $(r_1, \ldots, r_m; c_1, \ldots, c_n)$, in the row-wise and column-wise senses?

e) Explain how to generate all contingency tables for $(r_1, \ldots, r_m; c_1, \ldots, c_n)$ in lexicographic order.

62. [*M41*] Show that all contingency tables for $(r_1, \ldots, r_m; c_1, \ldots, c_n)$ can be generated by changing exactly four entries of the matrix at each step.

▸ **63.** [*M30*] Construct a genlex Gray code for all of the $2^s \binom{s+t}{t}$ *subcubes* that have s digits and t asterisks, using only the transformations $*0 \leftrightarrow 0*$, $*1 \leftrightarrow 1*$, $0 \leftrightarrow 1$. For example, one such cycle when $s = t = 2$ is

$$(00{**}, 01{**}, 0{*}1{*}, 0{**}1, 0{**}0, 0{*}0{*}, {*}00{*}, {*}01{*}, {*}0{*}1, {*}0{*}0, {**}00, {**}01,$$
$${**}11, {**}10, {*}1{*}0, {*}1{*}1, {*}11{*}, {*}10{*}, 1{*}0{*}, 1{**}0, 1{**}1, 1{*}1{*}, 11{**}, 10{**}).$$

64. [*M40*] Enumerate the total number of genlex Gray paths on subcubes that use only the transformations allowed in exercise 63. How many of those paths are cycles?

▸ **65.** [*22*] Given $n \geq t \geq 0$, show that there is a Gray path through all of the canonical bases $(\alpha_1, \ldots, \alpha_t)$ of exercise 12, changing just one bit at each step. For example, one such path when $n = 3$ and $t = 2$ is

$$\begin{array}{ccccccc} 001 & 101 & 101 & 001 & 001 & 011 & 010 \\ 010\,, & 010\,, & 110\,, & 110\,, & 100\,, & 100\,, & 100\,. \end{array}$$

66. [*46*] Consider the Ising configurations of exercise 13 for which $a_0 = 0$. Given n, t, and r, is there a Gray code for these configurations in which all transitions have the forms $0^k1 \leftrightarrow 10^k$ or $01^k \leftrightarrow 1^k0$? For example, in the case $n = 9$, $t = 5$, $r = 6$, there is a unique cycle

$$(010101110,\ 010110110,\ 011010110,\ 011011010,\ 011101010,\ 010111010).$$

67. [*M01*] If α is a t-combination, what is (a) $\partial^t\alpha$? (b) $\partial^{t+1}\alpha$?

▸ **68.** [*M22*] How large is the smallest set A of t-combinations for which $\|\partial A\| < \|A\|$?

69. [*M25*] What is the maximum value of $\kappa_t N - N$, for $N \geq 0$?

70. [*M20*] How many t-cliques can a million-edge graph have?

▸ **71.** [*M22*] Show that if N has the degree-t combinatorial representation (57), there is an easy way to find the degree-s combinatorial representation of the complementary number $M = \binom{s+t}{t} - N$, whenever $N < \binom{s+t}{t}$. Derive (63) as a consequence.

72. [*M23*] (A. J. W. Hilton, 1976.) Let A be a set of s-combinations and B a set of t-combinations, both contained in $U = \{0, \ldots, n-1\}$ where $n \geq s + t$. Show that if A and B are *cross-intersecting*, in the sense that $\alpha \cap \beta \neq \emptyset$ for all $\alpha \in A$ and $\beta \in B$, then so are the sets Q_{Mns} and Q_{Nnt} defined in Theorem K, where $M = \|A\|$ and $N = \|B\|$.

73. [*M21*] What are $\|\varrho P_{Nt}\|$ and $\|\varrho Q_{Nnt}\|$ in Theorem K?

74. [*M20*] The right-hand side of (6o) is not always the degree-$(t-1)$ combinatorial representation of $\kappa_t N$, because $v-1$ might be zero. Show, however, that a positive integer N has at most two representations if we allow $v=0$ in (57), and both of them yield the same value $\kappa_t N$ according to (6o). Therefore

$$\kappa_k\kappa_{k+1}\ldots\kappa_t N = \binom{n_t}{k-1} + \binom{n_{t-1}}{k-2} + \cdots + \binom{n_v}{k-1+v-t} \qquad \text{for } 1\le k\le t.$$

75. [*M20*] Find a simple formula for $\kappa_t(N+1)-\kappa_t N$.

▸ **76.** [*M26*] Prove the following properties of the κ functions by manipulating binomial coefficients, without assuming Theorem K:

a) $\kappa_t(M+N)\le\kappa_t M+\kappa_t N$.

b) $\kappa_t(M+N)\le\max(\kappa_t M,N)+\kappa_{t-1}N$.

Hint: $\binom{m_t}{t}+\cdots+\binom{m_1}{1}+\binom{n_t}{t}+\cdots+\binom{n_1}{1}$ is equal to $\binom{m_t\vee n_t}{t}+\cdots+\binom{m_1\vee n_1}{1}+\binom{m_t\wedge n_t}{t}+\cdots+\binom{m_1\wedge n_1}{1}$, where $\vee$ and $\wedge$ denote max and min.

77. [*M22*] Show that Theorem K follows easily from inequality (b) in the previous exercise. Conversely, both inequalities are simple consequences of Theorem K. *Hint:* Any set A of t-combinations can be written $A=A_1+A_0 0$, where $A_1=\{\alpha\in A\mid 0\notin\alpha\}$.

78. [*M23*] Prove that if $t\ge 2$, we have $M\ge\mu_t N$ if and only if $M+\lambda_{t-1}M\ge N$.

79. [*HM26*] (L. Lovász, 1979.) The function $\binom{x}{t}$ increases monotonically from 0 to ∞ as x increases from $t-1$ to ∞; hence we can define

$$\underline{\kappa}_t N = \binom{x}{t-1}, \qquad \text{if } N=\binom{x}{t} \text{ and } x\ge t-1.$$

Prove that $\kappa_t N\ge\underline{\kappa}_t N$ for all integers $t\ge 1$ and $N\ge 0$. *Hint:* Equality holds when x is an integer.

▸ **80.** [*M27*] Show that the minimum shadow sizes in Theorem M are given by (64).

81. [*HM31*] The Takagi function of Fig. 27 is defined for $0\le x\le 1$ by the formula

$$\tau(x)=\sum_{k=1}^{\infty}\int_0^x r_k(t)\,dt,$$

where $r_k(t)=(-1)^{\lfloor 2^k t\rfloor}$ is the Rademacher function of Eq. 7.2.1.1–(16).

a) Prove that $\tau(x)$ is continuous in the interval $[0\,..\,1]$, but its derivative does not exist at any point.

b) Show that $\tau(x)$ is the only continuous function that satisfies

$$\tau(\tfrac12 x) \;=\; \tau(1-\tfrac12 x) \;=\; \tfrac12 x+\tfrac12\tau(x) \qquad \text{for } 0\le x\le 1.$$

c) What is the asymptotic value of $\tau(\epsilon)$ when ϵ is small?

d) Prove that $\tau(x)$ is rational when x is rational.

e) Find all roots of the equation $\tau(x)=1/2$.

f) Find all roots of the equation $\tau(x)=\max_{0\le x\le 1}\tau(x)$.

82. [*HM46*] Determine the set R of all rational numbers r such that the equation $\tau(x)=r$ has uncountably many solutions. If $\tau(x)$ is rational and x is irrational, is it true that $\tau(x)\in R$?

83. [*HM27*] If $T = \binom{2t-1}{t}$, prove the asymptotic formula

$$\kappa_t N - N \;=\; \frac{T}{t}\Big(\tau\Big(\frac{N}{T}\Big) + O\Big(\frac{(\log t)^3}{t}\Big)\Big) \qquad \text{for } 0 \le N \le T.$$

84. [*HM21*] Relate the functions $\lambda_t N$ and $\mu_t N$ to the Takagi function $\tau(x)$.

85. [*M20*] Prove the law of spread/core duality, $X^{\sim+} = X^{\circ\sim}$.

86. [*M21*] True or false: (a) $X \subseteq Y^\circ$ if and only if $Y^\sim \subseteq X^{\sim\circ}$; (b) $X^{\circ+\circ} = X^\circ$; (c) $\alpha M \le N$ if and only if $M \le \beta N$.

87. [*M20*] Explain why cross order is useful, by completing the proof of Lemma S.

88. [*16*] Compute the α and β functions for the $2 \times 2 \times 3$ torus (69).

89. [*M22*] Prove the basic compression lemma, (85).

90. [*M24*] Prove Theorem W for two-dimensional toruses $T(l, m)$, $l \le m$.

91. [*M28*] Let $x = x_1 \dots x_{n-1}$ be the Nth element of the torus $T(m_1, \dots, m_{n-1})$, and let S be the set of all elements of $T(m_1, \dots, m_{n-1}, m)$ that are $\preceq x_1 \dots x_{n-1}(m-1)$ in cross order. If N_a elements of S have final component a, for $0 \le a < m$, prove that $N_{m-1} = N$ and $N_{a-1} = \alpha N_a$ for $1 \le a < m$, where α is the spread function for standard sets in $T(m_1, \dots, m_{n-1})$.

92. [*M25*] (a) Find an N for which the conclusion of Theorem W is false when the parameters $m_1, m_2, \dots, m_n$ have not been sorted into nondecreasing order. (b) Where does the proof of that theorem use the hypothesis that $m_1 \le m_2 \le \cdots \le m_n$?

93. [*M20*] Show that the ∂ half of Corollary C follows from the θ half. *Hint:* The complements of the multicombinations (91) with respect to U are 3211, 3210, 3200, 3110, 3100, 3000, 2110, 3100, 2000, 1000.

94. [*15*] Explain why Theorems K and M follow from Corollary C.

▸ **95.** [*M22*] If S is an infinite sequence $(s_0, s_1, s_2, \dots)$ of positive integers, let

$$\binom{S(n)}{k} = [z^k] \prod_{j=0}^{n-1} (1 + z + \cdots + z^{s_j});$$

thus $\binom{S(n)}{k}$ is the ordinary binomial coefficient $\binom{n}{k}$ if $s_0 = s_1 = s_2 = \cdots = 1$.

Generalizing the combinatorial number system, show that every nonnegative integer N has a unique representation

$$N = \binom{S(n_t)}{t} + \binom{S(n_{t-1})}{t-1} + \cdots + \binom{S(n_1)}{1}$$

where $n_t \ge n_{t-1} \ge \cdots \ge n_1 \ge 0$ and $\{n_t, n_{t-1}, \dots, n_1\} \subseteq \{s_0 \cdot 0, s_1 \cdot 1, s_2 \cdot 2, \dots\}$. Use this representation to give a simple formula for the numbers $\|\partial P_{Nt}\|$ in Corollary C.

▸ **96.** [*M26*] The text remarked that the vertices of a convex polyhedron can be perturbed slightly so that all of its faces are simplexes. In general, any set of combinations that contains the shadows of all its elements is called a *simplicial complex*; thus C is a simplicial complex if and only if $\alpha \subseteq \beta$ and $\beta \in C$ implies that $\alpha \in C$, if and only if C is an order ideal with respect to set inclusion.

The *size vector* of a simplicial complex C on n vertices is $(N_0, N_1, \dots, N_n)$ when C contains exactly N_t combinations of size t.

a) What are the size vectors of the five regular solids (the tetrahedron, cube, octahedron, dodecahedron, and icosahedron), when their vertices are slightly tweaked?

b) Construct a simplicial complex with size vector $(1, 4, 5, 2, 0)$.
c) Find a necessary and sufficient condition that a given size vector $(N_0, N_1, \ldots, N_n)$ is feasible.
d) Prove that $(N_0, \ldots, N_n)$ is feasible if and only its "dual" vector $(\overline{N}_0, \ldots, \overline{N}_n)$ is feasible, where we define $\overline{N}_t = \binom{n}{t} - N_{n-t}$.
e) List all feasible size vectors $(N_0, N_1, N_2, N_3, N_4)$ and their duals. Which of them are self-dual?

97. [*30*] Continuing exercise 96, find an efficient way to count the number of feasible size vectors $(N_0, N_1, \ldots, N_n)$ when $n \le 100$.

98. [*M25*] A *clutter* is a set C of combinations that are incomparable, in the sense that $\alpha \subseteq \beta$ and $\alpha, \beta \in C$ implies $\alpha = \beta$. The size vector of a clutter is defined as in exercise 96.
a) Find a necessary and sufficient condition that $(M_0, M_1, \ldots, M_n)$ is the size vector of a clutter.
b) List all such size vectors in the case $n = 4$.

▸ **99.** [*M30*] (Clements and Lindström.) Let A be a "simplicial multicomplex," a set of submultisets of the multiset U in Corollary C with the property that $\partial A \subseteq A$. How large can the total weight $\nu A = \sum\{\|\alpha\| \mid \alpha \in A\}$ be when $\|A\| = N$?

100. [*M25*] If $f(x_1, \ldots, x_n)$ is a Boolean formula, let $F(p)$ be the probability that $f(x_1, \ldots, x_n) = 1$ when each variable x_j independently is 1 with probability p.
a) Calculate $G(p)$ and $H(p)$ for the Boolean formulas $g(w, x, y, z) = wxz \vee wyz \vee xy\bar{z}$, $h(w, x, y, z) = \bar{w}yz \vee xyz$.
b) Show that there is a *monotone* Boolean function $f(w, x, y, z)$ such that $F(p) = G(p)$, but there is no such function with $F(p) = H(p)$. Explain how to test this condition in general.

101. [*HM35*] (F. S. Macaulay, 1927.) A *polynomial ideal* I in the variables $\{x_1 \ldots, x_s\}$ is a set of polynomials closed under the operations of addition, multiplication by a constant, and multiplication by any of the variables. It is called *homogeneous* if it consists of all linear combinations of a set of homogeneous polynomials, namely of polynomials like $xy + z^2$ whose terms all have the same degree. Let N_t be the maximum number of linearly independent elements of degree t in I. For example, if $s = 2$, the set of all $\alpha(x_0, x_1, x_2)(x_0x_1^2 - 2x_1x_2^2) + \beta(x_0, x_1, x_2)x_0x_1x_2^2$, where α and β run through all possible polynomials in $\{x_0, x_1, x_2\}$, is a homogeneous polynomial ideal with $N_0 = N_1 = N_2 = 0$, $N_3 = 1$, $N_4 = 4$, $N_5 = 9$, $N_6 = 15$, ...
a) Prove that for any such ideal I there is another ideal I' in which all homogeneous polynomials of degree t are linear combinations of N_t independent *monomials*. (A monomial is a product of variables, like $x_1^3x_2x_5^4$.)
b) Use Theorem M and (6_4) to prove that $N_{t+1} \ge N_t + \kappa_s N_t$ for all $t \ge 0$.
c) Show that $N_{t+1} > N_t + \kappa_s N_t$ occurs for only finitely many t. (This statement is equivalent to "Hilbert's basis theorem," proved by David Hilbert in *Göttinger Nachrichten* (1888), 450–457; *Math. Annalen* **36** (1890), 473–534.)

▸ **102.** [*M38*] The shadow of a subcube $a_1 \ldots a_n$, where each a_j is either 0 or 1 or $*$, is obtained by replacing some $*$ by 0 or 1. For example,

$$\partial 0{*}11{*}0 \;=\; \{0011{*}0,\ 011{*}0,\ 0{*}1100,\ 0{*}1110\}.$$

Find a set P_{Nst} such that, if A is any set of N subcubes $a_1 \ldots a_n$ having s digits and t asterisks, $\|\partial A\| \ge \|P_{Nst}\|$.

103. [*M41*] The shadow of a binary string $a_1 \ldots a_n$ is obtained by deleting one of its bits. For example,

$$\partial 110010010 = \{10010010,\ 11010010,\ 11000010,\ 11001000,\ 11001001\}.$$

Find a set P_{Nn} such that, if A is any set of N binary strings $a_1 \ldots a_n$, $\|\partial A\| \geq \|P_{Nn}\|$.

104. [*M20*] A *universal cycle of t-combinations* for $\{0, 1, \ldots, n-1\}$ is a cycle of $\binom{n}{t}$ numbers whose blocks of t consecutive elements run through every t-combination $\{c_1, \ldots, c_t\}$. For example,

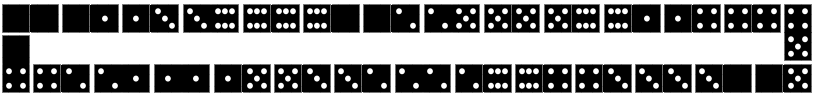
$$(0214506132051624315263042536 4103546)$$

is a universal cycle when $t = 3$ and $n = 7$.

Prove that no such cycle is possible unless $\binom{n}{t}$ is a multiple of n.

105. [*M21*] (L. Poinsot, 1809.) Find a "nice" universal cycle of 2-combinations for $\{0, 1, \ldots, 2m\}$. *Hint:* Consider the differences of consecutive elements, mod $(2m+1)$.

106. [*22*] (O. Terquem, 1849.) Poinsot's theorem implies that all 28 dominoes of a traditional "double-six" set can be arranged in a cycle so that the spots of adjacent dominoes match each other:



How many such cycles are possible?

107. [*M31*] Find universal cycles of 3-combinations for the sets $\{0, \ldots, n-1\}$ when $n \bmod 3 \neq 0$.

108. [*M31*] Find universal cycles of 3-*multicombinations* for $\{0, 1, \ldots, n-1\}$ when $n \bmod 3 \neq 0$ (namely for combinations $d_1 d_2 d_3$ with repetitions permitted).

▸ **109.** [*26*] *Cribbage* is a game played with 52 cards, where each card has a suit (♣, ♢, ♡, or ♠) and a face value (A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, or K). One feature of the game is to compute the score of a 5-card combination $C = \{c_1, c_2, c_3, c_4, c_5\}$, where one card c_k is called the *starter*. The score is the sum of points computed as follows, for each subset S of C and each choice of k: Let $\|S\| = s$.

i) Fifteens: If $\sum\{v(c) \mid c \in S\} = 15$, where $\big(v(\texttt{A}), v(\texttt{2}), v(\texttt{3}), \ldots, v(\texttt{9}), v(\texttt{10}), v(\texttt{J}), v(\texttt{Q}), v(\texttt{K})\big) = (1, 2, 3, \ldots, 9, 10, 10, 10, 10)$, score two points.

ii) Pairs: If $s = 2$ and both cards have the same face value, score two points.

iii) Runs: If $s \geq 3$ and the face values are consecutive, and if C does not contain a run of length $s+1$, score s points.

iv) Flushes: If $s = 4$ and all cards of S have the same suit, and if $c_k \notin S$, score $4 + [c_k$ has the same suit as the others$]$.

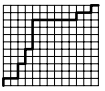
v) Nobs: If $s = 1$ and $c_k \notin S$, score 1 if the card is J of the same suit as c_k.

For example, if you hold {J♣, 5♣, 5♢, 6♡} and if 4♣ is the starter, you score 4×2 for fifteens, 2 for a pair, 2×3 for runs, plus 1 for nobs, totalling 17.

Exactly how many combinations and starter choices lead to a score of x points, for $x = 0, 1, 2, \ldots$?

SECTION 7.2.1.3

1. Given a multiset, form the sequence $e_t \dots e_2 e_1$ from right to left by listing the distinct elements first, then those that appear twice, then those that appear thrice, etc. Let us set $e_{-j} \leftarrow s - j$ for $0 \le j \le s$, so that every element e_j for $1 \le j \le t$ is equal to some element to its right in the sequence $e_t \dots e_1 e_0 \dots e_{-s}$. If the first such element is $e_{c_j - s}$, we obtain a solution to (3). Conversely, every solution to (3) yields a unique multiset $\{e_1, \dots, e_t\}$, because $c_j < s + j$ for $1 \le j \le t$.

2. Start at the bottom left corner; then go up for each 0, go right for each 1. The result is  .

3. In this algorithm, variable r is the least positive index such that $q_r > 0$.

F1. [Initialize.] Set $q_j \leftarrow 0$ for $1 \le j \le t$, and $q_0 \leftarrow s$. (We assume that $st > 0$.)

F2. [Visit.] Visit the composition $q_t \dots q_0$. Go to F4 if $q_0 = 0$.

F3. [Easy case.] Set $q_0 \leftarrow q_0 - 1$, $r \leftarrow 1$, and go to F5.

F4. [Tricky case.] Terminate if $r = t$. Otherwise set $q_0 \leftarrow q_r - 1$, $q_r \leftarrow 0$, $r \leftarrow r + 1$.

F5. [Increase q_r.] Set $q_r \leftarrow q_r + 1$ and return to F2. ∎

[See *CACM* **11** (1968), 430; **12** (1969), 187. The task of generating such compositions in *decreasing* lexicographic order is more difficult.]

4. We can reverse the roles of 0 and 1 in (14), so that $0^{q_t} 10^{q_{t-1}} 1 \dots 10^{q_1} 10^{q_0} = 1^{r_s} 01^{r_{s-1}} 0 \dots 01^{r_1} 01^{r_0}$. This gives $0^1 10^0 10^2 10^2 10^4 10^0 10^0 10^0 10^0 10^0 10^1 10^0 10^1 10^0 = 1^0 01^2 01^0 01^1 01^0 01^1 01^0 01^0 01^0 01^6 01^2 01^1$. Lexicographic order of $a_{n-1} \dots a_1 a_0$ corresponds to lexicographic order of $r_s \dots r_1 r_0$.

Incidentally, there's also a multiset connection: $\{d_t, \dots, d_1\} = \{r_s \cdot s, \dots, r_0 \cdot 0\}$. For example, $\{10, 10, 8, 6, 2, 2, 2, 2, 2, 2, 1, 1, 0\} = \{0 \cdot 11, 2 \cdot 10, 0 \cdot 9, 1 \cdot 8, 0 \cdot 7, 1 \cdot 6, 0 \cdot 5, 0 \cdot 4, 0 \cdot 3, 6 \cdot 2, 2 \cdot 1, 1 \cdot 0\}$.

5. (a) Set $x_j = c_j - \lfloor (j-1)/2 \rfloor$ in each t-combination of $n + \lfloor t/2 \rfloor$. (b) Set $x_j = c_j + j + 1$ in each t-combination of $n - t - 2$.

(A similar approach finds all solutions $(x_t, \dots, x_1)$ to the inequalities $x_{j+1} \ge x_j + \delta_j$ for $0 \le j \le t$, given the values of x_{t+1}, $(\delta_t, \dots, \delta_1)$, and x_0.)

6. Assume that $t > 0$. We get to T3 when $c_1 > 0$; to T5 when $c_2 = c_1 + 1 > 1$; to T4 for $2 \le j \le t+1$ when $c_j = c_1 + j - 1 \ge j$. So the counts are: T1, 1; T2, $\binom{n}{t}$; T3, $\binom{n-1}{t}$; T4, $\binom{n-2}{t-1} + \binom{n-2}{t-2} + \cdots + \binom{n-t-1}{0} = \binom{n-1}{t-1}$; T5, $\binom{n-2}{t-1}$; T6, $\binom{n-1}{t-1} + \binom{n-2}{t-1} - 1$.

7. A procedure slightly simpler than Algorithm T suffices: Assume that $s < n$.

S1. [Initialize.] Set $b_j \leftarrow j + n - s - 1$ for $1 \le j \le s$; then set $j \leftarrow 1$.

S2. [Visit.] Visit the combination $b_s \dots b_2 b_1$. Terminate if $j > s$.

S3. [Decrease b_j.] Set $b_j \leftarrow b_j - 1$. If $b_j < j$, set $j \leftarrow j + 1$ and return to S2.

S4. [Reset $b_{j-1} \dots b_1$.] While $j > 1$, set $b_{j-1} \leftarrow b_j - 1$, $j \leftarrow j - 1$, and repeat until $j = 1$. Go to S2. ∎

(See S. Dvořák, *Comp. J.* **33** (1990), 188. Notice that if $x_k = n - b_k$ for $1 \le k \le s$, this algorithm runs through all combinations $x_s \dots x_2 x_1$ of $\{1, 2, \dots, n\}$ with $1 \le x_s < \cdots < x_2 < x_1 \le n$, in *increasing* lexicographic order.)

8. **A1.** [Initialize.] Set $a_n \dots a_0 \leftarrow 0^{s+1}1^t$, $q \leftarrow t$, $r \leftarrow 0$. (We assume that $0 < t < n$.)

A2. [Visit.] Visit the combination $a_{n-1} \dots a_1 a_0$. Go to A4 if $q = 0$.

A3. [Replace $\dots 01^q$ by $\dots 101^{q-1}$.] Set $a_q \leftarrow 1$, $a_{q-1} \leftarrow 0$, $q \leftarrow q - 1$; then if $q = 0$, set $r \leftarrow 1$. Return to A2.

A4. [Shift block of 1s.] Set $a_r \leftarrow 0$ and $r \leftarrow r + 1$. Then if $a_r = 1$, set $a_q \leftarrow 1$, $q \leftarrow q + 1$, and repeat step A4.

A5. [Carry to left.] Terminate if $r = n$; otherwise set $a_r \leftarrow 1$.

A6. [Odd?] If $q > 0$, set $r \leftarrow 0$. Return to A2. ∎

In step A2, q and r point respectively to the rightmost 0 and 1 in $a_{n-1} \dots a_0$. Steps A1, ..., A6 are executed with frequency 1, $\binom{n}{t}$, $\binom{n-1}{t-1}$, $\binom{n}{t} - 1$, $\binom{n-1}{t}$, $\binom{n-1}{t} - 1$.

9. (a) The first $\binom{n-1}{t}$ strings begin with 0 and have $2A_{(s-1)t}$ bit changes; the other $\binom{n-1}{t-1}$ begin with 1 and have $2A_{s(t-1)}$. And $\nu(01^t0^{s-1} \oplus 10^s1^{t-1}) = 2\min(s,t)$.

(b) Solution 1 (direct): Let $B_{st} = A_{st} + \min(s,t) + 1$. Then

$$B_{st} = B_{(s-1)t} + B_{s(t-1)} + [s=t] \quad \text{when } st > 0; \qquad B_{st} = 1 \quad \text{when } st = 0.$$

Consequently $B_{st} = \sum_{k=0}^{\min(s,t)} \binom{s+t-2k}{s-k}$. If $s \le t$ this is $\le \sum_{k=0}^{s} \binom{s+t-k}{s-k} = \binom{s+t+1}{s} = \binom{s+t}{s}\frac{s+t+1}{t+1} < 2\binom{s+t}{t}$.

Solution 2 (indirect): The algorithm in answer 8 makes $2(x+y)$ bit changes when steps (A3, A4) are executed (x, y) times. Thus $A_{st} \le \binom{n-1}{t-1} + \binom{n}{t} - 1 < 2\binom{n}{t}$.

[The comment in answer 7.2.1.1–3 therefore applies to combinations as well.]

10. Each scenario corresponds to a $(4,4)$-combination $b_4b_3b_2b_1$ or $c_4c_3c_2c_1$ in which A wins games $\{8-b_4, 8-b_3, 8-b_2, 8-b_1\}$ and N wins games $\{8-c_4, 8-c_3, 8-c_2, 8-c_1\}$, because we can assume that the losing team wins the remaining games in a series of 8. (Equivalently, we can generate all permutations of {A, A, A, A, N, N, N, N} and omit the trailing run of As or Ns.) The American League wins if and only if $b_1 \neq 0$, if and only if $c_1 = 0$. The formula $\binom{c_4}{4} + \binom{c_3}{3} + \binom{c_2}{2} + \binom{c_1}{1}$ assigns a unique integer between 0 and 69 to each scenario.

For example, ANANAA $\iff a_7 \dots a_1a_0 = 01010011 \iff b_4b_3b_2b_1 = 7532 \iff c_4c_3c_2c_1 = 6410$, and this is the scenario of rank $\binom{6}{4} + \binom{4}{3} + \binom{1}{2} + \binom{0}{1} = 19$ in lexicographic order. (Notice that the term $\binom{c_j}{j}$ will be zero if and only if it corresponds to a trailing N.)

11. AAAA (9 times), NNNN (8), and ANAAA (7) were most common. Exactly 27 of the 70 failed to occur, including all four beginning with NNNA. (We disregard the games that were tied because of darkness, in 1907, 1912, and 1922. The case ANNAAA should perhaps be excluded too, because it occurred only in 1920 as part of ANNAAAA in a best-of-nine series. The scenario NNAAANN occurred for the first time in 2001.)

12. (a) Let V_j be the subspace $\{a_{n-1} \dots a_0 \in V \mid a_k = 0 \text{ for } 0 \le k < j\}$, so that $\{0 \dots 0\} = V_n \subseteq V_{n-1} \subseteq \cdots \subseteq V_0 = V$. Then $\{c_1, \dots, c_t\} = \{c \mid V_c \neq V_{c+1}\}$, and α_k is the unique element $a_{n-1} \dots a_0$ of V with $a_{c_j} = [j=k]$ for $1 \le j \le t$.

Incidentally, the $t \times n$ matrix corresponding to a canonical basis is said to be in *reduced row-echelon form*. It can be found by a standard "triangulation" algorithm (see exercise 4.6.1–19 and Algorithm 4.6.2N).

(b) The 2-nomial coefficient $\binom{n}{t}_2 = 2^t\binom{n-1}{t}_2 + \binom{n-1}{t-1}_2$ of exercise 1.2.6–58 has the right properties, because $2^t\binom{n-1}{t}_2$ binary vector spaces have $c_t < n-1$ and $\binom{n-1}{t-1}_2$ have $c_t = n-1$. [In general the number of canonical bases with r asterisks is the number of

partitions of r into at most t parts, with no part exceeding $n-t$, and this is $[z^r]\binom{n}{t}_z$ by Eq. 7.2.1.4–(oo). See D. E. Knuth, *J. Combinatorial Theory* **10** (1971), 178–180.]

(c) The following algorithm assumes that $n > t > 0$ and that $a_{(t+1)j} = 0$ for $t \le j \le n$.

V1. [Initialize.] Set $a_{kj} \leftarrow [j = k-1]$ for $1 \le k \le t$ and $0 \le j < n$. Also set $q \leftarrow t$, $r \leftarrow 0$.

V2. [Visit.] (At this point we have $a_{k(k-1)} = 1$ for $1 \le k \le q$, $a_{(q+1)q} = 0$, and $a_{1r} = 1$.) Visit the canonical basis $(a_{1(n-1)} \ldots a_{11}a_{10}, \ldots, a_{t(n-1)} \ldots a_{t1}a_{t0})$. Go to V4 if $q > 0$.

V3. [Find block of 1s.] Set $q \leftarrow 1, 2, \ldots$, until $a_{(q+1)(q+r)} = 0$. Terminate if $q + r = n$.

V4. [Add 1 to column $q+r$.] Set $k \leftarrow 1$. If $a_{k(q+r)} = 1$, set $a_{k(q+r)} \leftarrow 0$, $k \leftarrow k+1$, and repeat until $a_{k(q+r)} = 0$. Then if $k \le q$, set $a_{k(q+r)} \leftarrow 1$; otherwise set $a_{q(q+r)} \leftarrow 1$, $a_{q(q+r-1)} \leftarrow 0$, $q \leftarrow q-1$.

V5. [Shift block right.] If $q = 0$, set $r \leftarrow r+1$. Otherwise, if $r > 0$, set $a_{k(k-1)} \leftarrow 1$ and $a_{k(r+k-1)} \leftarrow 0$ for $1 \le k \le q$, then set $r \leftarrow 0$. Go to V2. ∎

Step V2 finds $q > 0$ with probability $1 - (2^{n-t}-1)/(2^n-1) \approx 1 - 2^{-t}$, so we could save time by treating this case separately.

(d) Since $999999 = 4\binom{8}{4}_2 + 16\binom{7}{4}_2 + 5\binom{6}{3}_2 + 5\binom{5}{3}_2 + 8\binom{4}{3}_2 + 0\binom{3}{2}_2 + 4\binom{2}{2}_2 + 1\binom{1}{1}_2 + 2\binom{0}{1}_2$, the millionth output has binary columns 4, 16/2, 5, 5, 8/2, 0, 4/2, 1, 2/2, namely

$$\begin{aligned}
\alpha_1 &= 0\,0\,1\,1\,0\,0\,0\,1\,1,\\
\alpha_2 &= 0\,0\,0\,0\,0\,0\,1\,0\,0,\\
\alpha_3 &= 1\,0\,1\,1\,1\,0\,0\,0\,0,\\
\alpha_4 &= 0\,1\,0\,0\,0\,0\,0\,0\,0.
\end{aligned}$$

[*Reference:* E. Calabi and H. S. Wilf, *J. Combinatorial Theory* **A22** (1977), 107–109.]

13. Let $n = s + t$. There are $\binom{s-1}{\lceil (r-1)/2 \rceil}\binom{t-1}{\lfloor (r-1)/2 \rfloor}$ configurations beginning with 0 and $\binom{s-1}{\lfloor (r-1)/2 \rfloor}\binom{t-1}{\lceil (r-1)/2 \rceil}$ beginning with 1, because an Ising configuration that begins with 0 corresponds to a composition of s 0s into $\lceil (r+1)/2 \rceil$ parts and a composition of t 1s into $\lfloor (r+1)/2 \rfloor$ parts. We can generate all such pairs of compositions and weave them into configurations. [See E. Ising, *Zeitschrift für Physik* **31** (1925), 253–258; J. M. S. Simões Pereira, *CACM* **12** (1969), 562.]

14. Start with $l[j] \leftarrow j-1$ and $r[j-1] \leftarrow j$ for $1 \le j \le n$; $l[0] \leftarrow n$, $r[n] \leftarrow 0$. To get the next combination, assuming that $t > 0$, set $p \leftarrow s$ if $l[0] > s$, otherwise $p \leftarrow r[n]-1$. Terminate if $p \le 0$; otherwise set $q \leftarrow r[p]$, $l[q] \leftarrow l[p]$, and $r[l[p]] \leftarrow q$. Then if $r[q] > s$ and $p < s$, set $r[p] \leftarrow r[n]$, $l[r[n]] \leftarrow p$, $r[s] \leftarrow r[q]$, $l[r[q]] \leftarrow s$, $r[n] \leftarrow 0$, $l[0] \leftarrow n$; otherwise set $r[p] \leftarrow r[q]$, $l[r[q]] \leftarrow p$. Finally set $r[q] \leftarrow p$ and $l[p] \leftarrow q$.

[See Korsh and Lipschutz, *J. Algorithms* **25** (1997), 321–335, where the idea is extended to a loopless algorithm for multiset permutations. *Caution:* This exercise, like exercise 7.2.1.1–16, is more academic than practical, because the routine that visits the linked list might need a loop that nullifies any advantage of loopless generation.]

15. (The stated fact is true because lexicographic order of $c_t \ldots c_1$ corresponds to lexicographic order of $a_{n-1} \ldots a_0$, which is reverse lexicographic order of the complementary sequence $1 \ldots 1 \oplus a_{n-1} \ldots a_0$.) By Theorem L, the combination $c_t \ldots c_1$ is

visited *before* exactly $\binom{b_s}{s}+\cdots+\binom{b_2}{2}+\binom{b_1}{1}$ others have been visited, and we must have

$$\binom{b_s}{s}+\cdots+\binom{b_1}{1}+\binom{c_t}{t}+\cdots+\binom{c_1}{1} \;=\; \binom{s+t}{t}-1.$$

This general identity can be written

$$\sum_{j=0}^{n-1} x_j \binom{j}{x_0+\cdots+x_j} + \sum_{j=0}^{n-1} \bar{x}_j \binom{j}{\bar{x}_0+\cdots+\bar{x}_j} \;=\; \binom{n}{x_0+\cdots+x_{n-1}} - 1$$

when each x_j is 0 or 1, and $\bar{x}_j = 1 - x_j$; it follows also from the equation

$$x_n \binom{n}{x_0+\cdots+x_n} + \bar{x}_n \binom{n}{\bar{x}_0+\cdots+\bar{x}_n} \;=\; \binom{n+1}{x_0+\cdots+x_n} - \binom{n}{x_0+\cdots+x_{n-1}}.$$

16. Since $999999 = \binom{1414}{2}+\binom{1008}{1} = \binom{182}{3}+\binom{153}{2}+\binom{111}{1} = \binom{71}{4}+\binom{56}{3}+\binom{36}{2}+\binom{14}{1} = \binom{43}{5}+\binom{32}{4}+\binom{21}{3}+\binom{15}{2}+\binom{6}{1}$, the answers are (a) 1414 1008; (b) 182 153 111; (c) 71 56 36 14; (d) 43 32 21 15 6; (e) 1000000 999999 ... 2 0.

17. By Theorem L, n_t is the largest integer such that $N \geq \binom{n_t}{t}$; the remaining terms are the degree-$(t-1)$ representation of $N - \binom{n_t}{t}$.

A simple sequential method for $t > 1$ starts with $x = 1$, $c = t$, and sets $c \leftarrow c+1$, $x \leftarrow xc/(c-t)$ zero or more times until $x > N$; then we complete the first phase by setting $x \leftarrow x(c-t)/c$, $c \leftarrow c-1$, at which point we have $x = \binom{c}{t} \leq N < \binom{c+1}{t}$. Set $n_t \leftarrow c$, $N \leftarrow N - x$; terminate with $n_1 \leftarrow N$ if $t = 2$; otherwise set $x \leftarrow xt/c$, $t \leftarrow t-1$, $c \leftarrow c-1$; while $x > N$ set $x \leftarrow x(c-t)/c$, $c \leftarrow c-1$; repeat. This method requires $O(n)$ arithmetic operations if $N < \binom{n}{t}$, so it is suitable unless t is small and N is large.

When $t = 2$, exercise 1.2.4–41 tells us that $n_2 = \lfloor\sqrt{2N+2} + \frac{1}{2}\rfloor$. In general, n_t is $\lfloor x \rfloor$ where x is the largest root of $x^{\underline{t}} = t!\,N$; this root can be approximated by reverting the series $y = (x^{\underline{t}})^{1/t} = x - \frac{1}{2}(t-1) + \frac{1}{24}(t^2-1)x^{-1} + \cdots$ to get $x = y + \frac{1}{2}(t-1) + \frac{1}{24}(t^2-1)/y + O(y^{-3})$. Setting $y = (t!\,N)^{1/t}$ in this formula gives a good approximation, after which we can check that $\binom{\lfloor x\rfloor}{t} \leq N < \binom{\lfloor x\rfloor+1}{t}$ or make a final adjustment. [See A. S. Fraenkel and M. Mor, *Comp. J.* **26** (1983), 336–343.]

18. A complete binary tree of $2^n - 1$ nodes is obtained, with an extra node at the top, like the "tree of losers" in replacement selection sorting (Fig. 63 in Section 5.4.1). Therefore explicit links aren't necessary; the right child of node k is node $2k+1$, and the left sibling is node $2k$, for $1 \leq k < 2^{n-1}$.

This representation of a binomial tree has the curious property that node $k = (0^a 1\alpha)_2$ corresponds to the combination whose binary string is $0^a 1 \alpha^R$.

19. It is post(1000000), where $\text{post}(n) = 2^k + \text{post}(n - 2^k + 1)$ if $2^k \leq n < 2^{k+1}$, and $\text{post}(0) = 0$. So it is 11110100001001000100.

20. $f(z) = (1+z^{w_{n-1}})\ldots(1+z^{w_1})/(1-z)$, $g(z) = (1+z^{w_0})f(z)$, $h(z) = z^{w_0}f(z)$.

21. The rank of $c_t \ldots c_2 c_1$ is $\binom{c_t+1}{t}-1$ minus the rank of $c_{t-1}\ldots c_2 c_1$. [See H. Lüneburg, *Abh. Math. Sem. Hamburg* **52** (1982), 208–227.]

22. Since $999999 = \binom{1415}{2}-\binom{406}{1} = \binom{183}{3}-\binom{98}{2}+\binom{21}{1} = \binom{72}{4}-\binom{57}{3}+\binom{32}{2}-\binom{27}{1} = \binom{44}{5}-\binom{40}{4}+\binom{33}{3}-\binom{13}{2}+\binom{3}{1}$, the answers are (a) 1414 405; (b) 182 97 21; (c) 71 56 31 26; (d) 43 39 32 12 3; (e) 1000000 999999 999998 999996 ... 0.

23. There are $\binom{n-r}{t-r}$ combinations with $j > r$, for $r = 1, 2, \ldots, t$. (If $r = 1$ we have $c_2 = c_1 + 1$; if $r = 2$ we have $c_1 = 0$, $c_2 = 1$; if $r = 3$ we have $c_1 = 0$, $c_2 = 1$, $c_4 = c_3 + 1$; etc.) Thus the mean is $\big(\binom{n}{t}+\binom{n-1}{t-1}+\cdots+\binom{n-t}{0}\big)/\binom{n}{t} = \binom{n+1}{t}/\binom{n}{t} = (n+1)/(n+1-t)$.

The average running time per step is approximately proportional to this quantity; thus the algorithm is quite fast when t is small, but slow if t is near n.

24. In fact $j_k - 2 \le j_{k+1} \le j_k + 1$ when $j_k \equiv t$ (modulo 2) and $j_k - 1 \le j_{k+1} \le j_k + 2$ when $j_k \not\equiv t$, because R5 is performed only when $c_i = i - 1$ for $1 \le i < j$.

Thus we could say, "If $j \ge 4$, set $j \leftarrow j-1-[j \text{ odd}]$ and go to R5" at the end of R2, if t is odd; "If $j \ge 3$, set $j \leftarrow j-1-[j \text{ even}]$ and go to R5" if t is even. The algorithm will then be loopless, since R4 and R5 will be performed at most twice per visit.

25. Assume that $N > N'$ and $N - N'$ is minimum; furthermore let c_t be minimum, subject to those assumptions. Then $c_t > c'_t$.

If there is an element $x \notin C \cup C'$ with $0 \le x < c_t$, map each t-combination of $C \cup C'$ by changing $j \mapsto j-1$ for $j > x$; or, if there is an element $x \in C \cap C'$, map each t-combination that contains x into a $(t-1)$-combination by omitting x and changing $j \mapsto x - j$ for $j < x$. In either case the mapping preserves alternating lexicographic order; hence $N - N'$ must exceed the number of combinations between the images of C and C'. But c_t is minimum, so no such x can exist. Consequently $t = m$ and $c_t = 2m - 1$.

Now if $c'_m < c_m - 1$, we could decrease $N - N'$ by increasing c'_m. Therefore $c'_m = 2m-2$, and the problem has been reduced to finding the *maximum* of $\operatorname{rank}(c_{m-1} \dots c_1) - \operatorname{rank}(c'_{m-1} \dots c'_1)$, where rank refers to Gray binary code.

Let $f(s,t) = \max\bigl(\operatorname{rank}(b_s \dots b_1) - \operatorname{rank}(c_t \dots c_1)\bigr)$ over all $\{b_s, \dots, b_1, c_t \dots, c_1\} = \{0, \dots, s+t-1\}$. Then $f(s,t)$ satisfies the curious recurrence

$$f(s,0) = f(0,t) = 0; \qquad f(1,t) = t;$$
$$f(s,t) = \tbinom{s+t-1}{s} + \max\bigl(f(t-1,s-1), f(s-2,t)\bigr) \quad \text{if } st > 0.$$

When $s + t = 2u + 2$ the solution turns out to be

$$f(s,t) = \binom{2u+1}{t-1} + \sum_{j=1}^{u-r} \binom{2u+1-2j}{r} + \sum_{j=0}^{r-1} \binom{2j+1}{j}, \qquad r = \min(s-2, t-1),$$

with the maximum occurring at $f(t-1,s-1)$ when $s \le t$ and at $f(s-2,t)$ when $s \ge t+2$.

Therefore the minimum $N - N'$ occurs for

$$C = \{2m-1\} \cup \{2m-2-x \mid 1 \le x \le 2m-2, \quad x \bmod 4 \le 1\},$$
$$C' = \{2m-2\} \cup \{2m-2-x \mid 1 \le x \le 2m-2, \quad x \bmod 4 \ge 2\};$$

and it equals $\binom{2m-1}{m-1} - \sum_{k=0}^{m-2} \binom{2k+1}{k} = 1 + \sum_{k=1}^{m-1} \binom{2k}{k-1}$. [A. J. van Zanten, *IEEE Trans.* **IT-37** (1991), 1229–1233.]

26. (a) Yes: The first is $0^{n-\lceil t/2 \rceil} 1^{t \bmod 2} 2^{\lfloor t/2 \rfloor}$ and the last is $2^{\lfloor t/2 \rfloor} 1^{t \bmod 2} 0^{n-\lceil t/2 \rceil}$; transitions are substrings of the forms $02^a1 \leftrightarrow 12^a0$, $02^a2 \leftrightarrow 12^a1$, $10^a1 \leftrightarrow 20^a0$, $10^a2 \leftrightarrow 20^a1$.

(b) No: If $s = 0$ there is a big jump from 02^t0^{r-1} to 20^r2^{t-1}.

27. The following procedure extracts all combinations $c_1 \dots c_k$ of Γ_n that have weight $\le t$: Begin with $k \leftarrow 0$ and $c_0 \leftarrow n$. Visit $c_1 \dots c_k$. If k is even and $c_k = 0$, set $k \leftarrow k-1$; if k is even and $c_k > 0$, set $c_k \leftarrow c_k - 1$ if $k = t$, otherwise $k \leftarrow k+1$ and $c_k \leftarrow 0$. On the other hand if k is odd and $c_k + 1 = c_{k-1}$, set $k \leftarrow k-1$ and $c_k \leftarrow c_{k+1}$ (but terminate if $k = 0$); if k is odd and $c_k + 1 < c_{k-1}$, set $c_k \leftarrow c_k + 1$ if $k = t$, otherwise $k \leftarrow k+1$, $c_k \leftarrow c_{k-1}$, $c_{k-1} \leftarrow c_k + 1$. Repeat.

(This loopless algorithm reduces to that of exercise 7.2.1.1–12(b) when $t = n$, with slight changes of notation.)

28. True. Bit strings $a_{n-1}\ldots a_0=\alpha\beta$ and $a'_{n-1}\ldots a'_0=\alpha\beta'$ correspond to index lists $(b_s\ldots b_1=\theta\chi,\ c_t\ldots c_1=\phi\psi)$ and $(b'_s\ldots b'_1=\theta\chi',\ c'_t\ldots c'_1=\phi\psi')$ such that everything between $\alpha\beta$ and $\alpha\beta'$ begins with α if and only if everything between $\theta\chi$ and $\theta\chi'$ begins with θ and everything between $\phi\psi$ and $\phi\psi'$ begins with ϕ. For example, if $n=10$, the prefix $\alpha=01101$ corresponds to prefixes $\theta=96$ and $\phi=875$.

(But just having $c_t\ldots c_1$ in genlex order is a much weaker condition. For example, *every* such sequence is genlex when $t=1$.)

29. (a) $\texttt{-}^k\texttt{0}^{l+1}$ or $\texttt{-}^k\texttt{0}^{l+1}\texttt{+}\pm^m$ or $\pm^k$, for $k,l,m\ge 0$.

(b) No; the successor is always smaller in balanced ternary notation.

(c) For all α and all $k,l,m\ge 0$ we have $\alpha\texttt{0-}^{k+1}\texttt{0}^{l}\texttt{+}\pm^m\to\alpha\texttt{-+}^k\texttt{0}^{l+1}\texttt{-}\pm^m$ and $\alpha\texttt{+-}^k\texttt{0}^{l+1}\texttt{+}\pm^m\to\alpha\texttt{0+}^{k+1}\texttt{0}^{l}\texttt{-}\pm^m$; also $\alpha\texttt{0-}^{k+1}\texttt{0}^{l}\to\alpha\texttt{-+}^k\texttt{0}^{l+1}$ and $\alpha\texttt{+-}^k\texttt{0}^{l+1}\to\alpha\texttt{0+}^{k+1}\texttt{0}^{l}$.

(d) Let the jth sign of α_i be $(-1)^{a_{ij}}$, and let it be in position b_{ij}. Then we have $(-1)^{a_{ij}+b_{i(j-1)}}=(-1)^{a_{(i+1)j}+b_{(i+1)(j-1)}}$ for $0\le i<k$ and $1\le j\le t$, if we let $b_{i0}=0$.

(e) By parts (a), (b), and (c), α belongs to some chain $\alpha_0\to\cdots\to\alpha_k$, where α_k is final (has no successor) and α_0 is initial (has no predecessor). By part (d), every such chain has at most $\binom{s+t}{t}$ elements. But there are 2^s final strings, by (a), and there are $2^s\binom{s+t}{t}$ strings with s signs and t zeros; so k must be $\binom{s+t}{t}-1$.

Reference: *SICOMP* **2** (1973), 128–133.

30. Assume that $t>0$. Initial strings are the negatives of final strings. Let σ_j be the initial string $\texttt{0}^t\texttt{-}\tau_j$ for $0\le j<2^{s-1}$, where the kth character of τ_j for $1\le k<s$ is $(-1)^{a_k}$ when j is the binary number $(a_{s-1}\ldots a_1)_2$; thus $\sigma_0=\texttt{0}^t\texttt{-++}\ldots\texttt{+}$, $\sigma_1=\texttt{0}^t\texttt{--+}\ldots\texttt{+}$, $\ldots$, $\sigma_{2^{s-1}-1}=\texttt{0}^t\texttt{---}\ldots\texttt{-}$. Let ρ_j be the final string obtained by inserting $\texttt{-0}^t$ after the first run of minus signs in τ_j; thus $\rho_0=\texttt{-0}^t\texttt{++}\ldots\texttt{+}$, $\rho_1=\texttt{--0}^t\texttt{+}\ldots\texttt{+}$, $\ldots$, $\rho_{2^{s-1}-1}=\texttt{--}\ldots\texttt{-0}^t$. We also let $\sigma_{2^{s-1}}=\sigma_0$ and $\rho_{2^{s-1}}=\rho_0$. Then we can prove by induction that the chain beginning with σ_j ends with ρ_j when t is even, with ρ_{j-1} when t is odd, for $1\le j\le 2^{s-1}$. Therefore the chain beginning with $-\rho_j$ ends with $-\sigma_j$ or $-\sigma_{j+1}$.

Let $A_j(s,t)$ be the sequence of (s,t)-combinations derived by mapping the chain that starts with σ_j, and let $B_j(s,t)$ be the analogous sequence derived from $-\rho_j$. Then, for $1\le j\le 2^{s-1}$, the reverse sequence $A_j(s,t)^R$ is $B_j(s,t)$ when t is even, $B_{j-1}(s,t)$ when t is odd. The corresponding recurrences when $st>0$ are

$$A_j(s,t)=\begin{cases}1A_j(s,t-1),\ 0A_{\lfloor(2^{s-1}-1-j)/2\rfloor}(s-1,t)^R, & \text{if } t \text{ is even;}\\ 1A_j(s,t-1),\ 0A_{\lfloor j/2\rfloor}(s-1,t), & \text{if } t \text{ is even;}\end{cases}$$

and when $st>0$ all 2^{s-1} of these sequences are distinct.

Chase's sequence C_{st} is $A_{\lfloor 2^s/3\rfloor}(s,t)$, and $\widehat{C}_{st}$ is $A_{\lfloor 2^{s-1}/3\rfloor}(s,t)$. Incidentally, the homogeneous sequence K_{st} of (31) is $A_{2^{s-1}-[t\text{ even}]}(s,t)^R$.

31. (a) $2^{\binom{s+t}{t}-1}$ solves the recurrence $f(s,t)=2f(s-1,t)f(s,t-1)$ when $f(s,0)=f(0,t)=1$. (b) Now $f(s,t)=(s+1)!\,f(s,t-1)\ldots f(0,t-1)$ has the solution

$$(s+1)!^t\,s!^{\binom{t}{2}}(s-1)!^{\binom{t+1}{3}}\ldots 2!^{\binom{s+t-2}{s}}\;=\;\prod_{r=1}^{s}(r+1)!^{\binom{s+t-1-r}{t-2}+[r=s]}.$$

32. (a) No simple formula seems to exist, but the listings can be counted for small s and t by systematically computing the number of genlex paths that run through all weight-t strings from a given starting point to a given ending point via revolving-door

moves. The totals for $s+t \le 6$ are

$$\begin{array}{ccccccccccccc}
 & & & & & & 1 & & & & & & \\
 & & & & & 1 & & 1 & & & & & \\
 & & & & 1 & & 2 & & 1 & & & & \\
 & & & 1 & & 4 & & 4 & & 4 & & & \\
 & & 1 & & 8 & & 20 & & 8 & & 1 & & \\
 & 1 & & 16 & & 160 & & 160 & & 16 & & 1 & \\
1 & & 32 & & 2264 & & 17152 & & 2264 & & 32 & & 1
\end{array}$$

and $f(4,4) = 95{,}304{,}112{,}865{,}280$; $f(5,5) \approx 5.92646 \times 10^{48}$. [This class of combination generators was first studied by G. Ehrlich, *JACM* **20** (1973), 500–513, but he did not attempt to enumerate them.]

(b) By extending the proof of Theorem N, one can show that all such listings or their reversals must run from 1^t0^s to $0^a1^t0^{s-a}$ for some a, $1 \le a \le s$. Moreover, the number n_{sta} of possibilities, given s, t, and a, satisfies

$$n_{sta} = \begin{cases} n_{s(t-1)1}\, n_{(s-1)t(a-1)}, & \text{if } a > 1, \\ n_{s(t-1)2}\, n_{(s-1)t1} + \cdots + n_{s(t-1)s}\, n_{(s-1)t(s-1)}, & \text{if } a = 1 < s, \end{cases}$$

when $st > 0$. This recurrence has the remarkable solution $n_{sta} = 2^{m(s,t,a)}$, where

$$m(s,t,a) = \begin{cases} \binom{s+t-3}{t} + \binom{s+t-5}{t-2} + \cdots + \binom{s-1}{2}, & \text{if } t \text{ is even;} \\ \binom{s+t-3}{t} + \binom{s+t-5}{t-2} + \cdots + \binom{s}{3} + s - a - [a<s], & \text{if } t \text{ is odd.} \end{cases}$$

33. Consider first the case $t = 1$: The number of near-perfect paths from i to $j > i$ is $f\big(j-i-[i>0]-[j<n-1]\big)$, where $\sum_j f(j)z^j = 1/(1-z-z^3)$. (By coincidence, the same sequence $f(j)$ arises in Caron's polyphase merge on 6 tapes, Table 5.4.2–2.) The sum over $0 \le i < j < n$ is $3f(n) + f(n-1) + f(n-2) + 2 - n$; and we must double this, to cover cases with $j > i$.

When $t > 1$ we can construct $\binom{n}{t} \times \binom{n}{t}$ matrices that tell how many genlex listings begin and end with particular combinations. The entries of these matrices are sums of products of matrices for the case $t-1$, summed over all paths of the type considered for $t = 1$. The totals for $s+t \le 6$ turn out to be

$$\begin{array}{ccccccccccc}
 & & & & 1 & & & & & & \\
 & & & 1 & & 1 & & & & & \\
 & & 1 & & 2 & & 1 & & & & \\
 & 1 & & 6 & & 2 & & 1 & & & \\
1 & & 12 & & 10 & & 2 & & 1 & & \\
 & 20 & & 44 & & 10 & & 2 & & 1 & \\
34 & & 238 & & 68 & & 10 & & 2 & & 1
\end{array}
\qquad
\begin{array}{ccccccccccc}
 & & & & 1 & & & & & & \\
 & & & 1 & & 1 & & & & & \\
 & & 1 & & 2 & & 1 & & & & \\
 & 1 & & 2 & & 0 & & 1 & & & \\
1 & & 2 & & 2 & & 0 & & 1 & & \\
 & 1 & & 2 & & 0 & & 0 & & 1 & \\
1 & & 2 & & 6 & & 0 & & 0 & & 1
\end{array}$$

where the right-hand triangle shows the number of *cycles*, $g(s,t)$. Further values include $f(4,4) = 17736$; $f(5,5) = 9{,}900{,}888{,}879{,}984$; $g(4,4) = 96$; $g(5,5) = 30{,}961{,}456{,}320$.

There are exactly 10 such schemes when $s = 2$ and $n \ge 4$. For example, when $n = 7$ they run from 43210 to 65431 or 65432, or from 54321 to 65420 or 65430 or 65432, or the reverse.

34. The minimum can be computed as in the previous answer, but using min-plus matrix multiplication $c_{ij} = \min_k(a_{ik} + b_{kj})$ instead of ordinary matrix multiplication $c_{ij} = \sum_k a_{ik}b_{kj}$. (When $s = t = 5$, the genlex path in Fig. 26(e) with only 49 imperfect transitions is essentially unique. There is a genlex cycle for $s = t = 5$ that has only 55 imperfections.)

35. From the recurrences (35) we have $a_{st} = b_{s(t-1)} + [s>1][t>0] + a_{(s-1)t}$, $b_{st} = a_{s(t-1)} + a_{(s-1)t}$; consequently $a_{st} = b_{st} + [s>1][t\,\text{odd}]$ and $a_{st} = a_{s(t-1)} + a_{(s-1)t} + [s>1][t\,\text{odd}]$. The solution is

$$a_{st} = \sum_{k=0}^{t/2} \binom{s+t-2-2k}{s-2} - [s>1][t\,\text{even}];$$

this sum is approximately $s/(s+2t)$ times $\binom{s+t}{t}$.

36. Consider the binary tree with root node (s,t) and with recursively defined subtrees rooted at $(s-1,t)$ and $(s,t-1)$ whenever $st>0$; the node (s,t) is a leaf if $st=0$. Then the subtree rooted at (s,t) has $\binom{s+t}{t}$ leaves, corresponding to all (s,t)-combinations $a_{n-1}\ldots a_1a_0$. Nodes on level l correspond to prefixes $a_{n-1}\ldots a_{n-l}$, and leaves on level l are combinations with $r=n-l$.

Any genlex algorithm for combinations $a_{n-1}\ldots a_1a_0$ corresponds to preorder traversal of such a tree, after the children of the $\binom{s+t}{t}-1$ branch nodes have been ordered in any desired way; that, in fact, is why there are $2^{\binom{s+t}{t}-1}$ such genlex schemes (exercise 31(a)). And the operation $j\leftarrow j+1$ is performed exactly once per branch node, namely after both children have been processed.

Incidentally, exercise 7.2.1.2–6(a) implies that the average value of r is $s/(t+1)+t/(s+1)$, which can be $\Omega(n)$; thus the extra time needed to keep track of r is worthwhile.

37. (a) In the lexicographic case we needn't maintain the w_j table, since a_j is active for $j\ge r$ if and only if $a_j=0$. After setting $a_j\leftarrow 1$ and $a_{j-1}\leftarrow 0$ there are two cases to consider if $j>1$: If $r=j$, set $r\leftarrow j-1$; otherwise set $a_{j-2}\ldots a_0\leftarrow 0^r1^{j-1-r}$ and $r\leftarrow j-1-r$ (or $r\leftarrow j$ if r was $j-1$).

(b) Now the transitions to be handled when $j>1$ are to change $a_j\ldots a_0$ as follows: $01^r\to 1101^{r-2}$, $010^r\to 10^{r+1}$, $010^a1^r\to 110^{a+1}1^{r-1}$, $10^r\to 010^{r-1}$, $110^r\to 010^{r-1}1$, $10^a1^r\to 0^a1^{r+1}$; these six cases are easily distinguished. The value of r should change appropriately.

(c) Again the case $j=1$ is trivial. Otherwise $01^a0^r\to 101^{a-1}0^r$; $0^a1^r\to 10^a1^{r-1}$; $101^a0^r\to 01^{a+1}0^r$; $10^a1^r\to 0^a1^{r+1}$; and there is also an ambiguous case, which can occur only if $a_{n-1}\ldots a_{j+1}$ contains at least one 0: Let $k>j$ be minimal with $a_k=0$. Then $10^r\to 010^{r-1}$ if k is odd, $10^r\to 0^r1$ if k is even.

38. The same algorithm works, except that (i) step C1 sets $a_{n-1}\ldots a_0\leftarrow 01^t0^{s-1}$ if n is odd or $s=1$, $a_{n-1}\ldots a_0\leftarrow 001^t0^{s-2}$ if n is even and $s>1$, with an appropriate value of r; (ii) step C3 interchanges the roles of even and odd; (iii) step C5 goes to C4 also if $j=1$.

39. In general, start with $r\leftarrow 0$, $j\leftarrow s+t-1$, and repeat the following steps until $st=0$:

$$r\leftarrow r+[w_j=0]\binom{s+t-a_j}{s-a_j},\quad s\leftarrow s-[a_j=0],\quad t\leftarrow t-[a_j=1],\quad j\leftarrow j-1.$$

Then r is the rank of $a_{n-1}\ldots a_1a_0$. So the rank of 11001001000011111101101010 is $\binom{25}{11}+\binom{24}{11}+\binom{20}{10}+\binom{19}{9}+\binom{18}{7}+\binom{15}{6}+\binom{7}{4}+\binom{6}{2}+\binom{5}{2}+\binom{2}{2}=2390131$.

40. We start with $N\leftarrow 999999$, $v\leftarrow 0$, and repeat the following steps until $st=0$: If $v=0$, set $t\leftarrow t-1$ and $a_{s+t}\leftarrow 1$ if $N<\binom{s+t-1}{s}$, otherwise set $N\leftarrow N-\binom{s+t-1}{s}$, $v\leftarrow (s+t) \bmod 2$, $s\leftarrow s-1$, $a_{s+t}\leftarrow 0$. If $v=1$, set $v\leftarrow (s+t) \bmod 2$, $s\leftarrow s-1$, and $a_{s+t}\leftarrow 0$ if $N<\binom{s+t-1}{t}$, otherwise set $N\leftarrow N-\binom{s+t-1}{t}$, $t\leftarrow t-1$, $a_{s+t}\leftarrow 1$.

Finally if $s = 0$, set $a_{t-1} \ldots a_0 \leftarrow 1^t$; if $t = 0$, set $a_{s-1} \ldots a_0 \leftarrow 0^s$. The answer is $a_{25} \ldots a_0 = 11101001111111010100100001$.

41. Let $c(0), \ldots, c(2^n - 1) = C_n$ where $C_{2n} = 0C_{2n-1}$, $1C_{2n-1}$; $C_{2n+1} = 0C_{2n}$, $1\widehat{C}_{2n}$; $\widehat{C}_{2n} = 1C_{2n-1}$, $0\widehat{C}_{2n-1}$; $\widehat{C}_{2n+1} = 1\widehat{C}_{2n}$, $0\widehat{C}_{2n}$; $C_0 = \widehat{C}_0 = \epsilon$. Then $a_j \oplus b_j = b_{j+1} \wedge (b_{j+2} \vee (b_{j+3} \wedge (b_{j+4} \vee \cdots)))$ if j is even, $b_{j+1} \vee (b_{j+2} \wedge (b_{j+3} \vee (b_{j+4} \wedge \cdots)))$ if j is odd. Curiously we also have the inverse relation $c\big((\ldots \bar{a}_5 a_4 \bar{a}_3 a_2 \bar{a}_1 a_0)_2\big) = (\ldots \bar{b}_5 b_4 \bar{b}_3 b_2 \bar{b}_1 b_0)$.

42. Equation (40) shows that the left context $a_{n-1} \ldots a_{l+1}$ does not affect the behavior of the algorithm on $a_{l-1} \ldots a_0$ if $a_l = 0$ and $l > r$. Therefore we can analyze Algorithm C by counting combinations that end with certain bit patterns, and it follows that the number of times each operation is performed can be represented as $[w^s z^t]\, p(w,z)/(1-w^2)^2(1-z^2)^2(1-w-z)$ for an appropriate polynomial $p(w,z)$.

For example, the algorithm goes from C5 to C4 once for each combination that ends with $01^{2a+1}01^{2b+1}$ or has the form $1^{a+1}01^{2b+1}$, for integers $a, b \ge 0$; the corresponding generating functions are $w^2z^2/(1-z^2)^2(1-w-z)$ and $w(z^2+z^3)/(1-z^2)^2$.

Here are the polynomials $p(w,z)$ for key operations. Let $W = 1 - w^2$, $Z = 1 - z^2$.

C3 → C4:	$wzW(1+wz)(1-w-z^2)$:	C5($r \leftarrow 1$):	$w^2zW^2Z(1-wz-z^2)$:
C3 → C5:	$wzW(w+z)(1-wz-z^2)$:	C5($r \leftarrow j-1$):	$w^2z^3W^2(1-wz-z^2)$:
C3 → C6:	$w^2z^2W(w+z)$:	C6($j = 1$):	w^2zW^2Z:
C3 → C7:	$w^2zW(1+wz)$:	C6($r \leftarrow j-1$):	$w^2z^3W^2$:
C4($j = 1$):	$wzW^2Z(1-w-z^2)$:	C6($r \leftarrow j$):	w^3z^2WZ:
C4($r \leftarrow j-1$):	$w^3zWZ(1-w-z^2)$:	C7 → C6:	w^2zW^2:
C4($r \leftarrow j$):	$wz^2W^2(1+z-2wz-z^2-z^3)$:	C7($r \leftarrow j$):	w^4zWZ:
C5 → C4:	$wz^2W^2(1-wz-z^2)$:	C7($r \leftarrow j-2$):	$w^3z^2W^2$:
C5($r \leftarrow j-2$):	$w^4zWZ(1-wz-z^2)$:		

The asymptotic value is $\binom{s+t}{t}\big(p(1-x,x)/(2x-x^2)^2(1-x^2)^2 + O(n^{-1})\big)$, for fixed $0 < x < 1$, if $t = xn + O(1)$ as $n \to \infty$. Thus we find, for example, that the four-way branching in step C3 takes place with relative frequencies $x + x^2 - x^3 : 1 : x : 1 + x - x^2$.

Incidentally, the number of cases with j odd exceeds the number of cases with j even by

$$\sum_{k,l \ge 1} \binom{s+t-2k-2l}{s-2k} [2k+2l \le s+t] + [s \text{ odd}][t \text{ odd}],$$

in *any* genlex scheme that uses (39). This quantity has the interesting generating function $wz/(1+w)(1+z)(1-w-z)$.

43. The identity is true for all nonnegative integers x, except when $x = 1$.

44. In fact, $C_t(n) - 1 = \widehat{C}_t(n-1)^R$, and $\widehat{C}_t(n) - 1 = C_t(n-1)^R$. (Hence $C_t(n) - 2 = C_t(n-2)$, etc.)

45. In the following algorithm, r is the least subscript with $c_r \ge r$.

CC1. [Initialize.] Set $c_j \leftarrow n - t - 1 + j$ and $z_j \leftarrow 0$ for $1 \le j \le t+1$. Also set $r \leftarrow 1$. (We assume that $0 < t < n$.)

CC2. [Visit.] Visit the combination $c_t \ldots c_2 c_1$. Then set $j \leftarrow r$.

CC3. [Branch.] Go to CC5 if $z_j \ne 0$.

CC4. [Try to decrease c_j.] Set $x \leftarrow c_j + (c_j \bmod 2) - 2$. If $x \ge j$, set $c_j \leftarrow x$, $r \leftarrow 1$; otherwise if $c_j = j$, set $c_j \leftarrow j - 1$, $z_j \leftarrow c_{j+1} - \big((c_{j+1} + 1) \bmod 2\big)$, $r \leftarrow j$; otherwise if $c_j < j$, set $c_j \leftarrow j$, $z_j \leftarrow c_{j+1} - \big((c_{j+1} + 1) \bmod 2\big)$, $r \leftarrow \max(1, j-1)$; otherwise set $c_j \leftarrow x$, $r \leftarrow j$. Return to CC2.

CC5. [Try to increase c_j.] Set $x \leftarrow c_j + 2$. If $x < z_j$, set $c_j \leftarrow x$; otherwise if $x = z_j$ and $z_j \neq 0$, set $c_j \leftarrow x - (c_{j+1} \bmod 2)$; otherwise set $z_j \leftarrow 0$, $j \leftarrow j + 1$, and go to CC3 (but terminate if $j > t$). If $c_1 > 0$, set $r \leftarrow 1$; otherwise set $r \leftarrow j - 1$. Return to CC2. ▮

46. Eq. (40) implies that $u_k = (b_j + k + 1) \bmod 2$ when j is minimal with $b_j > k$. Then (37) and (38) yield the following algorithm, where we assume for convenience that $3 \le s < n$.

CB1. [Initialize.] Set $b_j \leftarrow j - 1$ for $1 \le j \le s$; also set $z \leftarrow s + 1$, $b_z \leftarrow 1$. (When subsequent steps examine the value of z, it is the smallest index such that $b_z \neq z - 1$.)

CB2. [Visit.] Visit the dual combination $b_s \ldots b_2 b_1$.

CB3. [Branch.] If b_2 is odd: Go to CB4 if $b_2 \neq b_1 + 1$, otherwise to CB5 if $b_1 > 0$, otherwise to CB6 if b_z is odd. Go to CB9 if b_2 is even and $b_1 > 0$. Otherwise go to CB8 if $b_{z+1} = b_z + 1$, otherwise to CB7.

CB4. [Increase b_1.] Set $b_1 \leftarrow b_1 + 1$ and return to CB2.

CB5. [Slide b_1 and b_2.] If b_3 is odd, set $b_1 \leftarrow b_1 + 1$ and $b_2 \leftarrow b_2 + 1$; otherwise set $b_1 \leftarrow b_1 - 1$, $b_2 \leftarrow b_2 - 1$, $z \leftarrow 3$. Go to CB2.

CB6. [Slide left.] If z is odd, set $z \leftarrow z - 2$, $b_{z+1} \leftarrow z + 1$, $b_z \leftarrow z$; otherwise set $z \leftarrow z - 1$, $b_z \leftarrow z$. Go to CB2.

CB7. [Slide b_z.] If b_{z+1} is odd, set $b_z \leftarrow b_z + 1$ and terminate if $b_z \ge n$; otherwise set $b_z \leftarrow b_z - 1$, then if $b_z < z$ set $z \leftarrow z + 1$. To CB2.

CB8. [Slide b_z and b_{z+1}.] If b_{z+2} is odd, set $b_z \leftarrow b_{z+1}$, $b_{z+1} \leftarrow b_z + 1$, and terminate if $b_{z+1} \ge n$. Otherwise set $b_{z+1} \leftarrow b_z$, $b_z \leftarrow b_z - 1$, then if $b_z < z$ set $z \leftarrow z + 2$. To CB2.

CB9. [Decrease b_1.] Set $b_1 \leftarrow b_1 - 1$, $z \leftarrow 2$, and return to CB2. ▮

Notice that this algorithm is *loopless*. Chase gave a similar procedure for the sequence $\widehat{C}^R_{st}$ in *Cong. Num.* **69** (1989), 233–237. It is truly amazing that this algorithm defines precisely the complements of the indices $c_t \ldots c_1$ produced by the algorithm in the previous exercise.

47. We can, for example, use Algorithm C and its reverse (exercise 38), with w_j replaced by a d-bit number whose bits represent activity at different levels of the recursion. Separate pointers r_0, r_1, ..., r_{d-1} are needed to keep track of the r-values on each level. (Many other solutions are possible.)

48. There are permutations $\pi_1, \ldots, \pi_M$ such that the kth element of Λ_j is $\pi_k \alpha_j \uparrow \beta_{k-1}$. And $\pi_k \alpha_j$ runs through all permutations of $\{s_1 \cdot 1, \ldots, s_d \cdot d\}$ as j varies from 0 to $N - 1$.

Historical note: The first publication of a homogeneous revolving-door scheme for (s,t)-combinations was by Éva Török, *Matematikai Lapok* **19** (1968), 143–146, who was motivated by the generation of multiset permutations. Many authors have subsequently relied on the homogeneity condition for similar constructions, but this exercise shows that homogeneity is not necessary.

49. We have $\lim_{z\to q}(z^{km+r} - 1)/(z^{lm+r} - 1) = 1$ when $0 < r < m$, and the limit is $\lim_{z\to q}(kmz^{km-1})/(lmz^{lm-1}) = k/l$ when $r = 0$. So we can pair up factors of the numerator $\prod_{n-k<a\le n}(z^a - 1)$ with factors of the denominator $\prod_{0<b\le k}(z^b - 1)$ when $a \equiv b$ (modulo m).

Notes: In the special case $m = 2$, $q = -1$, the second factor vanishes only when n is even and k is odd. The formula $\binom{n}{k}_q = \binom{n}{n-k}_q$ holds for all $n \ge 0$, but $\binom{\lfloor n/m \rfloor}{\lfloor k/m \rfloor}$ is *not*

always equal to $\binom{\lfloor n/m\rfloor}{\lfloor (n-k)/m\rfloor}$. We do, however, have $\lfloor k/m\rfloor + \lfloor (n-k)/m\rfloor = \lfloor n/m\rfloor$ in the case when $n \bmod m \geq k \bmod m$; otherwise the second factor is zero.

50. The stated coefficient is zero when $n_1 \bmod m + \cdots + n_t \bmod m \geq m$. Otherwise it equals

$$\binom{\lfloor (n_1+\cdots+n_t)/m\rfloor}{\lfloor n_1/m\rfloor,\ldots,\lfloor n_t/m\rfloor}\binom{(n_1+\cdots+n_t) \bmod m}{n_1 \bmod m,\ldots,n_t \bmod m}_q,$$

by Eq. 1.2.6–(43); here each upper index is the sum of the lower indices.

51. All paths clearly run between 000111 and 111000, since those vertices have degree 1. Fourteen total paths reduce to four under the stated equivalences. The path in (50), which is equivalent to itself under reflection-and-reversal, can be described by the delta sequence $A = 3452132523414354123$; the other three classes are $B = 3452541453414512543$, $C = 3452541453252154123$, $D = 3452134145341432543$. D. H. Lehmer found path C [*AMM* **72** (1965), part II, 36–46]; D is essentially the path constructed by Eades, Hickey, and Read.

(Incidentally, perfect schemes aren't really rare, although they seem to be difficult to construct systematically. The case $s = 3$, $t = 5$ has 4,050,046 of them.)

52. We may assume that each s_j is nonzero and that $d > 1$. Then the difference between permutations with an even and odd number of inversions is $\binom{\lfloor (s_0+\cdots+s_d)/2\rfloor}{\lfloor s_0/2\rfloor,\ldots,\lfloor s_d/2\rfloor} \geq 2$, by exercise 50, unless at least two of the multiplicities s_j are odd.

Conversely, if at least two multiplicities are odd, a general construction by G. Stachowiak [*SIAM J. Discrete Math.* **5** (1992), 199–206] shows that a perfect scheme exists. Indeed, his construction applies to a variety of topological sorting problems; in the special case of multisets it gives a Hamiltonian circuit in all cases with $d > 1$ and $s_0 s_1$ odd, except when $d = 2$, $s_0 = s_1 = 1$, and s_2 is even.

53. See *AMM* **72** (1965), Part II, 36–46.

54. Assuming that $st \neq 0$, a Hamiltonian path exists if and only if s and t are not both even; a Hamiltonian circuit exists if and only if, in addition, ($s \neq 2$ and $t \neq 2$) or $n = 5$. [T. C. Enns, *Discrete Math.* **122** (1993), 153–165.]

55. [*Discrete Math.* **48** (1984), 163–171.] This problem is equivalent to the "middle levels conjecture," which states that there is a Gray path through all binary strings of length $2t-1$ and weights $\{t-1, t\}$. In fact, such strings can almost certainly be generated by a delta sequence of the special form $\alpha_0\alpha_1\ldots\alpha_{2t-2}$ where the elements of α_k are those of α_0 shifted by k, modulo $2t-1$. For example, when $t = 3$ we can start with $a_5a_4a_3a_2a_1a_0 = 000111$ and repeatedly swap $a_0 \leftrightarrow a_\delta$, where δ runs through the cycle (4134 5245 1351 2412 3523). The middle levels conjecture is known to be true for $t \leq 15$ [see I. Shields and C. D. Savage, *Cong. Num.* **140** (1999), 161–178].

56. Yes; there is a near-perfect genlex solution for all m, n, and t when $n \geq m > t$. One such scheme, in bitstring notation, is $1A_{(m-t)(t-1)}0^{n-m}$, $01A_{(m-t)(t-1)}0^{n-m-1}$, $\ldots$, $0^{n-m}1A_{(m-t)(t-1)}$, $0^{n-m+1}1A_{(m-1-t)(t-1)}$, $\ldots$, $0^{n-t}1A_{0(t-1)}$, using the sequences A_{st} of (35).

57. Solve the previous problem with m and n reduced by $t-1$, then add $j-1$ to each c_j. (Case (a), which is particularly simple, was probably known to Czerny.)

58. The generating function $G_{mnt}(q) = \sum g_{mntk}q^k$ for the number g_{mntk} of chords reachable in k steps from $0^{n-t}1^t$ satisfies $G_{mmt}(q) = \binom{m}{t}_q$ and $G_{m(n+1)t}(q) = G_{mnt}(q) + q^{tn-(t-1)m}\binom{m-1}{t-1}_q$, because the latter term accounts for cases with $c_t = n$ and $c_1 > n-m$. A perfect scheme is possible only if $|G_{mnt}(-1)| \leq 1$. But if $n \geq m > t \geq 2$, this

condition holds only when $m = t + 1$ or $(n - t)t$ is odd, by exercise 49. So there is no perfect solution when $t = 4$ and $m > 5$. (Many chords have only two neighbors when $n = t + 2$, so one can easily rule out that case. All cases with $n \geq m > 5$ and $t = 3$ apparently do have perfect paths when n is even.)

59. The following solution uses lexicographic order, taking care to ensure that the average amount of computation per step is bounded. We may assume that $stm_s \ldots m_0 \neq 0$ and $t \leq m_s + \cdots + m_1 + m_0$.

Q1. [Initialize.] Set $q_j \leftarrow 0$ for $1 \leq j \leq s$, and $x = t$.

Q2. [Distribute.] Set $j \leftarrow 0$. Then while $x > m_j$, set $q_j \leftarrow m_j$, $x \leftarrow x - m_j$, $j \leftarrow j + 1$, and repeat until $x \leq m_j$. Finally set $q_j \leftarrow x$.

Q3. [Visit.] Visit the bounded composition $q_s \ldots q_1 q_0$.

Q4. [Pick up the rightmost units.] If $j = 0$, set $x \leftarrow q_0 - 1$, $j \leftarrow 1$. Otherwise if $q_0 = 0$, set $x \leftarrow q_j - 1$, $q_j \leftarrow 0$, and $j \leftarrow j + 1$. Otherwise go to Q7.

Q5. [Full?] Terminate if $j > s$. Otherwise if $q_j = m_j$, set $x \leftarrow x + m_j$, $q_j \leftarrow 0$, $j \leftarrow j + 1$, and repeat this step.

Q6. [Increase q_j.] Set $q_j \leftarrow q_j + 1$. Then if $x = 0$, set $q_0 \leftarrow 0$ and return to Q3. Otherwise go to Q2.

Q7. [Increase and decrease.] While $q_j = m_j$, set $j \leftarrow j+1$ and repeat until $q_j < m_j$ (but terminate if $j > s$). Then set $q_j \leftarrow q_j + 1$, $j \leftarrow j - 1$, $q_j \leftarrow q_j - 1$. If $q_0 = 1$, set $j \leftarrow 1$. Return to Q3. ∎

For example, if $m_s = \cdots = m_0 = 9$, the successors of the composition $3{+}9{+}9{+}7{+}0{+}0$ are $4{+}0{+}0{+}6{+}9{+}9$, $4{+}0{+}0{+}7{+}8{+}9$, $4{+}0{+}0{+}7{+}9{+}8$, $4{+}0{+}0{+}8{+}7{+}9$, ...

60. Let $F_s(t) = \emptyset$ if $t < 0$ or $t > m_s + \cdots + m_0$; otherwise let $F_0(t) = t$, and

$$F_s(t) = 0{+}F_{s-1}(t),\ 1{+}F_{s-1}(t-1)^R,\ 2{+}F_{s-1}(t-2),\ \ldots,\ m_s{+}F_{s-1}(t-m_s)^{R^{m_s}}$$

when $s > 0$. This sequence can be shown to have the required properties; it is, in fact, equivalent to the compositions defined by the homogeneous sequence K_{st} of (31) under the correspondence of exercise 4, when restricted to the subsequence defined by the bounds $m_s, \ldots, m_0$. [See T. Walsh, *J. Combinatorial Math. and Combinatorial Computing* **33** (2000), 323–345, who has implemented it looplessly.]

61. (a) A $2 \times n$ contingency table with row sums r and $c_1 + \cdots + c_n - r$ is equivalent to solving $r = a_1 + \cdots + a_n$ with $0 \leq a_1 \leq c_1, \ldots, 0 \leq a_n \leq c_n$.

(b) We can compute it sequentially by setting $a_{ij} \leftarrow \min(r_i - a_{i1} - \cdots - a_{i(j-1)}, c_j - a_{1j} - \cdots - a_{(i-1)j})$ for $j = 1, \ldots, n$, for $i = 1, \ldots, m$. Alternatively, if $r_1 \leq c_1$, set $a_{11} \leftarrow r_1$, $a_{12} \leftarrow \cdots \leftarrow a_{1n} \leftarrow 0$, and do the remaining rows with c_1 decreased by r_1; if $r_1 > c_1$, set $a_{11} \leftarrow c_1$, $a_{21} \leftarrow \cdots \leftarrow a_{m1} \leftarrow 0$, and do the remaining columns with r_1 decreased by c_1. The second approach shows that at most $m + n - 1$ of the entries are nonzero. We can also write down the explicit formula

$$a_{ij} = \max\bigl(0, \min(r_i, c_j, r_1 + \cdots + r_i - c_1 - \cdots - c_{j-1}, c_1 + \cdots + c_j - r_1 - \cdots - r_{i-1})\bigr).$$

(c) The same matrix is obtained as in (b).

(d) Reverse left and right in (b) and (c), obtaining

$$a_{ij} = \max\bigl(0, \min(r_i, c_j, r_{i+1} + \cdots + r_m - c_1 - \cdots - c_{j-1}, c_1 + \cdots + c_j - r_i - \cdots - r_m)\bigr)$$

in both cases.

(e) Here we choose, say, row-wise order: Generate the first row just as for bounded compositions of r_1, with bounds $(c_1,\ldots,c_n)$; and for each row $(a_{11},\ldots,a_{1n})$, generate the remaining rows recursively in the same way, but with the column sums $(c_1-a_{11},\ldots,c_n-a_{1n})$. Most of the action takes place on the bottom two rows, but when a change is made to an earlier row the later rows must be re-initialized.

62. If a_{ij} and a_{kl} are positive, we obtain another contingency table by setting $a_{ij}\leftarrow a_{ij}-1$, $a_{il}\leftarrow a_{il}+1$, $a_{kj}\leftarrow a_{kj}+1$, $a_{kl}\leftarrow a_{kl}-1$. We want to show that the graph G whose vertices are the contingency tables for $(r_1,\ldots,r_m;c_1,\ldots,c_n)$, adjacent if they can be obtained from each other by such a transformation, has a Hamiltonian path.

When $m=n=2$, G is a simple path. When $m=2$ and $n=3$, G has a two-dimensional structure from which we can see that every vertex is the starting point of at least two Hamiltonian paths, having distinct endpoints. When $m=2$ and $n\ge4$ we can show, inductively, that G actually has Hamiltonian paths from any vertex to any other.

When $m\ge3$ and $n\ge3$, we can reduce the problem from m to $m-1$ as in answer 61(e), if we are careful not to "paint ourselves into a corner." Namely, we must avoid reaching a state where the nonzero entries of the bottom two rows have the form $\left(\begin{smallmatrix}1&a&0\\0&b&c\end{smallmatrix}\right)$ for some a, b, $c>0$ and a change to row $m-2$ forces this to become $\left(\begin{smallmatrix}0&a&1\\0&b&c\end{smallmatrix}\right)$. The previous round of changes to rows $m-1$ and m can avoid such a trap unless $c=1$ and it begins with $\left(\begin{smallmatrix}0&a+1&0\\1&b-1&1\end{smallmatrix}\right)$ or $\left(\begin{smallmatrix}1&a-1&1\\0&b+1&0\end{smallmatrix}\right)$. But that situation can be avoided too.

(A genlex method based on exercise 60 would be considerably simpler, and it almost always would make only four changes per step. But it would occasionally need to update $2\min(m,n)$ entries at a time.)

63. When $x_1\ldots x_s$ is a binary string and A is a list of subcubes, let $A\oplus x_1\ldots x_s$ denote replacing the digits $(a_1,\ldots,a_s)$ in each subcube of A by $(a_1\oplus x_1,\ldots,a_s\oplus x_s)$, from left to right. For example, $0{*}1{*}{*}10\oplus1010=1{*}1{*}{*}00$. Then the following mutual recursions define a Gray code, because A_{st} gives a Gray path from $0^s{*}^t$ to $10^{s0}1{*}^t$ and B_{st} gives a Gray path from $0^s{*}^t$ to ${*}01^{s-1}{*}^{t-1}$, when $st>0$:

$$\begin{aligned}A_{st}&=0B_{(s-1)t},\ {*}A_{s(t-1)}\oplus001^{s-2},\ 1B^R_{(s-1)t};\\B_{st}&=0A_{(s-1)t},\ 1B_{(s-1)t}\oplus010^{s-2},\ 1A_{s(t-1)}\oplus1^s.\end{aligned}$$

The strings 001^{s-2} and 010^{s-2} are simply 0^s when $s<2$; A_{s0} is Gray binary code; $A_{0t}=B_{0t}={*}^t$.

Incidentally, the somewhat simpler construction

$$G_{st}=\ {*}G_{s(t-1)},\ a_tG_{(s-1)t},\ a_{t-1}G^R_{(s-1)t},\qquad a_t=t \bmod 2,$$

defines a pleasant Gray *path* from ${*}^t0^s$ to $a_{t-1}{*}^t0^{s-1}$.

64. If a path P is considered equivalent to P^R and to $P\oplus x_1\ldots x_s$, the total number can be computed systematically as in exercise 33, with the following results for $s+t\le6$:

paths	cycles
1	1
1 1	1 1
1 2 1	1 1 1
1 3 3 1	1 1 1 1
1 5 10 4 1	1 2 1 1 1
1 6 36 35 5 1	1 2 3 1 1 1
1 9 310 4630 218 6 1	1 3 46 4 1 1 1

In general there are $t+1$ paths when $s=1$ and $\binom{\lceil s/2\rceil+2}{2}-(s \bmod 2)$ when $t=1$. The cycles for $s\le 2$ are unique. When $s=t=5$ there are approximately 6.869×10^{170} paths and 2.495×10^{70} cycles.

65. Let $G(n,0)=\epsilon$; $G(n,t)=\emptyset$ when $n<t$; and for $1\le t\le n$, let $G(n,t)$ be

$$\hat g(0)G(n-1,t),\ \hat g(1)G(n-1,t)^R,\ \ldots,\ \hat g(2^t-1)G(n-1,t)^R,\ \hat g(2^t-1)G(n-1,t-1),$$

where $\hat g(k)$ is a t-bit column containing the Gray binary number $g(k)$ with its least significant bit at the top. In this general formula we implicitly add a row of zeros below the bases of $G(n-1,t-1)$.

This remarkable rule gives ordinary Gray binary code when $t=1$, omitting $0\ldots00$. A cyclic Gray code is impossible because $\binom{n}{t}_2$ is odd.

66. A Gray path for compositions corresponding to Algorithm C implies that there is a path in which all transitions are $0^k1^l\leftrightarrow1^l0^k$ with $\min(k,l)\le2$. Perhaps there is, in fact, a cycle with $\min(k,l)=1$ in each transition.

67. (a) $\{\emptyset\}$; (b) $\emptyset$.

68. The least N with $\kappa_tN<N$ is $\binom{2t-1}{t}+\binom{2t-3}{t-1}+\cdots+\binom{1}{1}+1=\frac12\big(\binom{2t}{t}+\binom{2t-2}{t-1}+\cdots+\binom{0}{0}+1\big)$, because $\binom{n}{t-1}\le\binom{n}{t}$ if and only if $n\ge2t-1$.

69. From the identity

$$\kappa_t\big(\tbinom{2t-3}{t}+N'\big)-\big(\tbinom{2t-3}{t}+N'\big)=\kappa_t\big(\tbinom{2t-2}{t}+N'\big)-\big(\tbinom{2t-2}{t}+N'\big)=\tbinom{2t-2}{t}\tfrac{1}{t-1}+\kappa_{t-1}N'-N'$$

when $N'<\binom{2t-3}{t}$, we conclude that the maximum is $\binom{2t-2}{t}\frac1t+\binom{2t-4}{t-1}\frac{1}{t-2}+\cdots+\binom{2}{2}\frac11$, and it occurs at 2^{t-1} values of N when $t>1$.

70. Let C_t be the t-cliques. The first $\binom{1414}{t}+\binom{1009}{t-1}$ t-combinations visited by Algorithm L define a graph on 1415 vertices with 1000000 edges. If $\|C_t\|$ were larger, $\|\partial^{t-2}C_t\|$ would exceed 1000000. Thus the single graph defined by $P_{(1000000)_2}$ has the maximum number of t-cliques for all $t\ge2$.

71. $M=\binom{m_s}{s}+\cdots+\binom{m_u}{u}$ for $m_s>\cdots>m_u\ge u\ge1$, where $\{m_s,\ldots,m_u\}=\{s+t-1,\ldots,n_v\}\setminus\{n_t,\ldots,n_{v+1}\}$. (Compare with exercise 15, which gives $\binom{s+t}{t}-1-N$.)

If $\alpha=a_{n-1}\ldots a_0$ is the bit string corresponding to the combination $n_t\ldots n_1$, then v is 1 plus the number of trailing 1s in α, and u is the length of the rightmost run of 0s. For example, when $\alpha=1010001111$ we have $s=4$, $t=6$, $M=\binom84+\binom63$, $u=3$, $N=\binom96+\binom75$, $v=5$.

72. A and B are cross-intersecting $\iff\alpha\not\subseteq U\setminus\beta$ for all $\alpha\in A$ and $\beta\in B\iff A\cap\partial^{n-s-t}B^-=\emptyset$, where $B^-=\{U\setminus\beta\mid\beta\in B\}$ is a set of $(n-t)$-combinations. Since $Q^-_{Nnt}=P_{N(n-t)}$, we have $\|\partial^{n-s-t}B^-\|\ge\|\partial^{n-s-t}P_{N(n-t)}\|$, and $\partial^{n-s-t}P_{N(n-t)}=P_{N's}$ where $N'=\kappa_{s+1}\ldots\kappa_{n-t}N$. Thus if A and B are cross-intersecting we have $M+N'\le\|A\|+\|\partial^{n-s-t}B^-\|\le\binom ns$, and $Q_{Mns}\cap P_{N's}=\emptyset$.

Conversely, if $Q_{Mns}\cap P_{N's}\ne\emptyset$ we have $\binom ns<M+N'\le\|A\|+\|\partial^{n-s-t}B^-\|$, so A and B cannot be cross-intersecting.

73. $\|\varrho Q_{Nnt}\|=\kappa_{n-t}N$ (see exercise 93). Also, arguing as in (58) and (59), we find $\varrho P_{N5}=(n-1)P_{N5}\cup\cdots\cup10P_{N5}\cup\{543210,\ldots,987654\}$ in that particular case; and $\|\varrho P_{Nt}\|=(n+1-n_t)N+\binom{n_t+1}{t+1}$ in general.

74. The identity $\binom{n+1}{k}=\binom nk+\binom{n-1}{k-1}+\cdots+\binom{n-k}{0}$, Eq. 1.2.6–(10), gives another representation if $n_v>v$. But (60) is unaffected, since we have $\binom{n+1}{k-1}=\binom{n}{k-1}+\binom{n-1}{k-2}+\cdots+\binom{n-k+1}{0}$.

75. Represent $N+1$ by adding $\binom{v-1}{v-1}$ to (57); then use the previous exercise to deduce that $\kappa_t(N+1)-\kappa_t N=\binom{v-1}{v-2}=v-1$.

76. [D. E. Daykin, *Nanta Math.* **8**, 2 (1975), 78–83.] We work with extended representations $M=\binom{m_t}{t}+\cdots+\binom{m_u}{u}$ and $N=\binom{n_t}{t}+\cdots+\binom{n_v}{v}$ as in exercise 74, calling them *improper* if the final index u or v is zero. Call N *flexible* if it has both proper and improper representations, that is, if $n_v>v>0$.

(a) Given an integer S, find $M+N$ such that $M+N=S$ and $\kappa_t M+\kappa_t N$ is minimum, with M as large as possible. If $N=0$, we're done. Otherwise the max-min operation preserves both $M+N$ and $\kappa_t M+\kappa_t N$, so we can assume that $v\ge u\ge 1$ in the proper representations of M and N. If N is inflexible, $\kappa_t(M+1)+\kappa_t(N-1)=(\kappa_t M+u-1)+(\kappa_t N-v)<\kappa_t M+\kappa_t N$, by exercise 75; therefore N must be flexible. But then we can apply the max-min operation to M and the improper representation of N, increasing M: Contradiction.

This proof shows that equality holds if and only if $MN=0$, a fact that was noted in 1927 by F. S. Macaulay.

(b) Now we try to minimize $\max(\kappa_t M,N)+\kappa_{t-1}N$ when $M+N=S$, this time representing N as $\binom{n_{t-1}}{t-1}+\cdots+\binom{n_v}{v}$. The max-min operation can still be used if $n_{t-1}<m_t$; leaving m_t unchanged, it preserves $M+N$ and $\kappa_t M+\kappa_{t-1}N$ as well as the relation $\kappa_t M>N$. We arrive at a contradiction as in (a) if $N\ne 0$, so we can assume that $n_{t-1}\ge m_t$.

If $n_{t-1}>m_t$ we have $N>\kappa_t M$ and also $\lambda_t N>M$; hence $M+N<\lambda_t N+N=\binom{n_{t-1}+1}{t}+\cdots+\binom{n_v+1}{v}$, and we have $\kappa_t(M+N)\le\kappa_t(\lambda_t N+N)=N+\kappa_{t-1}N$.

Finally if $n_{t-1}=m_t=a$, let $M=\binom{a}{t}+M'$ and $N=\binom{a}{t-1}+N'$. Then $\kappa_t(M+N)=\binom{a+1}{t-1}+\kappa_{t-1}(M'+N')$, $\kappa_t M=\binom{a}{t-1}+\kappa_{t-1}M'$, and $\kappa_{t-1}N=\binom{a}{t-2}+\kappa_{t-2}N'$; the result follows by induction on t.

77. [A. J. W. Hilton, *Periodica Math. Hung.* **10** (1979), 25–30.] Let $M=\|A_1\|$ and $N=\|A_0\|$; we can assume that $t>0$ and $N>0$. Then $\|\partial A\|=\|\partial A_1\cup A_0\|+\|\partial A_0\|\ge\max(\|\partial A_1\|,\|A_0\|)+\|\partial A_0\|\ge\max(\kappa_t M,N)+\kappa_{t-1}N\ge\kappa_t(M+N)=\|P_{\|A\|t}\|$, by induction on $m+n+t$.

Conversely, let $A_1=P_{Mt}+1$ and $A_0=P_{N(t-1)}+1$; this notation means, for example, that $\{210,320\}+1=\{321,431\}$. Then $\kappa_t(M+N)\le\|\partial A\|=\|\partial A_1\cup A_0\|+\|(\partial A_0)0\|=\max(\kappa_t M,N)+\kappa_{t-1}N$, because $\partial A_1=P_{(\kappa_t M)(t-1)}+1$. [Schützenberger observed in 1959 that $\kappa_t(M+N)\le\kappa_t M+\kappa_{t-1}N$ if and only if $\kappa_t M\ge N$.]

For the first inequality, let A and B be disjoint sets of t-combinations with $\|A\|=M$, $\|\partial A\|=\kappa_t M$, $\|B\|=N$, $\|\partial B\|=\kappa_t N$. Then $\kappa_t(M+N)=\kappa_t\|A\cup B\|\le\|\partial(A\cup B)\|=\|\partial A\cup\partial B\|=\|\partial A\|+\|\partial B\|=\kappa_t M+\kappa_t N$.

78. In fact, $\mu_t(M+\lambda_{t-1}M)=M$, and $\mu_t N+\lambda_{t-1}\mu_t N=N+(n_2-n_1)[v=1]$ when N is given by (57).

79. If $N>0$ and $t>1$, represent N as in (57) and let $N=N_0+N_1$, where

$$N_0=\binom{n_t-1}{t}+\cdots+\binom{n_v-1}{v},\qquad N_1=\binom{n_t-1}{t-1}+\cdots+\binom{n_v-1}{v-1}.$$

Let $N_0=\binom{y}{t}$ and $N_1=\binom{z}{t-1}$. Then, by induction on t and $\lfloor x\rfloor$, we have $\binom{x}{t}=N_0+\kappa_t N_0\ge\binom{y}{t}+\binom{y}{t-1}=\binom{y+1}{t}$; $N_1=\binom{x}{t}-\binom{y}{t}\ge\binom{x}{t}-\binom{x-1}{t}=\binom{x-1}{t-1}$; and $\kappa_t N=N_1+\kappa_{t-1}N_1\ge\binom{z}{t-1}+\binom{z}{t-2}=\binom{z+1}{t-1}\ge\binom{x}{t-1}$.

Lovász actually proved a stronger result [*Combinatorial Problems and Exercises*, (Akadémiai Kiadó, 1979), problem 13.31(a)], which was strengthened further by R. M. Redheffer [*AMM* **103** (1996), 62–64]: Suppose $\binom{u}{t}=\binom{v}{t}+\binom{w}{t-1}$ where $u\ge t-1$,

$v \ge t-1$, and $w \ge t-2$. Then for $1 \le k < t$ we have $\binom{u}{k} \le \binom{v}{k} + \binom{w}{k-1}$ if and only if $v \ge w$; for $t < k \le \min(v+1, w+2)$ we have $\binom{u}{k} \le \binom{v}{k} + \binom{w}{k-1}$ if and only if $v \le w$.

80. For example, if the largest element of $\widehat{P}_{N5}$ is 66433, we have

$$\widehat{P}_{N5} = \{00000,\ldots,55555\} \cup \{60000,\ldots,65555\} \cup \{66000,\ldots,66333\} \cup \{66400,\ldots,66433\}$$

so $N = \binom{10}{5} + \binom{9}{4} + \binom{6}{3} + \binom{5}{2}$. Its lower shadow is

$$\partial\widehat{P}_{N5} = \{0000,\ldots,5555\} \cup \{6000,\ldots,6555\} \cup \{6600,\ldots,6633\} \cup \{6640,\ldots,6643\},$$

of size $\binom{9}{4} + \binom{8}{3} + \binom{5}{2} + \binom{4}{1}$.

If the smallest element of Q_{N95} is 66433, we have

$$\widehat{Q}_{N95} = \{99999,\ldots,70000\} \cup \{66666,\ldots,66500\} \cup \{66444,\ldots,66440\} \cup \{66433\}$$

so $N = \big(\binom{13}{9}+\binom{12}{8}+\binom{11}{7}\big) + \big(\binom{8}{6}+\binom{7}{5}\big) + \binom{5}{4} + \binom{3}{3}$. Its upper shadow is

$$\partial\widehat{Q}_{N95} = \{999999,\ldots,700000\} \cup \{666666,\ldots,665000\} \cup \{664444,\ldots,664400\} \cup \{664333,\ldots,664330\},$$

of size $\big(\binom{14}{9}+\binom{13}{8}+\binom{12}{7}\big) + \big(\binom{9}{6}+\binom{8}{5}\big) + \binom{6}{4} + \binom{4}{3} = N + \kappa_9 N$. The size, t, of each combination is essentially irrelevant, as long as $N \le \binom{s+t}{t}$; for example, the smallest element of $\widehat{Q}_{N98}$ is 99966433 in the case we have considered.

81. (a) The derivative would have to be $\sum_{k>0} r_k(x)$, but that series diverges.

[Informally, the graph of $\tau(x)$ shows "pits" of relative magnitude 2^{-k} at all odd multiples of 2^{-k}. Takagi's original publication, in *Proc. Physico-Math. Soc. Japan* (2) **1** (1903), 176–177, has been translated into English in his *Collected Papers* (Iwanami Shoten, 1973).]

(b) Since $r_k(1-t) = (1)^{\lceil 2^k t\rceil}$ when $k > 0$, we have $\int_0^{1-x} r_k(t)\,dt = \int_x^1 r_k(1-u)\,du = -\int_x^1 r_k(u)\,du = \int_0^x r_k(u)\,du$. The second equation follows from the fact that $r_k(\frac{1}{2}t) = r_{k-1}(t)$. Part (d) shows that these two equations suffice to define $\tau(x)$ when x is rational.

(c) Since $\tau(2^{-a}x) = a2^{-a}x + 2^{-a}\tau(x)$ for $0 \le x \le 1$, we have $\tau(\epsilon) = a\epsilon + O(\epsilon)$ when $2^{-a-1} \le \epsilon \le 2^{-a}$. Therefore $\tau(\epsilon) = \epsilon \lg \frac{1}{\epsilon} + O(\epsilon)$ for $0 < \epsilon \le 1$.

(d) Suppose $0 \le p/q \le 1$. If $p/q \le 1/2$ we have $\tau(p/q) = p/q + \tau(2p/q)/2$; otherwise $\tau(p/q) = (q-p)/q + \tau(2(q-p)/q)/2$. Therefore we can assume that q is odd. When q is odd, let $p' = p/2$ when p is even, $p' = (q-p)/2$ when p is odd. Then $\tau(p/q) = 2\tau(p'/q) - 2p'/q$ for $0 < p < q$; this system of $q-1$ equations has a unique solution. For example, the values for $q = 3, 4, 5, 6, 7$ are $2/3,\ 2/3;\ 1/2,\ 1/2,\ 1/2;\ 8/15,\ 2/3,\ 2/3,\ 8/15;\ 1/2,\ 2/3,\ 1/2,\ 2/3,\ 1/2;\ 22/49,\ 30/49,\ 32/49,\ 32/49,\ 30/49,\ 22/49$.

(e) The solutions $< \frac{1}{2}$ are $x = \frac{1}{4},\ \frac{1}{4} - \frac{1}{16},\ \frac{1}{4} - \frac{1}{16} - \frac{1}{64},\ \frac{1}{4} - \frac{1}{16} - \frac{1}{64} - \frac{1}{256},\ \ldots,\ \frac{1}{3}$.

(f) The value $\frac{2}{3}$ is achieved for $x = \frac{1}{2} \pm \frac{1}{8} \pm \frac{1}{32} \pm \frac{1}{128} \pm \cdots$, an uncountable set.

82. Consider paths starting from 0 in the digraph

$$\begin{array}{ccccccccccccc}
0 & \leftarrow & 1 & \leftarrow & 2 & \leftarrow & 3 & \leftarrow & 4 & \leftarrow & 5 & \leftarrow & \cdots \\
\updownarrow & & \updownarrow & & \updownarrow & & \updownarrow & & \updownarrow & & \updownarrow & & \\
1 & \rightarrow & 2 & \rightarrow & 3 & \rightarrow & 4 & \rightarrow & 5 & \rightarrow & 6 & \rightarrow & \cdots
\end{array}$$

Compute an associated value v, starting with $v \leftarrow -p$; horizontal moves change $v \leftarrow 2v$, vertical moves from node a change $v \leftarrow 2(qa - v)$. The path stops if we reach a node twice with the same value v. Transitions are not allowed to upper nodes with $-v \le -q$

or $v \ge qa$; they are not allowed to lower nodes with $v \le 0$ or $v \ge q(a+1)$. These restrictions force most steps of the path. (Node a in the upper row means, "Solve $\tau(x) = ax - v/q$"; in the lower row it means, "Solve $\tau(x) = v/q - ax$.") Empirical tests suggest that, for all integers $q > p > 0$, at least one such path exists, and that all such paths are finite. The equation $\tau(x) = p/q$ then has solutions $x = x_0$ defined by the sequence x_0, x_1, x_2, ... where $x_{k+1} = \frac{1}{2}x_k$ on a horizontal step and $x_{k+1} = 1 - \frac{1}{2}x_k$ on a vertical step; these are all the solutions to $\tau(x) = p/q$ when $x < \frac{1}{2}$, when q is not a power of 2.

For example, this procedure establishes that $\tau(x) = \frac{1}{5}$ only when x or $1-x$ is $3459/87040$; there are, similarly, just two solutions to $\tau(x) = 3/5$, having denominator $2^{46}(2^{56}-1)/3$.

Moreover, it appears that all cycles in the digraph that pass through node 0 define values of p and q such that $\tau(x) = p/q$ has uncountably many solutions. Such values are, for example, $2/3$, $8/15$, $8/21$, corresponding to the cycles (01), (0121), (012321). The value $32/63$ corresponds to (012121) and also to (0121012345454321), as well as to two other paths that do not return to 0.

83. [Frankl, Matsumoto, Ruzsa, and Tokushige, *J. Combinatorial Theory* **A69** (1995), 125–148.] If $a \le b$ we have

$$\binom{2t-1-b}{t-a} \Big/ T = t^{\underline{a}}(t-1)^{\underline{b-a}}/(2t-1)^{\underline{b}} = 2^{-b}\big(1 + f(a,b)t^{-1} + O(b^4/t^2)\big),$$

where $f(a,b) = a(1+b) - a^2 - b(1+b)/4 = f(a+1,b) - b + 2a$. Therefore if N has the combinatorial representation (57), and if we set $n_j = 2t - 1 - b_j$, we have

$$\frac{t}{T}\Big(\kappa_t N - N\Big) = \frac{b_t}{2^{b_t}} + \frac{b_{t-1}-2}{2^{b_{t-1}}} + \frac{b_{t-2}-4}{2^{b_{t-2}}} + \cdots + \frac{O(\log t)^3}{t},$$

the terms being negligible when b_j exceeds $2\lg t$. And one can show that

$$\tau\Big(\sum_{j=0}^{l} 2^{-e_j}\Big) \;=\; \sum_{j=0}^{l} (e_j - 2j)2^{-e_j}.$$

84. $N - \lambda_{t-1}N$ has the same asymptotic form as $\kappa_t N - N$, by (63), since $\tau(x) = \tau(1-x)$. So does $2\mu_t N - N$, up to $O\big(T(\log t)^3/t^2\big)$, because $\binom{2t-1-b}{t-a} = 2\binom{2t-2-b}{t-a}\big(1 + O(\log t)/t\big)$ when $b < 2\lg t$.

85. $x \in X^{\circ\sim} \iff \bar{x} \notin X^{\circ} \iff \bar{x} \notin X$ or $\bar{x} \notin X + e_1$ or $\cdots$ or $\bar{x} \notin X + e_n \iff x \in X^{\sim}$ or $x \in X^{\sim} - e_1$ or $\cdots$ or $x \in X^{\sim} - e_n \iff x \in X^{\sim+}$.

86. All three are true, using the fact that $X \subseteq Y^{\circ}$ if and only if $X^{+} \subseteq Y$: (a) $X \subseteq Y^{\circ} \iff X^{\sim} \supseteq Y^{\circ\sim} = Y^{\sim+} \iff Y^{\sim} \subseteq X^{\sim\circ}$. (b) $X^{+} \subseteq X^{+} \implies X \subseteq X^{+\circ}$; hence $X^{\circ} \subseteq X^{\circ+\circ}$. Also $X^{\circ} \subseteq X^{\circ} \implies X^{\circ+} \subseteq X$; hence $X^{\circ+\circ} \subseteq X^{\circ}$. (c) $\alpha M \le N \iff S_M^{+} \subseteq S_N \iff S_M \subseteq S_N^{\circ} \iff M \le \beta N$.

87. If $\nu x < \nu y$ then $\nu(x - e_k) < \nu(y - e_j)$, so we can assume that $\nu x = \nu y$ and $x < y$. We must have $y_j > 0$; otherwise $\nu(y - e_j)$ would exceed $\nu(x - e_k)$. If $x_i = y_i$ for $1 \le i \le j$, clearly $k > j$ and $x - e_k \prec y - e_j$. Otherwise $x_i > y_i$ for some $i \le j$; again we have $x - e_k \prec y - e_j$, unless $x - e_k = y - e_j$.

88. From the table

j =	0	1	2	3	4	5	6	7	8	9	10	11
$e_j + e_1$ =	e_1	e_0	e_4	e_5	e_2	e_3	e_8	e_9	e_6	e_7	e_{11}	e_{10}
$e_j + e_2$ =	e_2	e_4	e_0	e_6	e_1	e_8	e_3	e_{10}	e_5	e_{11}	e_7	e_9
$e_j + e_3$ =	e_3	e_5	e_6	e_7	e_8	e_9	e_{10}	e_0	e_{11}	e_1	e_2	e_4

we find $(\alpha 0, \alpha 1, \ldots, \alpha 12) = (0, 4, 6, 7, 8, 9, 10, 11, 11, 12, 12, 12, 12)$; $(\beta 0, \beta 1, \ldots, \beta 12) = (0, 0, 0, 0, 1, 1, 2, 3, 4, 5, 6, 8, 12)$.

89. Let $Y = X^+$ and $Z = C_k X$, and let $N_a = \|X_k(a)\|$ for $0 \le a < m_k$. Then

$$\|Y\| = \sum_{a=0}^{m_k-1} \|Y_k(a)\| = \sum_{a=0}^{m_k-1} \big\|(X_k(a-1) + e_k) \cup (X_k(a) + E_k(0))\big\| \\ \ge \sum_{a=0}^{m_k-1} \max(N_{a-1}, \alpha N_a),$$

where $a - 1$ stands for $(a - 1) \bmod m_k$ and the α function comes from the $(n - 1)$-dimensional torus, because $\|X_k(a) + E_k(0)\| \ge \alpha N_a$ by induction. Also

$$\|Z^+\| = \sum_{a=0}^{m_k-1} \|Z_k^+(a)\| = \sum_{a=0}^{m_k-1} \big\|(Z_k(a-1) + e_k) \cup (Z_k(a) + E_k(0))\big\| \\ = \sum_{a=0}^{m_k-1} \max(N_{a-1}, \alpha N_a),$$

because both $Z_k(a - 1) + e_k$ and $Z_k(a) + E_k(0)$ are standard in $n - 1$ dimensions.

90. Let there be N_a points in row a of a totally compressed array, where row 0 is at the bottom; thus $l = N_{-1} \ge N_0 \ge \cdots \ge N_{m-1} \ge N_m = 0$. We show first that there is an optimum X for which the "bad" condition $N_a = N_{a+1}$ never occurs except when $N_a = 0$ or $N_a = l$. For if a is the smallest bad subscript, suppose $N_{a-1} > N_a = N_{a+1} = \cdots = N_{a+k} > N_{a+k+1}$. Then we can always decrease N_{a+k} by 1 and add 1 to some N_b for $b \le a$ without increasing $\|X^+\|$, except in cases where $k = 1$ and $N_{a+2} = N_{a+1} - 1$ and $N_b = N_a + a - b$ for $0 \le b \le a$. Exploring such cases further, we can find a subscript d such that $N_c = N_{a+1} + a + 1 - c > 0$ for $a < c < d$, and either $N_d = 0$ or $N_d < N_{d+1} - 1$. Then it is OK to decrease N_c by 1 for $a < c < d$ and increase N_b by 1 for $0 \le b < d - a - 1$. (It is important to note that if $N_d = 0$ we have $N_0 \ge d - 1$; hence $d = m$ implies $l = m$.)

Repeating such transformations until $N_a > N_{a+1}$ whenever $N_a \ne l$ and $N_{a+1} \ne 0$, we reach situation (86), and the proof can be completed as in the text.

91. Let $x + k$ denote the lexicographically smallest element of $T(m_1, \ldots, m_{n-1})$ that exceeds x and has weight $\nu x + k$, if any such element exists. For example, if $m_1 = m_2 = m_3 = 4$ and $x = 211$, we have $x+1 = 212$, $x+2 = 213$, $x+3 = 223$, $x+4 = 233$, $x+5 = 333$, and $x + 6$ does not exist; in general, $x + k + 1$ is obtained from $x + k$ by increasing the rightmost component that can be increased. If $x + k = (m_1 - 1, \ldots, m_{n-1} - 1)$, let us set $x + k + 1 = x + k$. Then if $S(k)$ is the set of all elements of $T(m_1, \ldots, m_{n-1})$ that are $\le x + k$, we have $S(k + 1) = S(k)^+$. Furthermore, the elements of S that end in a are those whose first $n - 1$ components are in $S(m - 1 - a)$.

The result of this exercise can be stated more intuitively: As we generate n-dimensional standard sets S_1, S_2, ..., the $(n - 1)$-dimensional standard sets on each layer become spreads of each other just after each point is added to layer $m - 1$. Similarly, they become cores of each other just before each point is added to layer 0.

92. (a) Suppose the parameters are $2 \le m'_1 \le m'_2 \le \cdots \le m'_n$ when sorted properly, and let k be minimal with $m_k \ne m'_k$. Then take $N = 1 + \mathrm{rank}(0, \ldots, 0, m'_k - 1, 0, \ldots, 0)$. (We must assume that $\min(m_1, \ldots, m_n) \ge 2$, since parameters equal to 1 can be placed anywhere.)

(b) Only in the proof for $n = 2$, buried inside the answer to exercise 90. That proof is incorporated by induction when n is larger.

93. Complementation reverses lexicographic order and changes ϱ to ∂.

94. For Theorem K, let $d = n - 1$ and $s_0 = \cdots = s_d = 1$. For Theorem M, let $d = s$ and $s_0 = \cdots = s_d = t + 1$.

95. In such a representation, N is the number of t-multicombinations of $\{s_0 \cdot 0, s_1 \cdot 1, s_2 \cdot 2, \dots\}$ that precede $n_t n_{t-1} \dots n_1$ in lexicographic order, because the generalized coefficient $\binom{S(n)}{t}$ counts the multicombinations whose leftmost component is $< n$.

If we truncate the representation by stopping at the rightmost nonzero term $\binom{S(n_v)}{v}$, we obtain a nice generalization of (6o):

$$\|\partial P_{N_t}\| = \binom{S(n_t)}{t-1} + \binom{S(n_{t-1})}{t-2} + \cdots + \binom{S(n_v)}{v-1}.$$

[See G. F. Clements, *J. Combinatorial Theory* **A37** (1984), 91–97. The inequalities $s_0 \ge s_1 \ge \cdots \ge s_d$ are needed for the validity of Corollary C, but not for the calculation of $\|\partial P_{N_t}\|$. Some terms $\binom{S(n_k)}{k}$ for $t \ge k > v$ may be zero. For example, when $N = 1$, $t = 4$, $s_0 = 3$, and $s_1 = 2$, we have $N = \binom{S(1)}{4} + \binom{S(1)}{3} = 0 + 1$.]

96. (a) The tetrahedron has four vertices, six edges, four faces: $(N_0, \dots, N_4) = (1, 4, 6, 4, 1)$. The octahedron, similarly, has $(N_0, \dots, N_6) = (1, 6, 8, 8, 0, 0, 0)$, and the icosahedron has $(N_0, \dots, N_{12}) = (1, 12, 30, 20, 0, \dots, 0)$. The hexahedron, aka the 3-cube, has eight vertices, 12 edges, and six square faces; perturbation breaks each square face into two triangles and introduces new edges, so we have $(N_0, \dots, N_8) = (1, 8, 18, 12, 0, \dots, 0)$. Finally, the perturbed pentagonal faces of the dodecahedron lead to $(N_0, \dots, N_{20}) = (1, 20, 54, 36, 0, \dots, 0)$.

(b) $\{210, 310\} \cup \{10, 20, 21, 30, 31\} \cup \{0, 1, 2, 3\} \cup \{\epsilon\}$.

(c) $0 \le N_t \le \binom{n}{t}$ for $0 \le t \le n$ and $N_{t-1} \ge \kappa_t N_t$ for $1 \le t \le n$. The second condition is equivalent to $\lambda_{t-1} N_{t-1} \ge N_t$ for $1 \le t \le n$, if we define $\lambda_0 1 = \infty$. These conditions are necessary for Theorem K, and sufficient if $A = \bigcup P_{N_t t}$.

(d) The complements of the elements not in a simplicial complex, namely the sets $\{\{0, \dots, n-1\} \setminus \alpha \mid \alpha \notin C\}$, form a simplicial complex. (We can also verify that the necessary and sufficient condition holds: $N_{t-1} \ge \kappa_t N_t \iff \lambda_{t-1} N_{t-1} \ge N_t \iff \kappa_{n-t+1} \overline{N}_{n-t+1} \le \overline{N}_{n-t}$, because $\kappa_{n-t} \overline{N}_{n-t+1} = \binom{n}{t} - \lambda_{t-1} N_{t-1}$ by exercise 93.)

(e) 00000 ↔ 14641; 10000 ↔ 14640; 11000 ↔ 14630; 12000 ↔ 14620; 13000 ↔ 14610; 14000 ↔ 14600; 12100 ↔ 14520; 13100 ↔ 14510; 14100 ↔ 14500; 13200 ↔ 14410; 14200 ↔ 14400; 13300 ↔ 14400; and the self-dual cases 14300, 13310.

97. The following procedure by S. Linusson [*Combinatorica* **19** (1999), 255–266], who considered also the more general problem for multisets, is considerably faster than a more obvious approach. Let $L(n, h, l)$ count feasible vectors with $N_t = \binom{n}{t}$ for $0 \le t \le l$, $N_{t+1} < \binom{n}{t+1}$, and $N_t = 0$ for $t > h$. Then $L(n, h, l) = 0$ unless $-1 \le l \le h \le n$; also $L(n, h, h) = L(n, h, -1) = 1$, and $L(n, n, l) = L(n, n-1, l)$ for $l < n$. When $n > h \ge l \ge 0$ we can compute $L(n, h, l) = \sum_{j=l}^{h} L(n-1, h, j) L(n-1, j-1, l-1)$, a recurrence that follows from Theorem K. (Each size vector corresponds to the complex $\bigcup P_{N_t t}$, with $L(n-1, h, j)$ representing combinations that do not contain the maximum element $n-1$ and $L(n-1, j-1, l-1)$ representing those that do.) Finally the grand total is $L(n) = \sum_{l=1}^{n} L(n, n, l)$.

We have $L(0)$, $L(1)$, $L(2)$, $\dots$ $= 2, 3, 5, 10, 26, 96, 553, 5461, 100709, 3718354, 289725509, \dots$; $L(100) \approx 3.2299 \times 10^{1842}$.

98. The maximal elements of a simplicial complex form a clutter; conversely, the combinations contained in elements of a clutter form a simplicial complex. Thus the two concepts are essentially equivalent.

(a) If $(M_0, M_1, \ldots, M_n)$ is the size vector of a clutter, then $(N_0, N_1, \ldots, N_n)$ is the size vector of a simplicial complex if $N_n = M_n$ and $N_t = M_t + \kappa_{t+1}N_{t+1}$ for $0 \le t < n$. Conversely, every such $(N_0, \ldots, N_n)$ yields an $(M_0, \ldots, M_n)$ if we use the lexicographically first N_t t-combinations. [G. F. Clements extended this result to general multisets in *Discrete Math.* **4** (1973), 123–128.]

(b) In the order of answer 96(e) they are 00000, 00001, 10000, 00040, 01000, 00030, 02000, 00120, 03000, 00310, 04000, 00600, 00100, 00020, 01100, 00210, 02100, 00500, 00200, 00110, 01200, 00400, 00300, 01010, 01300, 00010. Notice that $(M_0, \ldots, M_n)$ is feasible if and only if $(M_n, \ldots, M_0)$ is feasible, so we have a different sort of duality in this interpretation.

99. Represent A as a subset of $T(m_1, \ldots, m_n)$ as in the proof of Corollary C. Then the maximum value of νA is obtained when A consists of the N lexicographically smallest points $x_1 \ldots x_n$.

The proof starts by reducing to the case that A is compressed, in the sense that its t-multicombinations are $P_{\|A\cap T_t\|t}$ for each t. Then if y is the largest element $\in A$ and if x is the smallest element $\notin A$, we prove that $x < y$ implies $\nu x > \nu y$, hence $\nu(A \setminus \{y\} \cup \{x\}) > \nu A$. For if $\nu x = \nu y - k$ we could find an element of $\partial^k y$ that is greater than x, contradicting the assumption that A is compressed.

100. In general, $F(p) = N_0p^n + N_1p^{n-1}(1-p) + \cdots + N_n(1-p)^n$ when $f(x_1, \ldots, x_n)$ is satisfied by exactly N_t binary strings $x_1 \ldots x_n$ of weight t. Thus we find $G(p) = p^4 + 3p^3(1-p) + p^2(1-p)^2$; $H(p) = p^4 + p^3(1-p) + p^2(1-p)^2$.

(b) A monotone formula f is equivalent to a simplicial complex C under the correspondence $f(x_1, \ldots, x_n) = 1 \iff \{j-1 \mid x_j = 0\} \in C$. Therefore the functions $f(p)$ of monotone Boolean functions are those that satisfy the condition of exercise 96(c), and we obtain a suitable function by choosing the lexicographically last N_{n-t} t-combinations (which are complements of the first N_s s-combinations): $\{3210\}$, $\{321, 320, 310\}$, $\{32\}$ gives $f(w,x,y,z) = wxyz \vee xyz \vee wyz \vee wxz \vee yz = wxz \vee yz$.

M. P. Schützenberger observed that we can find the parameters N_t values easily from $f(p)$ by noting that $f\big(1/(1+u)\big) = (N_0 + N_1u + \cdots + N_nu^n)/(1+u)^n$. One can show that $H(p)$ is not equivalent to a monotone formula in any number of variables, because $(1+u+u^2)/(1+u)^4 = (N_0+N_1u+\cdots+N_nu^n)/(1+u)^n$ implies that $N_1 = n-3$, $N_2 = \binom{n-3}{2} + 1$, and $\kappa_2 N_2 = n-2$.

But the task of deciding this question is not so simple in general. For example, the function $(1 + 5u + 5u^2 + 5u^3)/(1+u)^5$ does not match any monotone formula in five variables, because $\kappa_3 5 = 7$; but it equals $(1 + 6u + 10u^2 + 10u^3 + 5u^4)/(1+u)^6$, which works fine with six.

101. (a) Choose N_t linearly independent polynomials of degree t in I; order their terms lexicographically, and take linear combinations so that the lexicographically smallest terms are distinct monomials. Let I' consist of all multiples of those monomials.

(b) Each monomial of degree t in I' is essentially a t-multicombination; for example, $x_1^3x_2x_5^4$ corresponds to 55552111. If M_t is the set of independent monomials for degree t, the ideal property is equivalent to saying that $M_{t+1} \supseteq \partial M_t$.

In the given example, $M_3 = \{x_0x_1^2\}$; $M_4 = \partial M_3 \cup \{x_0x_1x_2^2\}$; $M_5 = \partial M_4 \cup \{x_1x_2^4\}$, since $x_2^2(x_0x_1^2 - 2x_1x_2^2) - x_1(x_0x_1x_2^2) = -2x_1x_2^4$; and $M_{t+1} = \partial M_t$ thereafter.

(c) By Theorem M we can assume that $M_t = \widehat{Q}_{Mst}$. Let $N_t = \binom{n_{ts}}{s} + \cdots + \binom{n_{t2}}{2} + \binom{n_{t1}}{1}$, where $s+t \ge n_{ts} > \cdots > n_{t2} > n_{t1} \ge 0$; then $n_{ts} = s+t$ if and only if $n_{t(s-1)} = s-2, \ldots, n_{t1} = 0$. Furthermore we have

$$N_{t+1} \ge N_t + \kappa_s N_t = \binom{n_{ts} + [n_{ts} \ge s]}{s} + \cdots + \binom{n_{t2} + [n_{t2} \ge 2]}{2} + \binom{n_{t1} + [n_{t1} \ge 1]}{1}.$$

Therefore the sequence $(n_{ts} - t - \infty[n_{ts} < s], \ldots, n_{t2} - t - \infty[n_{t2} < 2], n_{t1} - t - \infty[n_{t1} < 1])$ is lexicographically nondecreasing as t increases, where we insert '$-\infty$' in components that have $n_{tj} = j-1$. Such a sequence cannot increase infinitely many times without exceeding the maximum value $(s, -\infty, \ldots, -\infty)$, by exercise 1.2.1–15(d).

102. Let P_{Nst} be the first N elements of a sequence determined as follows: For each binary string $x = x_{s+t-1} \ldots x_0$, in lexicographic order, write down $\binom{\nu x}{t}$ subcubes by changing t of the 1s to $*$s in all possible ways, in lexicographic order (considering $1 < *$). For example, if $x = 0101101$ and $t = 2$, we generate the subcubes 0101*0*, 010*10*, 010**01, 0*0110*, 0*01*01, 0*0*101.

[See B. Lindström, *Arkiv för Mat.* **8** (1971), 245–257; a generalization analogous to Corollary C appears in K. Engel, *Sperner Theory* (Cambridge Univ. Press, 1997), Theorem 8.1.1.]

103. The first N strings in cross order have the desired property. [T. N. Danh and D. E. Daykin, *J. London Math. Soc.* (2) **55** (1997), 417–426.]

Notes: Beginning with the observation that the "1-shadow" of the N lexicographically first strings of weight t (namely the strings obtained by deleting 1 bits only) consists of the first $\mu_t N$ strings of weight t, R. Ahlswede and N. Cai extended the Danh–Daykin theorem to allow insertion, deletion, and/or transposition of bits [*Combinatorica* **17** (1997), 11–29; *Applied Math. Letters* **11**,5 (1998), 121–126]. Uwe Leck has proved that no total ordering of *ternary strings* has the analogous minimum-shadow property [Preprint 98/6 (Univ. Rostock, 1998), 6 pages].

104. Every number must occur the same number of times in the cycle. Equivalently, $\binom{n-1}{t-1}$ must be a multiple of t. This necessary condition appears to be sufficient as well, provided that n is not too small with respect to t; but such a result may well be true yet impossible to prove. [See Chung, Graham, and Diaconis, *Discrete Math.* **110** (1992), 55–57.]

The next few exercises consider the cases $t = 2$ and $t = 3$, when elegant results are known. Similar but more complicated results have been derived for $t = 4$ and $t = 5$, and the case $t = 6$ has been partially resolved. The case $(n, t) = (12, 6)$ is currently the smallest for which the existence of a universal cycle is unknown.

105. Let the differences mod $(2m+1)$ be $1, 2, \ldots, m, 1, 2, \ldots, m, \ldots$, repeated $2m+1$ times; for example, the cycle for $m = 3$ is (013602561450346235124). This works because $1 + \cdots + m = \binom{m+1}{2}$ is relatively prime to $2m+1$. [*J. École Polytechnique* **4**, Cahier 10 (1810), 16–48.]

106. The seven doubles ⚅⚅, ⚅⚄, ..., ⚃⚃ can be inserted in 3^7 ways into any universal cycle of 3-combinations for $\{0, 1, 2, 3, 4, 5, 6\}$. The number of such universal cycles is the number of Eulerian circuits of the complete graph K_7, which can be shown to be 129,976,320 if we regard $(a_0 a_1 \ldots a_{20})$ as equivalent to $(a_1 \ldots a_{20} a_0)$ but not to the reverse-order cycle $(a_{20} \ldots a_1 a_0)$. So the answer is 284,258,211,840.

[This problem was first solved in 1859 by M. Reiss, whose method was so complicated that people doubted the result; see *Nouvelles Annales de Mathématiques* **8**

(1849), 74; **11** (1852), 115; *Annali di Matematica Pura ed Applicata* (2) **5** (1871–1873), 63–120. A considerably simpler solution, confirming Reiss's claim, was found by P. Jolivald and G. Tarry, who also enumerated the Eulerian circuits of K_9; see *Comptes Rendus Association Française pour l'Avancement des Sciences* **15**, part 2 (1886), 49–53; É. Lucas, *Récréations Mathématiques* **4** (1894), 123–151. Brendan D. McKay and Robert W. Robinson found an approach that is better still, enabling them to continue the enumeration through K_{21} by using the fact that the number of circuits is

$$(m-1)!^{2m+1}\ [z_0^{2m} z_1^{2m-2} \dots z_{2m}^{2m-2}]\ \det(a_{jk}) \prod_{1\le j<k\le 2m} (z_j^2 + z_k^2),$$

where $a_{jk} = -1/(z_j^2 + z_k^2)$ when $j \ne k$; $a_{jj} = -1/(2z_j^2) + \sum_{0\le k\le 2m} 1/(z_j^2 + z_k^2)$; see *Combinatorics, Probability, and Computing* **7** (1998), 437–449.]

C. Flye Sainte-Marie, in L'*Intermédiaire des Mathématiciens* **1** (1894), 164–165, noted that the Eulerian circuits of K_7 include 2×720 that have 7-fold symmetry under permutation of $\{0, 1, \dots, 6\}$ (namely Poinsot's cycle and its reverse), plus 32×1680 with 3-fold symmetry, plus 25778×5040 cycles that are asymmetric.

107. No solution is possible for $n < 7$, except in the trivial case $n = 4$. When $n = 7$ there are $12{,}255{,}208 \times 7!$ universal cycles, not considering $(a_0 a_1 \dots a_{34})$ to be the same as $(a_1 \dots a_{34} a_0)$, including cases with 5-fold symmetry like the example cycle in exercise 104.

When $n \ge 8$ we can proceed systematically as suggested by B. Jackson in *Discrete Math.* **117** (1993), 141–150; see also G. Hurlbert, *SIAM J. Disc. Math.* **7** (1994), 598–604: Put each 3-combination into the "standard cyclic order" $c_1 c_2 c_3$ where $c_2 = (c_1 + \delta) \bmod n$, $c_3 = (c_2 + \delta') \bmod n$, $0 < \delta, \delta' < n/2$, and either $\delta = \delta'$ or $\max(\delta, \delta') < n - \delta - \delta' \ne (n-1)/2$ or $(1 < \delta < n/4$ and $\delta' = (n-1)/2)$ or $(\delta = (n-1)/2$ and $1 < \delta' < n/4)$. For example, when $n = 8$ the allowable values of (δ, δ') are $(1,1)$, $(1,2)$, $(1,3)$, $(2,1)$, $(2,2)$, $(3,1)$, $(3,3)$; when $n = 11$ they are $(1,1)$, $(1,2)$, $(1,3)$, $(1,4)$, $(2,1)$, $(2,2)$, $(2,3)$, $(2,5)$, $(3,1)$, $(3,2)$, $(3,3)$, $(4,1)$, $(4,4)$, $(5,2)$, $(5,5)$. Then construct the digraph with vertices (c, δ) for $0 \le c < n$ and $1 < \delta < n/2$, and with arcs $(c_1, \delta) \to (c_2, \delta')$ for every combination $c_1 c_2 c_3$ in standard cyclic order. This digraph is connected and balanced, so it has an Eulerian circuit by Theorem 2.3.4.2D. (The peculiar rules about $(n-1)/2$ make the digraph connected when n is odd. The Eulerian circuit can be chosen to have n-fold symmetry when $n = 8$, but not when $n = 12$.)

108. When $n = 1$ the cycle (000) is trivial; when $n = 2$ there is no cycle; and there are essentially only two when $n = 4$, namely (0001112223330202 1313) and (0001112020333222 1313). When $n \ge 5$, let the multicombination $d_1 d_2 d_3$ be in standard cyclic order if $d_2 = (d_1 + \delta - 1) \bmod n$, $d_3 = (d_2 + \delta' - 1) \bmod n$, and (δ, δ') is allowable for $n + 3$ in the previous answer. Construct the digraph with vertices (d, δ) for $0 \le d < n$ and $1 < \delta < (n+3)/2$, and with arcs $(d_1, \delta) \to (d_2, \delta')$ for every multicombination $d_1 d_2 d_3$ in standard cyclic order; then find an Eulerian circuit.

Perhaps a universal cycle of t-multicombinations exists for $\{0, 1, \dots, n-1\}$ if and only if a universal cycle of t-combinations exists for $\{0, 1, \dots, n+t-1\}$.

109. A nice way to check for runs is to compute the numbers $b(S) = \sum\{2^{p(c)} \mid c \in S\}$ where $\big(p(\mathtt{A}), \dots, p(\mathtt{K})\big) = (1, \dots, 13)$; then set $l \leftarrow b(S) \wedge -b(S)$ and check that $b(S) + l = l \ll s$, and also that $\big((l \ll s) \vee (l \gg 1)\big) \wedge a = 0$, where $a = 2^{p(c_1)} \vee \dots \vee 2^{p(c_5)}$. The values of $b(S)$ and $\sum\{v(c) \mid c \in S\}$ are easily maintained as S runs through all 31 nonempty subsets in Gray-code order. The answers are (1009008, 99792, 2813796, 505008, 2855676, 697508, 1800268, 751324, 1137236, 361224, 388740, 51680, 317340,

19656, 90100, 9168, 58248, 11196, 2708, 0, 8068, 2496, 444, 356, 3680, 0, 0, 0, 76, 4) for $x = (0, \ldots, 29)$; thus the mean score is ≈ 4.769 and the variance is ≈ 9.768.

Note: A four-card flush is not allowed in the "crib." Then the distribution is a bit easier to compute, and it turns out to be (1022208, 99792, 2839800, 508908, 2868960, 703496, 1787176, 755320, 1118336, 358368, 378240, 43880, 310956, 16548, 88132, 9072, 57288, 11196, 2264, 0, 7828, 2472, 444, 356, 3680, 0, 0, 0, 76, 4); the mean and variance decrease to approximately 4.735 and 9.667.

INDEX AND GLOSSARY

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information. An answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.